Beyond Schema Evolution to Database Reorganization

Barbara Staudt Lerner
A. Nico Habermann

Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213

Abstract

While the contents of databases can be easily changed, their organization is typically extremely rigid. Some databases relax the rigidity of database organization somewhat by supporting simple changes to individual schemas. As described in this paper, OTGen supports not only more complex schema changes, but also database reorganization. A database administrator uses a declarative notation to describe mappings between objects created with old versions of schemas and their corresponding representations using new versions. OTGen generates a transformer that applies the mappings to update the database to the new definitions, thus facilitating improvements in performance, functionality, and usability of the database.[1]

[1]Support for research on Gandalf is provided in part by ZFE F2 KOM of Siemens Corporation, Munich, Germany and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order #4976, under contract F33615-87-C-1499 and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio, 45433-6543.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the US government, or other supporting institutions.

1. Introduction

Separation of interfaces from implementation is by now a generally accepted technique for localizing the effect of change. It gives the implementor the freedom of improving the code without affecting the client, while the latter can continue to use the specified interfaces and apply the provided operations as before. Only when client and implementor agree on a change in the specifications will it be necessary to modify the client's application.

However, changes in either specification or implementation not only affect programs, but also the representation of persistent data objects. The problem of modifying software is not limited to changing code, but also affects existing data objects that were generated with that software prior to modification. It is thus necessary to address the problem of how to adapt existing objects to the new requirements imposed by the modified software.

The problem of updating existing objects is well known to the software support groups that install new releases of an operating system. They rely heavily on their understanding of the impact of system modifications and bring existing data up to date with procedures of their own ad hoc invention.

Our work involves the automation of this updating process. Our basic approach is to apply a program generator to the delta of the data definitions (i.e., to the difference between the old and the new definitions), and let it produce the necessary programs and tables that can transform the existing data into the new formats.

Automation of this process has recently received some attention from the database world where this problem is particularly acute. Examples of projects addressing this issue are Orion [1] at MCC and GemStone [3] at Servio Logic. We share with these projects our basic approach as to the kind of data transformations that can be automated. However, our approach goes much further than any we know of in allowing the database administrator to reorganize the data. Reorganization is accomplished by editing the transformation tables with the assistance of our interactive OTGen environment. We found that this capability lifts the transformation technique from an interesting toy to a tool of great practical use.

Our experience in automatic database transformations goes back to 1986 when we developed TransformGen, a system for transforming tree-structured databases used in Gandalf programming environments [2, 5]. The TransformGen environment is still heavily used for editing abstract syntax specifications used to generate programming environments. When these specifications are modified, TransformGen simultaneously builds transformation tables that are later used to transform existing trees to the new format. An important next step in the process is that the developer can edit transformation tables to effect richer transformations, such as movement of data from one node to another, creation of new nodes, and context-dependent transformations. We showed in [5] that this implementation covers the whole range of changes a developer might want to perform on a database, and we showed that these changes are composable. Changes produced automatically by TransformGen are composed automatically with those changes already encoded in the table.

TransformGen has enabled us to upgrade our tools without hesitation. The ability to generate transformers easily has allowed us to develop and use early prototypes of systems without fear the data created will be unusable when the "real" system is released. This gradual evolution of a tool via repeated refinement can continue throughout a tool's lifetime, allowing the experiences gained from use of an early version to be applied to the redesign and upgrade of later versions, without losing the work done with the earlier versions of the tool.

Recently, we started looking into the possibility of replacing our tree-structured databases with more general object-oriented databases. However, current object-oriented databases lack the ability to redefine database structures and transform existing databases that we have been accustomed to with TransformGen. These considerations led to a redesign of TransformGen, resulting in our new system, called OTGen (Object Transformer Generator), that applies to object-oriented databases.

This paper starts with a discussion of the kind of modifications that can be handled automatically in an object-oriented database. It proceeds with a discussion of the important extensions that make the transformation process very practical.

2. Transforming Object-Oriented Databases

Because of the richness of the data structures that can be stored in an object-oriented database, some changes are required to adapt our tree-oriented transformer into one suitable for object-oriented databases. In this section, we describe our basic data model, the ways in which class definitions can be modified, the invariants that the transformation process must maintain, and the default transformation rules associated with the various class definition changes.

