Lazy Schema Evolution in Object-Oriented Databases

by

Yin Cheung

B.A., Computer Science and Economics
Wellesley College (1999)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

Author .....
Department of Electrical Engineering and Computer Science
August 31, 2001

Certified by .....
Barbara Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Committee on Graduate Students

Lazy Schema Evolution in Object-Oriented Databases

by

Yin Cheung

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2001, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Object-oriented database (OODB) systems provide persistent storage for large numbers of long-lived objects shared by many programs. At any point in time, the database obeys some *schema*. Over time, the system is likely to experience schema changes. The requirement for both good performance and sound semantics has made schema evolution a long-standing challenge. When and how to evolve existing objects to conform to a new schema is at the core of the problem. This thesis addresses this problem with both semantics analysis and an implementation.

First, we provide a simple interface for users to specify upgrades as a collection of class upgrades. Each class upgrade contains a transform function that converts an object of the old class into an object of the new class.

Next, the question of when to apply these transform functions is addressed. In order to preserve the availability of the system, we proposed a lazy upgrade model where the transformation of objects is delayed until they are accessed by applications. To guarantee the consistency of the database, our model requires that the upgrades be complete and the transform functions well-defined. The interleaving of applications and transforms poses some problems. Specifically, an object might belong to either an earlier or later version when it is encountered by a transform function. To cope with the former case, our model provides a mechanism for running nested transforms; to deal with the latter case, our model keeps snapshots of objects when they are transformed.

Finally, we implemented our approach in Thor, an object-oriented database system. The implementation was done with efficiency and performance in mind.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

Acknowledgments

I would like to thank my advisor, Barbara Liskov, whose insights and ideas have helped me develop this thesis. Her critical and careful proof-reading significantly improved the presentation of this thesis.

Chandra Boyapati and Liuba Shrira helped me clarfy many design and implementation issues through discussions. Chandra has also been great officemate whose wisdom and kindness always benefit those around him.

Thanks to Sidney Chang, Jinyang Li, Athicha Muthitacharoen, Bodhi Priyantha, Rodrigo Rodrigues, Ziqiang Tang, Xiaowei Yang, Karen Wang and many other people for making the Lab a pleasant place to work. And thanks to Laura and Q for great fun outside the Lab.

Finally, I would like to thank Kyle for always pushing me to finish my thesis. And thanks to my parents and my sister who provide the love and support I can always count on.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Object-oriented database (OODB) systems provide persistent storage for large numbers of long-lived objects shared by many programs. At any point in time, the database obeys some *schema*. A schema usually defines a relation on classes that make up the inheritance graph. This class graph defines not only how data is represented in the database but also an interface through which users and programmers of the database query and modify data. Each object in the database assumes a type in the schema and the database is type-correct as a whole.

Over time, the system is likely to experience changes that affect the interface to the persistent data. These changes are necessary when the old design and implementation become obsolete or inappropriate and modifications are needed to meet the new demands. These changes often involve updates of the database schema and support of *schema evolution* or *upgrades* is crucial for any realistic OODB systems.

Adding/removing classes/attributes/methods and changing the type/class hierarchy are among the common changes to the schema. Since an object-oriented database is populated by objects instantiating classes, a change in the schema necessarily yields some objects incompatible with post-evolution programs. Evolving existing objects to be compatible with the new schema is thus the core of schema evolution.

1.2 Challenges

Much work ([4, 25, 10, 24, 17, 18, 29, 27, 7, 23, 26]) has been done on schema evolution in the past. However, schema evolution remains a hard problem for researchers. The main challenges are the following.

1. A good schema evolution system should provide a simple programming interface, yet be flexible enough to allow arbitrary schema changes.

2. The system is long-lived, so errors introduced into the system are also long-lived. Hence, it is very important to preserve the consistency of the database.

3. The system contains a large number of objects and has many users. Running upgrades on many of these objects may take a long time. Hence, it is important to preserve the availability of the system.

4. A system that supports schema evolution should not sacrifice performance. Hence, the implementation should be efficient. In particular, upgrades are likely to be rare, so that the implementation should be optimized for the common case where there is no active upgrade.

1.3 Thesis Contribution

This thesis makes the following contributions:

1. We provide a simple yet flexible interface for programmers to define a schema upgrade as a collection of *class upgrades*.

