Maintaining Consistency in a Database with Changing Types

Stanley B. Zdonik
Brown University
Department of Computer Science,
Providence, RI 02912
sbz%brown@csnet-relay

Abstract

In this work, we address the problem of maintaining consistency between a set of persistent objects and a set of type definitions that can change. We cast this work in the context of an object-oriented database system. The solution involves the use of a version control mechanism and a set of error handlers associated with the versions of a type. We describe the structure of this error handling mechanism and demonstrate how it can be used to address this problem.

1. Introduction

Our research is focused on developing techniques to allow database technology to be used for advanced applications on high-performance workstations. A large class of applications of interest include those that involve design. This includes programming environments, computer-aided design, logical database design, and office information systems. We have been designing database tools that are well-suited for this environment.

Design environments are characterized by constant change. Traditional database tools have problems in dealing with certain kinds of change. In particular, changing the database schema in arbitrary ways is a very difficult process. We feel that it is important for a design database to be able to deal with change at all levels.

In this work we look at the problem of change to the type definitions (i.e., the database schema). It is natural to assume that during a design, views of the world (i.e., type definitions) will change. A database stores objects for long periods of time. Each object was created as an instance of some type at some point in that type's evolution. The type described all of the assumptions about that objects behavior. What happens when that type definition changes? Old objects might become incompatible with new types. Also, new objects might become incompatible with old types for which programs have already been written.

2. The Database Model

The database system that forms the basis for this work [ZW] supports an object-oriented model of data. It is in the tradition of much of the work on high-level semantic models [Ch,Co,HM,MBW,Sh,SFL], but it takes a view of data that is very closely aligned with many of the object-oriented programming languages [BS,Fl,GR]. It illustrates a new direction in database research characterized as object-oriented databases [CM,DGL,MS,MSOP,Ru,Zd1].

All objects are instances of some *type* which describes the behavior of its instances. A type T is a specification of behavior. As such, it describes a set of operations O, a set of properties P, and a set of constraints C that pertain to any of the instances of T. If x is an instance of T, any operation o in O can legally be applied to x, any property p in P is defined for x and and constraint c in C must be satisfied for x.

Types, operations, and properties are all objects in their own right and as such have a type that describes their behavior. Operations are active things that are supported by code. All operations have an *invoke* operation defined for them such that it is possible to invoke an operation defined on type T on any object of type T. Properties are objects [Zd2] that are used to relate other objects. For example, a property called *works-for* might be defined on the type *Person*. Works-for would relate a given person to the company object for which he or she works. As a first-class object, it is possible for properties to have properties. A common constraint on property types limits the acceptable values for the property. We will call the set of all legal values for a property p its *value class* and notate it as VC(p).

Types can be related to each other by means of a special property called *is-a*. The is-a property induces an inheritance relationship between types. If A is-a B, then all operations, properties, and constraints that are defined on B will also be defined on A. In this case, we say that A is a *subtype* of B and that B is a *supertype* of A. The system supports the ability for a type to have more than one supertype (i.e., multiple inheritance).

Each type has an implementation that is hidden. The implementation of a type includes a representation for instances and some code that implements the operations and properties. No code outside of this type definition has access to the implementation of any other type. Type definitions may only use the exported interface of other types. This includes a type and its subtypes. No subtype can make use of the implementation of any of its supertypes, and no supertype can make use of the implementation of any of its subtypes.

It is important to realize that the model of data described above is a part of a database system. As such, it governs the way in which persistent, sharable objects behave. Our system also addresses database notions of transaction, consistency, associative retrieval, and views.

3. The Problem Framework

We strive to provide *type change transparency*. Assume that T_1 and T_2 are two versions of type T. If f is an operation defined on T_2, f(x) should be well-defined for all x such that x is an instance of T (i.e., x is an instance of some version of T). Type change transparency is particularly useful for iterators that are to range over all instances of some type T, independently of time. A common example of this is in associative retrieval in which we are interested in all T's such that some predicate is satisfied.