2.1. Basic Object-Oriented Data Model

We assume a fairly standard basic data model. *Classes* encapsulate typed *instance variables* and *methods*. *Objects* are created dynamically. Each object has a *type*. The type is defined by a class. Classes are organized into a lattice. Each class has at least one superclass, except for the special *Object* class, which is the top of the lattice. A class inherits instance variables and methods from its superclasses. In addition, a class may override an inherited variable or method, by defining one locally with the same name. To be more formal, we describe our data model using the following invariants, which are very similar to those used by Orion [1].

Class Lattice Invariant. The subclass-superclass relationship forms a lattice, of which the pre-defined class *Object* is the root.

Unique Name Invariant. Each instance variable and method defined or inherited by a class must have a unique name. Each class must have a unique name.

Full Inheritance Invariant. A class inherits the union of instance variables and methods from its superclasses, unless it defines an instance variable or method with the same name. If more than one superclass defines the same instance variable or method, the one inherited is the one defined by the superclass that appears earliest in the class's superclass list.[2]

Type Compatibility Invariant. If a class *t* defines an instance variable with the same name as an instance variable it would otherwise inherit from superclass *s*, the type of *t*'s variable must be a subclass of the type of *s*'s variable.

Typed Variable Invariant. The type of each instance variable must have a corresponding class in the class lattice.

Numerous extensions can be made to this basic model without compromising the capabilities of OTGen. Possible extensions include the addition of new type constructors (such as set and sequence), class variables, or components (to provide "part of" semantics).

To transform a database defined with one class lattice into another database with a revised class lattice, we must solve several problems. First, we need to identify the kinds of changes that can be made to a class. Second, we need to define a set of default transformations indicating how existing objects should be transformed for each kind of class change. Third, we need to provide a mechanism by which a database administrator can override the default transformations, yet is prevented from violating the rules imposed by the data model. The remainder of this paper describes how OTGen solves these problems. Briefly, OTGen assists the database administrator in understanding the effects of changes to class definitions, provides the default transformations, and provides the mechanism to allow overriding of the default transformations.

[2]The details of how conflicts due to multiple inheritance are resolved is really orthogonal to the point of this paper. Any resolution scheme would work equally well for the purposes of transformation, as long as it can be statically determined. For this reason, we have chosen the simplest possible resolution scheme to avoid the presentation of unnecessary details.

2.2. Effects of Class Changes

Inheritance is a basic property of object-oriented systems. A clear model of inheritance is important to understand the ramifications of changing a class. In a data model that does not support inheritance, any change to a type affects only that type. In a data model with inheritance, however, changes to a single class might affect all subclasses of the changed class.

Below we describe each kind of change that can be made to a class. For each change, the database administrator initiates the change by changing the class definition using OTGen. If the change would violate the class lattice invariant by adding cycles to the graph, the change is rejected because it would be impossible for OTGen to recompute inheritance. Otherwise, OTGen recomputes the inheritance of the affected classes. If any naming conflicts arise during this computation, they are brought to the database administrator's attention. In addition, the database administrator is warned if the class lattice becomes disconnected, or if the type compatibility or typed variable invariants are violated. After being told of the conflicts and violations that would occur, the database administrator is given the option of committing the change or aborting it. Temporary violations of the class lattice, type compatibility, and typed variable invariants are allowed. However, before a database transformer can be generated, all invariants must hold.

The changes supported by OTGen are:

- Adding an instance variable
- Deleting an instance variable
- Renaming an instance variable
- Changing the type of an instance variable
- Adding a superclass to a class's superclass list
- Deleting a superclass from a class's superclass list
- Adding a new class
- Deleting a class
- Renaming a class

For example, consider the effects of adding an instance variable, *v3*, to a class *C*.

```
Class C subclass of S
     v1:  T1
     v2:  T2
```

Before

```
Class C subclass of S
     v1:  T1
     v2:  T2
     v3:  T3
```

After

Let's consider each invariant in turn.

Class Lattice Invariant. Adding an instance variable does not affect the class lattice invariant.