2. Based on class upgrades, we developed an efficient lazy upgrade propagation model where transformations of objects are delayed until they are accessed by applications. To guarantee the consistency of the database while applications and transforms are interleaved is non-trivial. This thesis provides a thorough investigation of this issue.

3. We implemented schema evolution in an object-oriented database system. For running normal applications, the overhead of the system is small.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 introduces the semantics of upgrades and our design for a system implementing lazy upgrades. Chapter 3 covers additional details, including the syntax and type checking of transform functions, simulation methods and the completeness of upgrades.

Our implementation is done in the context of Thor [3], an object-oriented database. Chapter 4 gives an overview of Thor. It is followed by Chapter 5, which describes our implementation in detail. Our work is compared with previous work in Chapter 6. Finally, Chapter 7 draws conclusion and discusses paths for future work.

Chapter 2

Semantics and Design

This chapter discusses semantics for running upgrades.

2.1 System Model

We assume an object-oriented database system. The system uses an object-oriented programming language, such as Java. This language is based on classes and is statically-typed.

The types defined in the language form a directed acyclic graph (DAG). Classes implement types and objects instantiate classes. Types and classes are identified by their names. We assume that there is a root type, *Object*, which is the supertype of all types. Objects are identified by their object ids. Objects store these object ids to refer to one another, forming an object graph. Objects can also contain primitive values such as integers, booleans, etc.

All fields of an object are *fully encapsulated*, which means they can only be accessed indirectly through method calls. From an object, we can also get to the class it belongs to and hence to its interface.

There is a special object that is designated as the persistent root of the database. All objects reachable from it are defined to be persistent. Objects that are not reachable from the persistent root or from any application are garbage collected.

To keep the database in a consistent state, applications access and modify the

objects within atomic transactions.

2.2 Upgrades

At any point in time, the database obeys some *schema*. A schema defines a relation on classes that make up the inheritance graph. This class graph defines not only how data is represented in the database but also an interface through which users and programmers of the database query and modify data. Each object in the database assumes a type in the schema and the database is type-correct as a whole.

Schemas are not expected to change often. However, sometimes we need to *upgrade* the schema, in order to

1. correct an implementation error;
2. make an implementation more efficient;
3. extend an interface; or,
4. modify an interface.

In cases 1, 2 and 3, the upgrade is *compatible* with respect to the previous schema because after the upgrade the objects still support the same interface as before the upgrade. In case 4, the upgrade is *incompatible* because the type changes to a different type.

An object retains its identity when it is upgraded. This means all other objects that refer to it continue to do so after the upgrade. However, when the upgrade is incompatible, objects that refer to the upgraded object either directly or indirectly may also need to be upgraded to guarantee the consistency of the database. In particular, every object whose implementation depends on properties of the old type of an upgraded object that are not supported by the new type will need to be upgraded to use the new type instead. All of these objects are said to be *affected* by the upgrade.

In general, an upgrade changes many objects, and these objects can belong to many different classes and types. An upgrade is said to be *complete* if it includes all

the objects affected by the upgrade. We assume that all upgrades are complete in this thesis. Checking of completeness is discussed in Chapter 3.

2.3 Upgrade as a Single Transaction

The simplest way to upgrade a database is to run the transformation as a single transaction. The transaction runs on a consistent database that conforms to the old schema and transforms it into a consistent database that conforms to the new schema.

This approach is error prone. The consistency of the database relies solely on the meticulousness of the upgrade definer. Excluding even one object from the transform transaction by mistake would result in a broken database. Therefore we want to shift the task of upgrading to the system, which can do it automatically.

In addition, running the upgrade as one transaction requires "stopping the world", that is, no application transaction is allowed to run during the time when the upgrade transaction is running. Because there might be many objects involved in the upgrade, database access can be interrupted for a long period of time. This is unacceptable for most systems.

For the above reasons, we do not take this approach. The following section describes an alternative approach.