Our approach allows type designers to add error handlers to each version of a type definition. These handlers allow each version of a type to handle behaviors that were not covered in its original form. A type designer adds handlers to other versions of a type whenever a new version is created. We also supply a methodology to assist a type designer in understanding which handlers need to be written when a new type version is created.

The kinds of changes to a type that we envision handling include:

- Inserting a new type into the type hierarchy (thereby changing the inheritance of it's subtypes).
- Deleting an existing type from the type hierarchy.
- Moving a type to a new position in the type lattice.
- Adding or deleting properties, operations, or constraints defined by a type.
- Modifying properties, operations, or constraints defined by a type.

For the purposes of this paper, we will simplify the problem by only considering reading and writing values to properties. Further, we will ignore subtyping. This restricts our attention to the last two items in the above list.

An alternative approach to this problem might be to try to update all objects to be consistent with some current version of the type. Although there are situations in which this is reasonable, we have chosen to reject it as a general solution for the following reasons. First, it might not be practical. If there are a large number of objects, the conversion might be very expensive. Examples of this can be seen in current database practise where to convert millions of records might involve several hours of processing. Second, it might not be possible. If the information held in one of the type versions is significantly different from that held in another, conversion might require making guesses for values (generating information) or discarding values (destroying information) that might be useful later. Third, it might not be desirable. If there are old programs that must operate with instances of old type versions, these programs would become inoperative if we converted the instances that they need. In many environments, converting all existing programs to be compatible with the new type definitions is impracticle.

3.1. The Reader's Problem

We separate the problem for property changes into two broad categories. The first category will be called the reader's problem and concerns what might happen to programs that read values from properties of objects that were defined in versions of a type other than the one that they are expecting.

When a type change strengthens a constraint on a property p, its domain of values becomes narrower. A program P written in terms of a new type T_j definition may read an *unknown* value. That is, P might receive a value from an object x that was created under a type version T_i that is not contained in the value class for p as defined in T_j.

Similarly, when a type change relaxes a constraint on a property, its domain of values becomes wider. A program P written in terms of an old type definition might read an *unknown* value.

3.2. The Writer's Problem

The second category will be called the writer's problem. It occurs when a program tries to write a property of an object that is of another type version than the one the program assumes. When a constraint is relaxed by a type change, programs using the new type definition may try to write an illegal value to an object created before the change. A program P may try write a value v to property p of an object x. Suppose that x is an instance of T_i. If v is defined to be in the value set of property p within the definition of T_j and v is not defined to be in the value set for of p within the definition of T_i, then program P which is written assuming instances of T_j could write v to x.p (the p property of x).

Similarly, when a constraint is strengthened, old programs applied to new objects fail if they attempt to write a value which is outside the new domain.

This behavior can also occur when a program expects an instance of a type S and it receives an instance of one of S's subtypes T. If T has refined a property defined by S, this will be analogous an old program's operating on an instance of a new type whose change involved strengthening a constraint. We point out this similarity here solely for interest since we are not considering the case of inheritance in this paper.

4. The Version Model

Our database system includes a version control mechanism that is built on top of the basic system kernel [Zd3]. Much work has been done recently on incorporating time and version histories into databases [KL,LDEGPWWW,Zd3]. For this discussion, it is sufficient to understand that versions of an object are collected into aggregates called *version sets.* A version set can only be added to at one end. All previous versions are read-only. It is possible to have a version set for which the ordering of versions is a partial order. That is, a version might have several successors (called alternatives). This corresponds to the case in which there are two or more competing versions in a design. They must coexist as equal latest versions. Each of these alternatives may then have subsequent independent version histories of their own.

Each change to a type is recorded as a new version in a version set for the named type. Each instance of a type is linked to the version under which it was created. For operations on single objects, this scheme is sufficient. If x is an instance of the ith version of T, T_i, then an operation f applied to x (i.e., f(x)) will be interpreted by looking in the T_i type definition to find the definition for f. If it is defined there, then the invokation proceeds, and if it is not an error exception is raised.