Unique Name Invariant. The unique name invariant requires that there be no other instance variable with the same name in that class. Therefore, if the new variable had been named *v1* or *v2*, the unique name invariant would have been violated. If a variable with the name *v3* was inherited, it will no longer be inherited, because *C*'s definition of *v3* will now override the inherited definition.

Full Inheritance Invariant. To ensure that the full inheritance invariant still holds, we must examine each subclass of *C*. If a subclass defines an instance variable named *v3* locally, it is not inherited. If it does not define *v3*, and does not already inherit *v3*, it should inherit *C*'s *v3*. If it already inherited *v3* from *C* (because it was inherited by *C*), we need to inherit the new definition of *v3*. If *v3* was inherited from a different superclass, we need to determine which superclass the subclass should inherit *v3* from by examining the ordering of superclasses. In any event, if *C*'s definition of *v3* is inherited by a subclass, we need to repeat this process to propagate the inheritance to all of this subclass's subclasses.

Type Compatibility Invariant. After having re-established the full inheritance invariant, we can now determine if the type compatibility invariant is violated. If *C*'s *v3* is overriding another definition of *v3*, we must check that the type of *C*'s *v3* is a subclass of the *v3* it is overriding. Also, if any subclasses of *C* fail to inherit *v3* because they define their own *v3*, we must ensure that the type of the subclass's *v3* is a subtype of *C*'s *v3*.

Typed Variable Invariant. To guarantee the typed variable invariant, we require *v3*'s type to correspond to a class in the class lattice.

A similar analysis must be made of each change listed above.

2.3. Invariants on Transformation

The purpose of the transformation process itself is to change a database from using one version of a set of class definitions to another. We require transformation to maintain the following three invariants:

Completeness. When a new set of class definitions is released, each object must be transformed to its new definition before it can be manipulated further. This could be implemented using either lazy or eager transformation.

Correctness. After transformation, each object must correspond to a definition of a class in the new set of definitions. In particular, the value for each instance variable must correspond to the type of the instance variable in the corresponding class definition.

Sharing. If two instance variables, in the same or different objects, point to a single object before transformation, and if

- neither instance variable is deleted,
- neither instance variable's default transformation is overridden,
- and the transformed object is type correct for each instance variable,

they will both point to the same object after transformation.

2.4. Default Transformations

With the above invariants in mind, we can now define default transformations for each kind of class change. Here we consider how a class change should affect existing objects in the database. We would like default transformations to affect the contents of the database as little as possible. If a class is not changed, the default transformation for objects of that class should simply copy the objects into the new database. If a class is changed, we want to preserve the maximum amount of information in the object, while changing it to conform to the new class definition.

For instance, let's reconsider the example from Section 2.2 of adding instance variable *v3* to class *C*. We have two cases to consider, depending on whether or not *C*'s *v3* is overriding another definition. First, if *v3* is overriding another definition, then objects of class *C* already have a value for *v3*. If so, we recursively transform the value. If the type of the transformed value is the same or a subtype of *C*'s *v3*, we can assign it to *v3* in the new database. If it is not type correct, we assign the special value *VOID* to *v3* and report an error. Similarly, if *v3* is not an overriding definition, then there is no existing value to assign to *v3*. Instead we initialize it to *VOID*.

Similar default transformations are defined by considering each kind of class change, and determining how to transform existing objects to conform to the new class definition while retaining the maximum amount of information.

Up to this point our model of transformation is not significantly different from that provided by Orion. In some cases, we allow more general transformations. For instance, Orion only allows a variable type to be generalized, while we allow arbitrary changes. In the cases where we differ, our default transformation typically increases the likelihood that information will be lost. By restricting variable type changes to generalizations, Orion assures that the variable's value will still be type correct in the new database, and so information will not be lost.

Why then should we allow arbitrary changes, and risk the loss of information from our database? The answer is that the normal evolution of programs requires not only changes to code, but also arbitrary changes to class definitions. To support general changes to class definitions, it is necessary to support database reorganization. The approach taken by other researchers (e.g., [1, 3, 4]) is that all desired database changes can be addressed by making changes local to individual classes. This assumes that the overall design of the database is correct, but that some information should be either added to or deleted from individual classes. It does not allow for an overall redesign and reorganization of the database. When enhancing a program that uses non-persistent data, a programmer can make arbitrary changes to all datatypes, procedures, and their relationships to each other to provide improved functionality, better performance, etc. We believe the same freedom should be afforded to programmers using databases.