2.4 Upgrade as a Collection of Transforms

In our system, an upgrade is defined as a set of *class upgrade*s. Each class upgrade is a tuple:

$$\langle C_{old}, C_{new}, TF \rangle$$

where C_{old} is the class to be modified by the upgrade, C_{new} is the corresponding class after the upgrade, and TF is a *transform function* of the following type:

$$C_{old} \rightarrow C_{new}$$

That is, a transform function takes as an argument an object of class C_{old} and produces an object of class C_{new}. The system then causes the new object to take on the identity of the old object. Exactly when this happens depends on how we choose to run the upgrade. Section 2.5 and Section 2.6 discuss this issue. The syntax for writing transform functions is described in Section 3.1.1.

Earlier systems that support upgrades, such as ORION [4] and Gemstone [25], do not have transform functions. Rather, the schema can only be changed by composing some fixed primitives. Later systems, such as ObjectStore [24], OTGen [17] and O_2 [30, 10], allow user-defined transform functions to enable more flexible schema changes. Some of these systems only allow access to data members of the object to be transformed in a transform function; our system removes this restriction by supporting method invocation on objects other than the object being transformed by the transform function. This gives our transform functions more expressive power.

We do impose two restrictions on the transform functions to ensure the consistency of the database:

1. Transform functions must be *well-defined*. A transform function is well-defined if it does not modify any existing objects other than the object being transformed. The justification for this restriction is given in the next section.

2. Transform functions must terminate, which implies two things. First, transform functions must return normally, with the new objects initialized. Secondly, transform functions must not create objects that belong to classes affected by the upgrade. This is to ensure that the upgrade will not go on forever. Type checking can be used to check these constraints to some extent (see Section 3.1.3).

We call an application of a transform function a *transform*. Thus, we can think of an upgrade as a collection of transforms.

Defining upgrades in the above fashion does have some fundamental limitations:

1. We have no way to transform more than one object in one transform.

2. Because the transforms are run independently, we can not impose any order on when objects get transformed.

The impact of these limitations is discussed in Section 2.6.5.

2.4.1 Upgrade Execution

At a high level, the execution of an upgrade happens in two stages. First, the upgrade is *installed*. As part of the upgrade installation, the system serializes the upgrade with respect to all previous upgrades and assigns it a unique *version number*. New class definitions and transform functions are checked to ensure type correctness. The system also checks transform functions for well-definedness, completeness, and termination. Finally, all classes affected by the upgrade are marked as no longer current and the system will not allow objects of these classes to be created.

Next, the upgrade is *propagated* to the whole database. All the objects belonging to the affected classes in the upgrade are transformed to their new forms as determined by the upgrade. The transformation is accomplished by applying the appropriate transform functions.

Conceptually when an upgrade is installed, it is propagated to the entire database immediately. Furthermore, the transforms run as independent transactions. Therefore, by the time the propagation of the nth upgrade occurs, all earlier upgrades have already been propagated and no objects belonging to old classes from earlier upgrades exist. Once this upgrade has been completely propagated, there will no longer be any objects in the database belonging to the old classes of this upgrade. At this point, old classes and transform code can be garbage collected.

Although the upgrades are propagated instantly after installation in the conceptual model, the real implementation does not have to work like this. In the following sections, we explore and compare two approaches: immediate and lazy upgrade propagation. Note that in either approach, objects are transformed in an arbitrary order. Thus, if transform functions were allowed to modify existing objects in the database, they might observe effects of other transforms. To guarantee the deterministic be-

havior of the system, we restrict transform functions to be well-defined.

2.5 Immediate Upgrade Propagation

Many systems ([4, 25, 24]) implement immediate upgrade propagation. This approach is consistent with the conceptual model, where *all* objects affected by the upgrade are transformed immediately after the upgrade is installed.

2.5.1 Basic Mechanism

First, we abort all running applications. No application is allowed to run when the upgrade is being propagated. In other words, we "stop the world" to run an upgrade.

Next, an empty space, S, is allocated to temporarily store all newly transformed objects. In the worst case, S needs to be large enough to hold the entire database.

Next, we find all objects that are affected by the upgrade. Since our system maintains a persistent root, we can find such objects by traversing the object graph from the root and checking each object to see whether it is affected. For systems that maintain a list of instances associated with each class, finding affected objects involves enumerating the instance lists of the classes affected by the upgrade.