Beyond this, each type carries a set of special exception handlers to deal with deficient cases represented by the reader's problem and the writer's problem. These handlers will be described below and constitute the framework needed to deal with the problem of changing types for properties.

We also define the *version set interface* to be the most general interface to a type. It is constructed as the conjunction of the type definitions for all versions of a type T. A type T is a triple (op(T), pr(T), C(T)) where op(T) is a set of operations, pr(T) is a set of properties, and c(T) is a set of constraints. If a type T has n versions $T_1 \dots T_n$, then the version set interface V which has the form of a type definition is defined by:

$$\begin{aligned} V = (op(V), pr(V), c(V)) = \\ &(op(T_1) \cup op(T_2) \cup \dots \cup op(T_n), \\ &(pr(T_1) \cup pr(T_2) \cup \dots \cup pr(T_n), \\ &(c(T_1) \cup c(T_2) \cup \dots \cup c(T_n)) \end{aligned}$$

It contains all properties and operations ever defined by some version of T and every value ever declared valid for property and operation parameters. If only one version of a type exists, the version set interface for that type is equivalent to the single type definition. The version set interface is useful in determining which handlers need to be created for a given type change. This process will be sketched at the end of the next section.

Our general approach to the problem of changing types adopts a philosophy that type designers are basically good citizens. They modify type definitions because the design process requires it. They know that this is a dangerous process, and they would be willing to do extra work if they knew that that work would help to avert disaster. Currently, this extra work is very unstructured. It is not clear what needs to be done when a type changes. Our solution gives type designers a framework for this task. It provides them with a structured approach to defining the correspondences between new types and old types.

5. Type Version Handlers

Each type version has three sets of handlers. A handler is a piece of code that gets control when certain undefined exceptions are raised by the get-property-value and the set-property-value operations. This code can do arbitrary things, but it often does a mapping from one property value to another. Two sets of handlers are concerned with reading property values, and the other is concerned with writing values. For reading values, a program could have its own handlers to deal with unanticipated cases.

For example, we might have a piece of code that tries to read the value of the color property of car, c.

```
y := get-property-value (c, color)
    except when undefined (y := black);
```

In this case, if the exception *undefined* is raised by the get-property-value operation, the program fragment opts to handle it by assuming that the color is black. If the code does not handle an exception explicitly (as above), then *undefined* exceptions on reading property values will be handled by the type version handlers.

We will first describe the handlers that are required for properties whose constraints have changed from one version of a type to the next. Here there is both a read case and a write case. In the following, assume that P is a program, T_i and T_j are type versions of type T with version-set-interface V, x and y are objects, and p is a property. Also, assume that P is written using T_j and x is of type T_i. If p is defined by multiple versions of a type, we will indicate the value class of p for type version T_i as $VC[T_i](p)$. When there is no ambiguity about type versions we will leave out the extra qualification. A type version T_i has write handlers for the case in which P tries to execute set-prop-value (x,p,y) and the following conditions hold:

1. $p \in pr(V)$ and $y \in VC(p)$
2. $p \notin pr(T_i)$ or ($p \in pr(T_i)$ and $y \notin VC[T_i](p)$)

A type version T_i has read handlers for the case in which P tries to execute get-property-value (x,p) which would return y, and the following conditions hold:

1. $p \in pr(V)$ and $y \in VC(p)$
2. $p \in pr(T_i)$ and $y \notin VC[T_i](p)$

The preceding discussion indicates that every type version T_i has a set of read handlers $R(T_i)$ and a set of write handlers $W(T_i)$. If a program P is written with assumptions for type T from T_j and P receives an object x of type version T_i, then a read by P on x will use $R(T_j)$ and a write by P on x will use $W(T_i)$.