3. Supporting Database Reorganization

The default transformations described above address only local changes to class definitions, not database reorganization. Our unique contribution is that in addition to using OTGen's default transformations, we allow the database administrator to override these default transformations in such a manner that arbitrary reorganization is possible. This section describes how this is done.

To provide general database reorganization in a database transformer, the database administrator must describe the relationships among objects in the old version of the database and those in the new. OTGen provides a tabular notation in which this is done. The table has one entry for each class defined in the old version of the database. It indicates how instances of that class should be transformed. The table is initialized by OTGen. When a class definition is changed, OTGen describes the effect by changing the table to reflect the default transformation provided by the change. The database administrator can then change the table to override the default transformation.

For example, suppose we have a class *C* with two instance variables *v1* and *v2*. (The transformation table lists both variables that are locally defined, and those that are inherited.) Initially, the entry for *C* is:

```
Class C:
    new self:  C
        v1:  Transform v1
        v2:  Transform v2
```

This is the simple copy transformation. Every instance of *C* in the old database is replaced with an instance of *C* in the new database. The values for the variables are the result of recursively transforming the values of the variables from the old database.

Suppose now a new variable *v3* is added to class *C*. The default transformation changes the table to:

```
Class C:
    new self:  C
        v1:  Transform v1
        v2:  Transform v2
        v3:  VOID
```

If the database administrator wishes to initialize the new variable, he can change the entry. For instance, to initialize it to 0, he would change the entry to:

```
Class C:
    new self:  C
        v1:  Transform v1
        v2:  Transform v2
        v3:  0
```

The interesting issues to address for database reorganization are the types of changes that can be expressed using this notation, and how OTGen can ensure that the invariants presented in Section 2.3 are not violated.

3.1. Expressiveness of Transformation Tables

The transformation tables defined by OTGen's default transformations use the following mapping operations to transform an object in an old database to one in a new database:

- Creation of an object in the new database with the same class as the object in the old database.
- Replacement of an object in the old database with *VOID* in the new database. This is used only when the old object's class has been deleted.
- Recursive transformation of an old instance variable's value into a value for the same instance variable in the new database.
- Initialization of an instance variable in the new database to *VOID*.

All of the default transformations can be expressed using the above four operations of the transformation tables.

To support database reorganization, OTGen provides additional operations. They are:

- Initialization of variables
- Context-dependent changes
- Movement of information from one object to another
- Creation of new objects
- Sharing of information among objects

We have already seen how variables can be initialized in transformation tables in the introduction to this section where *v3* was initialized to 0.

3.1.1. Context-Dependent Changes

Context-dependent transformations are supported by allowing boolean expressions to be attached to each portion of a transformation table entry. For instance, suppose we add a new class *D*, which is a subclass of *C* defined as follows:

```
Class D subclass of C is
    v2:  T4
```

Now, suppose we want to transform those instances of *C* whose value for *v2* is in *T4* to be instances of *D*. We could change the transformation table for *C* to be:

```
Class C:
    if TypeCheck (v2, T4)
        new self:  D
            v1:  Transform v1
            v2:  Transform v2
    else
        new self:  C
            v1:  Transform v1
            v2:  Transform v2
```

Here TypeCheck is a function provided by OTGen which is given an object and a class name. It returns true if the type of the object is the same or a subclass of the given class.[3]

3.1.2. Moving Information

Suppose we want to move a variable from one class to another, with the effect that the variable's value will move from an object of the first class to an object of the second class. For instance, suppose we have two classes *Outer* and *Inner*, and change them by moving instance variable *i2* from *Inner* to *Outer*, as shown below:

[3]In reality, we want to type check the transformed value of *v2*, so the first argument to TypeCheck should be Transform(*v2*), which first recursively transforms *v2* before performing the type check. The algorithm described in Section 3.2.3, which ensures the sharing invariant is maintained, would also ensure that *v2* would be transformed only once.