If an object is affected by the upgrade, a transform is triggered. This invokes the appropriate transform function on the object. The new objects produced by the transform is placed into the storage, S. When the traversal is completed and all affected objects have been transformed, the old objects in the database are replaced with their new counterparts in S. Thus, the new objects take on the identities of the old objects.

At this point, the upgrade propagation is complete and applications based on the new schema can start running.

2.5.2 Analysis

Pros

The advantage of the immediate approach is its straightforward implementation and clean semantics. There is a clear boundary between when and where transform functions and applications run. An upgrade runs on the *snapshot* of the database conforming to the old schema and applications are oblivious to the upgrade since they only see the database conforming to the most up-to-date schema.

Cons

The size of the temporary space required by this approach is proportional to the number of objects affected by the upgrade in the whole database. Since upgrades is likely to affect a large number of objects, this storage can be huge.

In addition, finding and transforming all objects affected by the upgrade all at once will take a long time for a large, distributed database. This disrupted availability is unacceptable for most systems and this is why we decided to take a lazy approach instead.

2.6 Lazy Upgrade Propagation

In a lazy upgrade propagation approach, transformation of an object affected by the upgrade is delayed until the object is first accessed by an application transaction. In other words, transform functions are run "just in time" to transform objects needed by the application at the moment. Systems such as O_2 [10] and Shan Ming's system [29] take this approach.

The time overhead of the upgrade is incremental in the lazy approach – it is spread across applications. The system is not interrupted for a long period of time to perform the upgrade, only as long as it takes to transform objects immediately needed by an application. For applications that do not use objects affected by the upgrade, there is no cost. However, interleaving applications and transforms can

cause type inconsistency and invariant violations if the implementation is not careful. Section 2.6.1 discusses the interleaving of applications and transforms in our system. Section 2.6.3 and Section 2.6.4 explore two ways to run lazy upgrades and Section 2.6.5 discusses the interaction between upgrades and invariants.

2.6.1 Interleaving Applications and Transforms

Unlike in the immediate approach, object transformation is done on an as-needed basis. Like the immediate approach, applications only see the most up-to-date objects and are oblivious to the upgrades.

Each object that an application accesses must be checked to see if it belongs to the current schema. If it does, the application uses it as it is. If it does not, in a naive implementation, the application transaction is aborted and a transform transaction is started to run the appropriate transform function on this object. When the transform transaction completes, the application is rerun.

Aborting a transaction to run a transform is not efficient, especially if the transaction has done a large amount of work already. Furthermore, the path of execution may not be the same when an application is rerun because some objects may have been changed by some other applications. Therefore, instead of aborting, we suspend the triggering transaction to run a transform. The transaction continues running after the transform completes if it does not *conflict* with the transform. If there is a conflict, the triggering transaction is aborted. A triggering transaction conflicts with the transform it triggered if it accessed objects that were then modified by the transform. Note that it is impossible for a well-defined transform function to conflict with other applications or transforms since it does not modify any objects other than the one being transformed.

A *stack* can be used to save the necessary states of the suspended transactions so that they can be restored and run in the reverse order they started. This mechanism is similar to the *context-switching* in operating systems.

Because of the laziness, one schema may be installed before an earlier one is propagated to the entire database. As a result, the database may contain objects from

multiple schemas at any point in time. Therefore, a running transform transaction may encounter objects from different versions, even though applications can only see the most up-to-date objects. There are two cases that we need to handle:

1. The transform transaction encounters an object, `o`, of an earlier version. At this point, we run transforms on `o` as many times as necessary in order to bring it up to the version that existed at the time the transform function's upgrade was installed.

2. The transform transaction encounters an object, `o`, that has already been transformed to a version beyond that of the current transform function. The system needs to provide the old interface and its state for the transform function.

Case 2 highlights the need for the system to keep versions of objects, as is done in both designs to be described in Section 2.6.3 and Section 2.6.4.

2.6.2 Time Analysis of Lazy Upgrade Propagation

For the immediate approach, the time overhead of an upgrade is paid once and for all at the beginning of the upgrade. This is the time it takes for the entire database to be transformed and it may be long. But after the upgrade is complete, applications can run normally without any extra overhead.