This solution is correct for the cases in which the property in question is defined on both T_i and T_j. The third kind of handler adds a further refinement for the read case. For properties that are *undefined* on an object but are in the version set interface for the type, the type version of the object should have a chance to supply a value. We add to the above, another set of read-handlers $R_u(T_i)$ to each type. If P tries to read property p from x, and p is undefined, then the handlers R_u(type-of (x)) will be used (where type-of returns the type version of its argument).

As was mentioned above, the version set interface can be used to help guide a type programmer in adding handlers to type version definitions. If a type version does not define a property that is defined in the version set interface, then an undefined-property handler must be written for that type. If a type version T_i defines a property but does not permit some value v in the value set from the version set interface, then there must be a write handler for T_i that covers the value v. There must also be a read handler to service programs that are written using T_i and that read from objects of another type version T_j.

6. Example

We will now illustrate some of the mechanisms described above by a simple example. Consider the following type definitions which describe two versions of the type Car:

```
Define Type Car_1
  Supertypes: Vehicle_1
  Properties:
    Color: {red, blue, black}
  Handlers:
    Read-Undefined-Property
      for gas-type return leaded
    Write
      for value(gas-type) = leaded
        return
      for value (gas-type) = unleaded
        raise incorrect-value-error

Define Type Car_2
  Supertypes: Vehicle_1
  Properties:
    Color: {black}
    Gas-type: {leaded, unleaded}
  Handlers:
    Read
      for value (color) = red
        return (black)
      for value (color) = blue
        return (black)
```

Car_1 is defined with a color property that can take on the values red, blue, or black. Car_2 refines this property by restricting its value to only black. In addition, Car_2 adds the gas-type property that was not present on Car_1.

The type Car_1 was initially defined with no handlers. When Car_2 was defined, the type designer added the handlers to Car_1 to handle the potential problem cases. Also, since Car_2 is to coexist with Car_1, the definition of Car_2 must include some handlers to deal with the problem cases that might occur if a program written with the assumptions of Car_2 encounters an instance of Car_1.

The *Read-Undefined-Property* handlers on Car_1 will take care of the case in which a program written using Car_2 tries to read the gas-type property from an instance of Car_1. In this case, the handler will return a value of leaded, under the assumption that all old cars that did not keep track of their gas type used leaded gas since that is all that was available.

The *Write* handlers for Car_1 handle gas-type symmetrically. If a program P written using Car_2 tries to write a value of leaded to the gas-type property of x, an instance of Car_1, the write will simply result in no action (i.e., a no-op). This is the converse of the assumption made by the read-undefined-property handler. If P tries to write a value of leaded to the gas type property of x, the handler raises an incorrect-value-error exception.

The *Read* handlers on Car_2 cover the case in which a program P using Car_2 encounters an instance of Car_1 which has a value of red or blue for its color property. These values would be undefined in the context of Car_2. Thus, the handlers have an opportunity to transform them in whatever way makes sense for the application. In this case, they treat red and blue cars as if they were black.

7. Summary

We have described the problem of changing types and managing the consistency of one type version with objects that might have been created under a different version. We have presented a solution that applies to the case of undefined properties and undefined property values.

An extention of this solution involves sketching similar structures for the problems of changing operations. We also need to discuss the question of how introducing the type hierarchy into the problem description affects the final result. Another class of problem is concerned with arbitrary relinking of the is-a hierarchy.

We are also in the process of building a database designers workbench to assist type definers in writing type definitions and adding handlers. These handlers might be added to pre-existing type versions as well as the current changed version. This environment would be able to prompt type designers for the handlers that they must create whenever they modify a type definition.

8. References

[ABBHS] M. Ahlsen, A. Bjornerstedt, S. Britts, C. Hulten, L. Soderlund, "Making Type Changes Transparent", University of Stockholm, SYSLAB Report No. 22, February, 1984.

[Bo] A. Borgida, "Language Features for Flexible Handling of Exceptions in Information Systems", ACM Transactions on Database Systems, Vol. 10, No. 4, December, 1985.