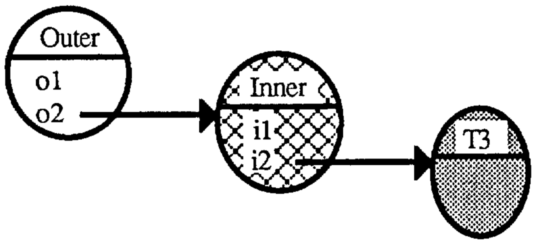
```
Class Outer subclass of S1 is
    o1:  T1
    o2:  Inner

Class Inner subclass of S2 is
    i1:  T2
    i2:  T3
```

Before

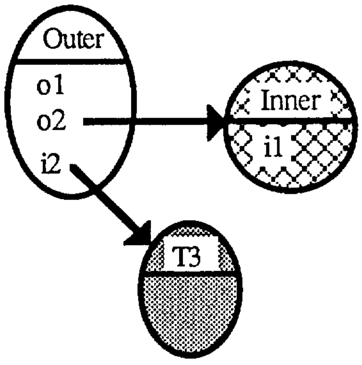
```
Class Outer subclass of S1 is
    o1:  T1
    o2:  Inner
    i2:  T3

Class Inner subclass of S2 is
    i1:  T2
```

After

The effect that we want on the database is demonstrated by Figure 3-1.



Before



After

Figure 3-1: Moving Objects in a Database

The transformation table entries generated by OTGen's default transformation treat this as two changes: the addition of a variable to Outer, and the deletion of a variable from Inner. The resulting table entries are:

```
Class Outer:
    new self:  Outer
        o1:  Transform o1
        o2:  Transform o2
        i2:  VOID

Class Inner
    new self:  Inner
        i1:  Transform i1
```

Using these tables, the value of *Inner*'s *i2* is lost. To retain the value in *Outer*'s *i2* instance variable, the database administrator must override the default transformation by changing the entry for *Outer* to:

```
Class Outer:
    new self:  Outer
        o1:  Transform o1
        o2:  Transform o2
        i2:  Transform o2.i2
```

Any expression that evaluates to an object in the old database may be used in a transform statement. In the above example we dereferenced the instance variables directly. Function calls are also acceptable.

3.1.3. Creating New Objects

Another desirable reorganization is to add objects that did not exist in the original database. For instance, suppose we want to modify objects of class *C* so that *v1* is an object of a new class *Wrapper* whose first field is the old value for *v1*. The definitions of *C* and *Wrapper* are as follows:

```
Class C subclass of S is
    v1:  T1
    v2:  T2
```

Before

```
Class C subclass of S is
    v1:  Wrapper
    v2:  T2

Class Wrapper subclass of T is
    w1:  T1
```

After

OTGen sees this change simply as a type change of an instance variable. The default transformation for a type change is a recursive transformation, where the assignment will fail if the transformed variable value is not type correct for the variable. To get the effect that the database administrator wants, he must change the transformation table entry to:

```
Class C:
    new self:  C
        v1:  Create WRAPPER
            w1:  Transform v1
        v2:  Transform v2
```

Figure 3-2 shows the effect this transformation has on an object of class *C*.

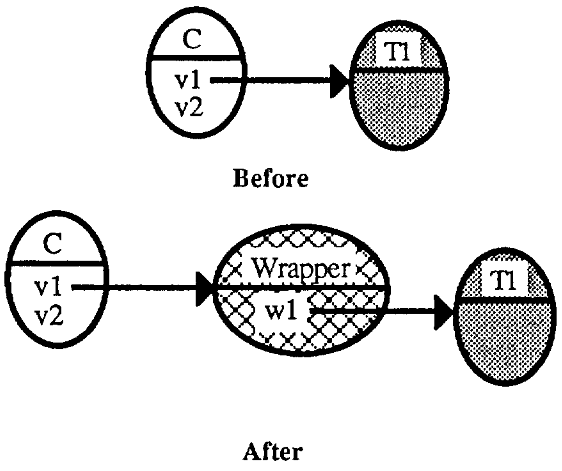


Figure 3-2: Creating Objects During Transformation

3.1.4. Sharing Information

The sharing invariant stipulates that sharing must be maintained across transformation. However, suppose the desired database reorganization is to introduce sharing. This can be done in one of two ways. If the object to be shared existed in the old database, simple transform statements, such as those used above, identifying the object to be shared will introduce the sharing. However, if the object to be shared is being created in the new database, sharing must be done explicitly using a *shared expression*.