By contrast, the lazy approach spreads the overhead of transforms over applications incrementally. Applications do not have to wait until the upgrade is fully propagated to start running but they "pay as they go" on individual transforms. Furthermore, there is a fixed cost of checking object versions imposed on transactions that does not exist in the immediate approach. An efficient implementation (see Chapter 5) can keep this cost low.

2.6.3 Upgrade Snapshot Approach

This approach is a straightforward emulation of the immediate approach described in Section 2.5. Like the immediate approach, we save the snapshot of the database as it

was at the beginning of the upgrade (hence "upgrade snapshot") and run transforms on this snapshot. The difference is, we save the snapshot incrementally and run the transforms lazily.

Design

The upgrade snapshot is only read by the transform functions. Two kinds of objects might be used by the transform functions in a particular upgrade and hence need to be saved:

1. Objects affected by the upgrade. Since they might be used by later transforms within this upgrade, their pre-transformed states (snapshots) need to be saved the first time they are transformed.

2. Objects that are not affected by the upgrade but might be read by some transform function of the upgrade. We call these objects *tf-read* objects.

   Unlike in the immediate scheme, tf-read objects can be modified by an application before they are read by a transform function. To prevent exposing the modification to transform functions, we need to save the pre-modified states of tf-read objects.

To save the snapshot, we allocate some space, P, which is initially empty. Affected objects are copied into P when they are first accessed by an application; and tf-read objects are copied into P when they are first modified by an application since the upgrade was installed.

Applications and transforms are interleaved as described in Section 2.6.1. When a transform function runs, it uses P to locate the right object to use. P is carried across upgrades and therefore may contain many versions of objects.

Objects that are no longer referenced are removed from P. However, old classes cannot be garbage collected until all objects of the class are transformed, which might take a long time.

Analysis

The cost of the upgrade snapshot approach is related to the time and space required to save the states of the tf-read objects. Ideally we would like to save only the states of objects that are modified and then later read by a transform, but this is clearly not a realizable goal.

One approach that might work would be to recognize (by static analysis of the transform functions) *tf-read classes*. Tf-read classes are classes whose objects *might* become tf-read objects. Then only these objects need to have their snapshots saved when they are modified.

However, identifying tf-read objects this way is far from precise. As a result, the space overhead is very likely to be more than what it could be. In a language like Java, where there is no parametric polymorphism, the `Object` class is used at a lot of places. For example, an object being transformed may refer (directly or indirectly) to objects as belonging to type `Object`. If the transform function uses `Object` methods on these objects, then we need to save the snapshot of these modified objects. In the worst case, we may end up saving the entire database.

Furthermore, unlike in the immediate approach, the temporary storage for the snapshot is carried across upgrade. Even though garbage collection can help to reduce the size, there is no obvious upper bound on this storage. In the worst case, the space overhead could be many times bigger than the database itself, containing many versions of the database.

In summary, the space overhead of upgrade snapshot approach is very hard to quantify and may be much larger than that of the immediate approach.

2.6.4 Transaction Snapshot Approach

As mentioned in Section 2.6.3, the space overhead of the upgrade snapshot approach can be huge. In particular, saving all objects of classes that might be read by some transform function seems to be an overkill. To solve this problem, we developed a different approach. In this approach, we use the snapshot of tf-read objects at the be-

ginning of an application transaction (hence "transaction snapshot") to approximate their upgrade snapshot. The space overhead is greatly reduced as a result.

Design

As in the upgrade snapshot approach, some space P is allocated to store the snapshot of affected objects. However, P no longer contains the snapshots of the tf-read objects. Instead, we use the *undo log* to record tf-read objects.

Undo logs are used in some transactional systems, e.g., Thor [3], to revert the state of the system when a transaction aborts. Each application transaction keeps an undo log, which is initially empty. The undo log records old copies of objects when they are first modified by the application transaction. This log is cleared when the application commits.

If a transform function needs to access a tf-read object, it looks for it first in the undo log of the triggering application transaction. If the object is not found there, the transform function simply uses the current object in the database. Therefore, the transform function might observe a modification of the tf-read object made by an earlier application transaction. Essentially, transform functions run on the snapshot of tf-read object at the beginning of the triggering application transaction, rather than the snapshot at the beginning of the upgrade. Both snapshots are equivalent only if no earlier application transaction modified the tf-read object since the upgrade installation. This point is discussed in Section 2.6.5.

