A Framework for Managing Schema Versioning in Object-Oriented Databases

Erik Odberg*
Division of Computer Science
Norwegian Institute of Technology

Abstract

An approach to schema modification management is presented which also takes into concern that alternative schema perspectives seems to be needed, in addition to revision-like changes. The primary goal is to support change transparency when the schema is modified, such that existing application programs and objects need not be affected by changes to the schema.

The approach is based on explicitly distinguishing Type (external interface of objects) and Class (object implementation and representation) - a Type is implemented by one or more Classes. Structural (representation) changes are accomplished by introducing new Classes (or Class versions), which does not require recompilation. Behavioral (Type) changes affect applications, as there may be incompatibilities between the Type version of the database object, and the application. The approach is based on automatically maintaining the totality of all versions’ interface for each Type, and provide mechanisms for defining new Class revisions to also implement this interface. In this way Type version incompatibilities are externally invisible. The paper presents the general mechanisms for managing schema changes, and identifies how the mechanisms are utilized.

1 Introduction and Motivation

Object-Oriented Databases (OODBs) are typically found most applicable in engineering and design oriented application areas, characterized by exploratory programming, unstable and evolving database specifications and requirements for flexibility. Some programming languages approach the need for flexibility by (partially) departing from the statically typed object model. Using prototypes [1] there is no type hierarchy - objects are created by cloning other objects to support sharing of object behavior. In [2] objects may be extended beyond the type-level specification, and thus achieve anomalous behavior. However, in a persistent environment we are also concerned about safety and efficiency, which indicates that some structure, represented by types, should be imposed.

According to Plato the world consists of concepts, which are at a higher level than the human perception of these. Different humans may have different perceptions of the same concept, and perceptions may evolve over time. A database schema usually is a perception (a model) of concepts, and thus naturally may evolve and have different appearances. Phenomena are instances of some concept, perceived by a human. The fact that a phenomenon is defined (created in the database) through one perception does not imply that it is invisible for humans with different perceptions. The phenomenon is, however, perceived differently.

Consequently, to have the database model reflect this cognitive model, it must be possible to change the database schema (define another or evolving perception), to have applications comply with different versions of the schema (have a perception) and to create instances of types as of this schema version (define a phenomenon). In addition there is a need to support for transparency for object access between the different schema versions (view phenomena through different perceptions). Thus, it is not possible to restrict schema changes to linear evolution (revisions) only, there is a need to support different variants of the schema to coexist. In this paper we propose mechanisms for the management of different schema versions, as well as indicating how these mechanisms are utilized.

1.1 Existing Approaches

Most existing approaches support a Schema evolution approach, in which existing specifications may be changed, but existing instances and applications must be made to comply with the new definition. Existing instances may be physically converted, either eagerly (GemStone [3]) or lazily (LISPO2 [4]), or they may be made to emulate the new specification (ORION [5]), in such a way that they behave as if they were instances of the new specifications.

This approach is problematic as existing applications and instances must be modified to correspond with the new specification. This may not be possible, may be time-consuming, may involve information loss or may be undesirable (according to the model in Section 1).

Schema versioning allows different versions of a schema definition to coexist, having instances and applications complying with different versions. An important goal is to support type change transparency [6]: Schema changes should be transparent for existing objects and applications - all objects should be transparently accessible irrespective of their version. Obviously, this requires management of possible schema version incompatibilities.

Encore [6] uses special error handler functions to handle requests for properties defined in some version of the type, but not for all. Request for a property not provided by the object (due to incompatibilities between the type version held by the dereferenced object and the one seen by the dereferencing application) implies that this error handler is invoked. Error handlers are manually implemented upon change, and may perform any computation. AVANCE [7] maintains substitute functions on attributes level, in principle between all versions. The Multiple view approach [8] allows for making new layers to add new properties (or hide old ones), and which may be implemented on-top of lower layers. Old representation is never obsoleted, and must be maintained even if newer versions are referenced.