For instance, suppose we have two classes *C1* and *C2*. Some instances of each of these classes share values via variables *v1* and *v2*, respectively. Suppose we want to wrap this shared value in a new class, called *Wrapper*, and share this value instead. An example of this transformation is shown in Figure 3-3.

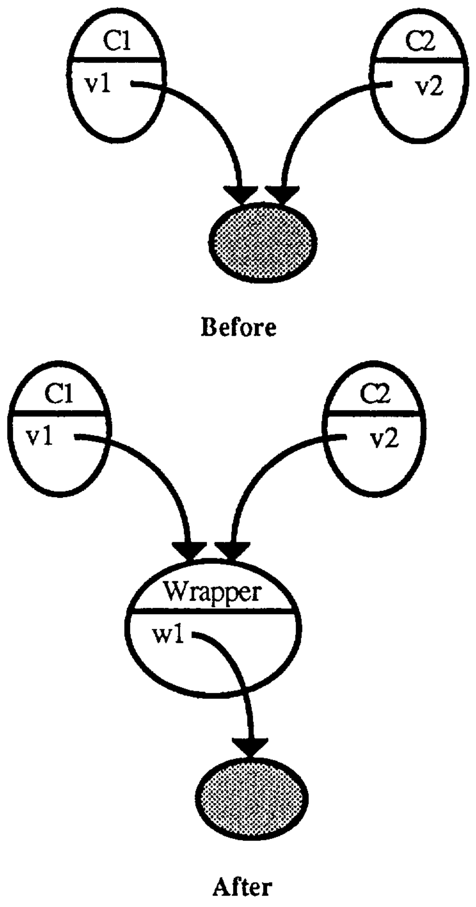


Figure 3-3: Introducing Sharing During Transformation

This sharing is achieved as follows:

```
Class C1
   new self:  C1
      v1:  Share NewWrap(v1)

Class C2
   new self:  C2
      v2:  Share NewWrap(v2)

Shared expression NewWrap(oldobj)
   Create WRAPPER
      w1:  Transform oldobj
```

A shared expression is a parameterized variable transformation. A shared expression is evaluated once for each distinct set of argument values it is instantiated with. After it is evaluated, the result is cached in a table indexed by the argument values. Then, if the shared expression is instantiated a second time with the same set of argument values, the cached value is returned. In this way, the sharing can be moved from one object to another as in the example above.

Any object from the old database can be passed to a shared expression. The shared expression may require any number of arguments, including zero. If zero arguments are used, then all references to the shared expression will result in sharing of a single object. Also, it is not necessary for a shared expression to use the arguments in evaluating the expression. They might just be used to control sharing by being used as indices into the cached table.

3.2. Guaranteeing Transformation Invariants

In this section, we describe how OTGen preserves the transformation invariants in the presence of arbitrary database reorganization.

3.2.1. Completeness

In OTGen we collect together class definition changes before updating database objects, rather than creating a new database version for each change. A database administrator performs a collection of related changes, affecting both class definitions and methods, and releases them as a unit. When changes are released, it is necessary to restart all database servers, but it is not necessary to transform the database immediately. Instead, we support lazy transformations, where a collection of connected objects is transformed when it is first accessed with the new version of the database server. In this way the downtime of the database is very short, with short delays the first time each object from an old version is accessed.

Each collection of connected objects has a root through which it can be externally accessed. Each root has a version number indicating the version of the database server that last accessed it. If the current database server is a newer version, all objects reachable from the root are transformed. If the objects are more than one version out-of-date, a series of transformers will be called, each updating the objects one version. By transforming all connected objects when the root is accessed, we can assure the completeness invariant.

3.2.2. Correctness

The correctness invariant of transformation is guaranteed by the transformation algorithm itself. All operations provided by transformation can be broken down into two basic components: creation of new objects, and assignment of values to instance variables. When an object is created, it is created using the latest version. Thus, new objects are type correct. Before a value is assigned to a variable (for instance during initialization, or recursive transformation of a variable), the value is tested for type correctness. If the value is not type correct, *VOID* is assigned, and an error message is produced.