Analysis

Compared with the upgrade snapshot approach, the transaction snapshot approach does not have the space overhead of the tf-read objects. This could be a big saving in space if most of the transform functions are complex. Furthermore, this approach completely avoids the potential space blowup caused by transform functions that use `Object` methods, as mentioned in Section 2.6.3.

2.6.5 Discussion

Because of its space advantage, we chose the transaction snapshot approach for our implementation over the upgrade snapshot approach. However, as mentioned earlier, the snapshot observed by a transform in this approach may not be identical to what would have been observed in either the immediate or the upgrade snapshot approach.

The difference arises only when some classes have *unencapsulated* representations. The *representation* of a class (or *rep* for short) is the set of objects accessible from the instance variables([19]). The rep is *encapsulated* if it is not possible to access any of these objects *except* by using methods of the containing object; otherwise the rep is unencapsulated and the offending object or objects are *exposed*.

A fully encapsulated rep is desirable because the correctness of a class can be determined through local reasoning. In particular, part of this reasoning is based on preserving the *rep invariant*. This is a property that must hold for every object in the class. When reasoning about each method, we can assume that the invariant holds when the method starts to run, and we must prove that the invariant holds when the method terminates.

An example of an unencasulated object is given in Figure 2-1. Here unencapsulated object `x` refers to object `z`, but code outside of `x` also refers to `z`. Thus `z` is exposed.

An unencapsulated rep is not a problem *provided* the implementation of the containing object depends only on invariant properties of the exposed objects. We call such dependencies *safe* dependencies. Examples of safe dependencies are:

- The containing object relies on the identity of the exposed object. This situation is not a problem because object identity does not change.
- The exposed objects are immutable. This situation is not a problem because it will not be possible for code outside the object to interfere with any assumptions made within the code about the exposed object.

Safe dependencies do not interfere with local reasoning about the rep invariant because code outside the class is unable to cause the invariant to not hold.

However, if the code of the containing object depends on non-invariant properties of the exposed objects, there might be a problem, because it is no longer possible to reason locally about the correctness of the code. In this case, we say that the rep is exposed ([19]).

For example, in Figure 2-1, suppose `z` is mutable and that `x` depends on a property of `z` that would be violated if `z` were modified. Then `x` would have an exposed rep.

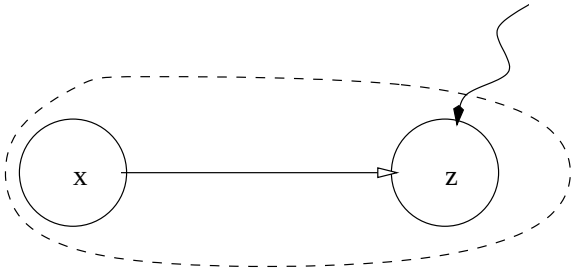

Figure 2-1: An example of a rep exposure

When there are rep exposures, there are two situations where the rep invariants are still preserved. Using our example, the rep invariant of `x` is preserved in the following two cases.

1. There is some encapsulating object `y` that points to both `x` and `z`. All modifications to `z` go through some methods of `y`, which ensure that the rep invariant of `x` holds with respect to `z`. This is shown in Figure 2-2.

2. All code outside `x` that accesses `z` preserves the rep invariants of `x`. In other words, any code that modifies `z` must also preserve the rep invariant of `x`, maybe by modifying it as well. In fact, this can be looked upon as the same as case 1, except that the encapsulating object is the whole database. This is shown in Figure 2-3.

Both cases above preserve rep invariants despite the rep exposure. However, case 2 is undesirable compared to case 1. From a software engineering point of view, requiring arbitrary code to preserve some rep invariant makes the system very fragile