* Detailed address: Division of Computer Science, Norwegian Institute of Technology (NTNU), N-7034 Trondheim-NTNU, Norway. Phone: +47 7 594484 Fax: +47 7 594460. Email: erik@idt.unio.no

1 LISPO2 also provides support for application modification and recompilation.
2 The Data Model: Types and Classes

For the discussions in this paper we assume as a basis a "classical" object-oriented data model based on a statically typed language as C++. The model distinguishes between the two explicit constructs Type and Class:

- Types describe interface - the externally accessible methods which the instances of this Type respond to. Multiple subtyping is supported - a Type may specialize one or more other Types by inheriting their interface, and add new ones locally. Subtyping is strict (additive only) and explicitly specified - the subtyping relationship also specifies substitutability. For each Type the Extent is a logical container of all instances of this Type (including subtypes).

- Classes implement Types, by providing implementation for the methods defined on the Type and appropriate representation (attributes) - commonly referred to as properties. A Class may inherit properties from one or more other Classes, multiple subclassing is supported. A subclass may exclude inherited properties, properties may be renamed and reimplemented. Subclassing is based on sharing - inheriting a method implies inheriting the representation as access functions to set/get the value. Thus, superclass representation can be modified without affecting inheriting Classes, as long as abstract representation is maintained.

Representation may be arbitrarily complex, and methods private to the Class may be specified. Each Class is explicitly associated with one (or more) Types - the association is valid if the Class at least implements the methods defined for a Type. A Type may be implemented by any number of Classes, and the Class hierarchy need not parallel the Type hierarchy.

In this way concept is distinguished from implementation, and concept specialization (subtyping) from implementation/representation inheritance (subclassing). An Abstract Type is a Type which is not implemented by any Class, and thus not instanciable. It serves as a conceptual abstraction only, and have an Extent. An Abstract Class is a Class which do not implement any Type, but may be inherited by other Classes. This constitutes implementation for reuse.

Figure 1 illustrates a possible Type/Class structure, with multiple Classes C_e and C_e1 implementing Type C_t. C_e implementing both C_t and D_t, A_t and B_t are abstract Types. E.e an abstract Class, C_e is defined through multiple subclassing (from E_e, C_e and C_e1) and D_t through multiple subtyping (from A_t and B_t).

Applications perceive database objects as instances of Types - object references have a Type. This is only specified upon creation of new objects, explicitly as a parameter to new or implicitly, to define object structure and implementation. Dereferencing a persistent reference returns an object with implementation and representation as specified upon creation.

3 The Approach

The approach adopts the schema versioning model. Existing applications and objects should be as unaffected as possible by changes to the schema. Transparency of change is the prime goal. Changes to Types and Classes are regarded in the context of other schema modifications - Hierarchy restructuring and additions/removals may create inter-Type/inter-Class dependencies, i.e., the unit of change is the schema, reflected in changes to the individual Types and Classes. Applications (statically) associate with one schema version, defining the view of Type definitions, and which Classes are visible for the creation of objects. Changing schema version association generally implies modification and recomposition.

3.1 The Modifications and the Problems

Different aspects of change must be managed in this framework. The problems to be resolved relate to maintenance of behavioral and structural/implementational consistency:

- New Classes are introduced as part of a new schema version. Only applications associating with such a schema version may create instances with this structure/implementation. Associating Class to Type is treated the same way. Class deletions are also part of a new schema version, and make the Class unavailable for applications associating with this schema version. Deleting associations of Class to a Type is treated the same way.

   However, applications must be able to dereference objects irrespective of their Class, even if this Class is not visible to the application programmer. We call this Transparent Object Dereferencing, and which may imply that Classes must be dynamically linked to the application (see Section 3.5, though).

- Class modifications give rise to Class versions, which can be a revision (making the former version obsolete) or a variant (which coexist with the former version(s)). Class variants associate with schema versions, i.e. different schema versions may contain different variants of the same Class, and this decides the structure/implementation of generated objects. When objects are dereferenced, its creation Class version applies according to the Transparent Object Dereferencing principle. A maximum of one variant of each Class may exist in each schema version.

---

2 For each attribute access functions are automatically created to get and set the value. This preserves protection and data independence and the functions may be reimplemented in subclasses. Normal access is perceived by programmers through special preprocessing.

3 Note that only Class names are visible, internal structure and implementation is invisible for applications.