OTGen provides some assistance to the database administrator during modification of the transformation tables, in order to increase the likelihood that the resulting transformations will be type correct. Whenever the database administrator requests the construction of a new object, OTGen provides a template listing the variables of the new object's class. The table is not complete until each variable transformation is specified. It is not possible for the database administrator to delete variable entries within a construction. In addition, when a variable transformation specifies that a new object should be constructed, OTGen can determine whether that construction will be type correct by comparing the new object's class to the variable's type. If the object's class is not the same or a subclass of the variable's type, the construction can be flagged as an error immediately, rather than waiting until transformation time.

3.2.3. Sharing

The sharing invariant is guaranteed by the transformation algorithm by associating unique identifiers (UIDs) with each object in the old database. When an object is recursively transformed, it maintains the same UID in the new version of the database. Each object created during transformation gets a new UID. Before recursively transforming a variable's value, the transformer looks at the UID of the value in the old database, and sees if an object exists in the new database with that UID. If so, the old object has already been transformed, and the corresponding new object can simply be assigned to the variable (assuming it is type correct).

Sharing introduced by shared expressions works in a similar manner. The tables caching the results of shared expressions are indexed by UIDs of the arguments, and return a UID for an object in the new database if a match is found. For instance, reconsider the example from Section 3.1.4. Suppose the shaded object has UID 3. When the *C1* object is transformed, it requests a shared value described by *NewWrap(3)*. Suppose *NewWrap*(3) has not yet been evaluated. The lookup in the cached table will fail. We therefore create a new object of type *Wrapper*. Suppose its UID is 4. We then make the following entry in the *NewWrap* table:

oldobj	`Result`
3	4

Now when the *C2* object is transformed, it requests the shared value described by *NewWrap(3)*. This time the lookup in the cached table succeeds, and the new shared object is assigned to variable *v2*.

4. Conclusions

The ability to change the format and reorganize the contents of a database are imperative if the database is to keep pace with the demands of its user community. In this paper, we have described the design of OTGen, a tool to aid the database administrator in the development of transformers to facilitate such updating of databases. The functionality achieved in this way goes beyond that provided by Orion [1], Gemstone [3], and Skarra and Zdonik's work [4]. While they allow simple changes to be performed to individual schema, we support not only more complex operations, but also support database reorganization, and arbitrarily complex transformations on the contents of individual objects as well as the database as a whole. We believe the evolution supported by OTGen is very important, since it allows databases to evolve as users' experiences create new demands on the database, rather than remain more or less committed to the original database design conceived probably with good intentions, but undoubtedly with lack of experience.

Acknowledgements

We were assisted in the design and implementation of TransformGen and OTGen by numerous members of the Gandalf project, most notably David Garlan, Charlie Krueger, Robert Stockton, and Benjamin Pierce. We also would like to thank those who commented on earlier versions of this paper: Charlie Krueger, Robert Stockton, and Richard Lerner.

REFERENCES

[1] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth.
Semantics and Implementation of Schema Evolution in Object-Oriented Databases.
In Umeshwar Dayal and Irv Traiger (editors), *Proceedings of the ACM SIGMOD 1987 Annual Conference*, pages 311-322. San Francisco, May, 1987.

[2] David Garlan, Charles W. Krueger, Barbara J. Staudt.
A Structural Approach to the Maintenance of Structure-Oriented Environments.
In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 160-170. Palo Alto, December, 1986.
Reprinted in SIGPLAN Notices, January 1987.

[3] D. Jason Penney and Jacob Stein.
Class Modification in the GemStone Object-Oriented DBMS.
In *OOPSLA '87 Proceedings*, pages 111-117. Orlando, Florida, October, 1987.

[4] Andrea H. Skarra and Stanley B. Zdonik.
The Management of Changing Types in an Object-Oriented Database.
In *OOPSLA '86 Proceedings*, pages 483-495. September, 1986.

[5] Barbara Staudt, Charles Krueger, and David Garlan.
TransformGen: Automating the Maintenance of Structure-Oriented Environments.
Technical Report CMU-CS-88-186, Department of Computer Science, Carnegie Mellon University, November, 1988.