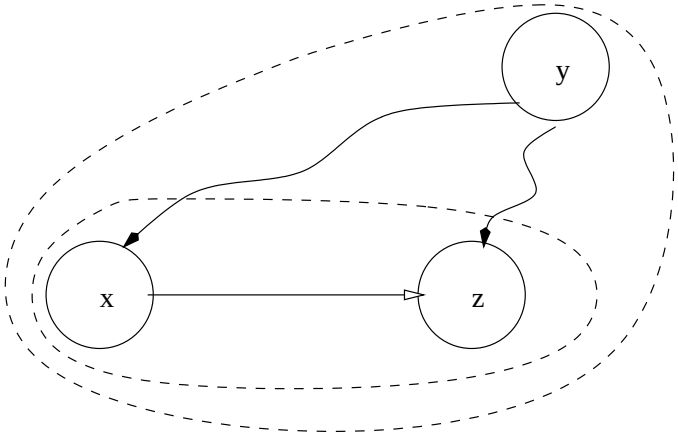

Figure 2-2: Encapsulating object ensures rep invariants

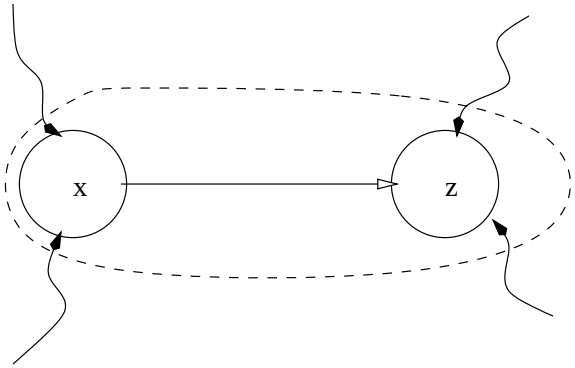

Figure 2-3: Access code ensures rep invariants

and error-prone. A system with encapsulation makes the system more robust and is therefore preferred.

Now let us see what happens when there are upgrades. When an old schema is evolved to a new schema, new invariants associated with the new schema may need to be established while the old ones are abandoned. The transform functions will be required to establish the new invariants. However, we also expect to be able to assume the old invariants when the transform functions start to run. The issue is whether an upgrade would violate this assumption.

In the immediate upgrade propagation approach, transform functions run entirely

on the snapshot of the old database and new transformed objects are put back into the database only at the end of the upgrade. Furthermore, applications run after the upgrade propagation completes. Therefore, transform functions do not observe any modifications from the application that might break some invariants of the old schema; and the applications are oblivious to the upgrade. For the upgrade snapshot approach, there is also no problem because it is a straightforward emulation of the immediate approach.

For the transaction snapshot approach, we know that applications will not observe broken invariants. This is because the propagation of the upgrade guarantees that applications are oblivious to the upgrade: they only observe objects of the new schema. Thus, assuming that applications are rep preserving, the new rep invariants are preserved.

However, transform functions in the transaction snapshot approach might observe modifications from applications, depending on which objects are accessed and when. Since the objects are transformed in arbitrary order, the transform functions might observe broken invariants. Consider the example we gave at the beginning of this section (Figure 2-1 and 2-2):

1. Both `x` and `z` are affected by the upgrade. Suppose `z` is transformed to `z'` before `x` is transformed. If the transform function of `x` uses `z`, we can still use the old version of `z` even if `z'` is modified by some application. This is because we save the old versions of transformed objects in this approach, as in the upgrade snapshot approach. On the other hand, since the transform function of `z` can not possibly rely on `x`, we do not have a problem if `x` is transformed before `z`. In other words, transform functions do not observe broken invariants if all objects involved are affected by the upgrade. Note that this is true even if we do not have the encapsulating object `y` as in Figure 2-3.

2. `x` is affected by the upgrade but `z` is not. Furthermore, the transform function of `x` uses `z`, i.e., `z` is a tf-read object. Now, suppose before `x` is transformed, `z` is modified by some application. Since `z` is a tf-read object, its snapshot is

not saved. Therefore, the transform of `x` would use the modified `z`. Since the modification of `z` does not necessarily preserve the old invariants (because it is caused by an application in the new schema), the rep invariant of `x` might be broken and this could be observed by the transform function.

As a concrete example, suppose `z` is a mutable object but the implementation of `x` depends on its value being constant. This assumption is ensured by the implementation of `y` in the old schema, assuming the structure in Figure 2-2. Now suppose the upgrade causes `y` to mutate `z` and the implementation of `x` is changed to accept this. Note that `y` and `x` are both affected by the upgrade while `z` is not.