[BS] D.G. Bobrow, M. Stefik, "The LOOPS Manual", Xerox Corporation, 1983.

[Ch] P.P.S. Chen, "The Entity-Relationship Model: Towards a Unified View of Data", ACM TODS 1, 1, March 1976.

[CM] G. Copeland and D. Maier, "Making Smalltalk a Database System", Proceedings of the ACM SIGMOD, Boston, Mass., June, 1984.

[Co] E.F. Codd, "Extending the Database Relational Model to Capture More Meaning". ACM Transactions on Database Systems 4, 4 (December 1979), 397-434.

[DGL] K. Dittrich, W. Gotthard, P.C. Lockemann, "DAMOKLES - A Database System for Software Engineering Environments", Proceedings of the IFIP 2.4 Workshop on Advanced Programming Environments, Trondheim, Norway, June, 1986.

[Fl] "Introduction to the Flavor System", in Reference Guide to Symbolics-Lisp, Symbolics Inc., 1985.

[GR] A. Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[HM] M. Hammer, D. McLeod, "Database Description with SDM: A Semantic Database Model", ACM TODS 6, 3, September 1981, 351-387.

[HR] H.B. Hunt, D.J. Rosenkrantz, "The Complexity of Testing Predicate Locks", Proceedings of the ACM SIGMOD, Boston, Mass., May-June, 1979.

[KL] R. Katz and T. Lehman, "Storage Structures for Versions and Alternatives", Computer Science Department, University of Wisconsin - Madison, Technical Report No. 479, July, 1982.

[LDEGPWWW] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, J. Woodfill, "Designing DBMS Support for the Temporal Dimension", Proceedings of the ACM SIGMOD, Boston, Mass., June, 1984.

[MBW] J. Mylopoulos, P.A. Bernstein, H.K.T. Wong, "A Language Facility for Designing Database-Intensive Applications", ACM Transactions on Database Systems, Vol 5, No. 2, June, 1980, pages 185-207.

[MS] D. Maier, J. Stein, "Indexing in an Object-Oriented DBMS", Technical Report CS/E-86-006, Oregon Graduate Center, Beaverton, OR, May, 1986.

[MSOP] D. Maier, J. Stein, A. Otis, A. Purdy, "Development of an Object-Oriented DBMS", Technical Report CS/E-86-005, Oregon Graduate Center, Beaverton, OR, April, 1986.

[Ru] A. Rudmik, "Choosing an Environment Data Model", Proceedings of the IFIP 2.4 Workshop on Advanced Programming Environments, Trondheim, Norway, June, 1986.

[Sh] D.W. Shipman, "The Functional Data Model and the Data Language DAPLEX", ACM TODS 6, 1 (1981), 140-173.

[SFL] J.M. Smith, S. Fox, and T. Landers, "ADAPLEX: Rational and Reference Manual", second edition, Computer Corporation of America, Cambridge, Mass., 1983.

[SS] J.M. Smith, D.C.P. Smith, "Database Abstractions: Aggregation", CACM 20, 6 (1977).

[WMy] H.K.T. Wong, J. Mylopoulos, "Two Views of Data Semantics: A Survey of Data Models in Artificial Intelligence and Database Management", INFOR 15, 3 (1977), 344-382.

[Zd1] S.B. Zdonik, "Object Mangement System Concepts", Proceedings of the Second ACM-SIGOA Conference on Office Information Systems, Toronto, Canada, June, 1984.

[Zd2] S.B. Zdonik, "Version Management in an Object-Oriented Database", Proceedings of the IFIP 2.4 Workshop on Advanced Programming Environments, Trondheim, Norway, June, 1986.

[Zd3] S.B. Zdonik, "Why Properties are Objects or Some Refinements to is-a", Proceedings of the ACM/IEEE Fall Joint Computer Conference, Austin, Texas, November, 1986.

[ZW] S.B. Zdonik and P. Wegner, "Language and Methodology for Object-Oriented Database Environments", Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, January, 1986.