Now suppose an application transaction calls a method `m` on `y`, which causes `y` to be transformed; furthermore, suppose `m` actually modifies `z`. Since the new rep invariant of `x` does not depend on `z`, `x` is not accessed in this application transaction. Later, when `x` is accessed and transformed, it accesses `z` with a modified state, which might break the invariant that the transform function assumes.

In case 2 above, the safe dependency between `x` and `z` is made unsafe by the upgrade. Upgrade definers should be careful to avoid such incompatible upgrades, e.g., turning immutable objects into mutable objects, especially in the case where there is not any encapsulating object outside the exposed rep. However, if the programmer must define such an upgrade, there are a couple of ways to get around the problem:

1. Let transform functions deal with it, i.e., the transform functions are aware of potential rep exposure caused by the upgrade. In our example, the transform function for `x` can be written to take into account the fact that `z` might be modified by the time the transform function is run on `x`. Note that this approach also works for the situation in Figure 2-3, where there is no encapsulating object.

   This solution places more of a burden on transform function writers. They no longer have the simple snapshot illusion as provided by the immediate approach and the upgrade snapshot approach; and they can no longer rely on the old rep

invariants if they are changed in the new schema. However, this solution is practical: after all, writers of transform functions need to understand both the old and new schemas and they need to think hard about issues related to invariants.

Defining "upgrade-aware" transform functions only solves part of the problem, however. Section 3.2 explains why it is not sufficient and how we can solve it using *simulation methods*.

2. We can extend the definition of upgrades to give programmers some control over the order in which transforms are applied. Such an extension would be inexpensive to implement in the case illustrated in Figure 2-2, in which the exposed object is encapsulated within a containing object. For example, we can allow the transform function for `y` to specify that `x` should be transformed immediately after the transform for `y` completes. This ensures that `x` is transformed before an application modifies `z`, i.e., before an application can interfere with the old rep invariant.

   In the case where the exposed object is not encapsulated inside any object, as in Figure 2-3, a general query mechanism might be required to determine the order to apply transforms. This could be both complicated to define and expensive to implement. Exploring what would be required for such a query mechanism is an interesting avenue for future work.

Chapter 3

Defining Upgrades

This chapter discusses a number of issues that arise in defining upgrades.

3.1 Transform functions

3.1.1 Syntax of Transform Functions

For every class upgrade, $\langle C_{old}, C_{new}, TF \rangle$ in an upgrade, there is exactly one transform function:

$$TF : C_{old} \rightarrow C_{new}$$

In this section, we explain the basic syntax of writing transform functions. In our system, transform functions are written using an extension of Java. A transform function is defined like a constructor for the new class:

```
NewClass (OldClass x) {
   // initialize new object of NewClass using x
   ...
}
```

If the names of the old and new classes are different, the transform function uses these names to refer to them. But sometimes one might want to use the same name

for both old and new classes in a compatible class upgrade. In this case, the transform code can use `className_old` and `className_new` to refer the old and new version of the same class `className`.

The body of a transform function is similar to a normal Java function. It refers to the object being transformed using whatever variable name is defined in the header, and it refers to the new object being constructed as `this`. It initializes all fields of the new object. During the initialization, the transform function may read other objects.

To illustrate how to write transform functions, let us consider an example. Suppose a schema has two classes, `Employer` and `Employee`. They have the following representations:

```
Employer {
   String name;
   String address;
}

Employee {
   String name;
   int salary; // monthly salary
   Employer employer;
}
```

In the new schema, both class representations of `Employer` and `Employee` are changed. The `address` field is removed from `Employer` and the `salary` field of `Employee` now represents the yearly rather than the monthly salary of the employee. The new classes are called `NewEmployer` and `NewEmployee`, respectively:

```
NewEmployer {
   String name;
}

NewEmployee {
```

```
    String name;
    int salary; // yearly salary
    NewEmployer employer;
}
```

The transform function for `Employer` is straightforward. We simply copy the `name` field and ignore the `address` field:

```
NewEmployer (Employer e) {
    name = e.name;
}
```

Note that our system assumes full encapsulation, i.e., application code must access fields of an object through its methods. However, we allow transform functions to directly access private fields of the object being transformed.

In our system, transform functions must explicitly initialize all fields of the new object, even if initialization is trivial like the assignment of `name` in the transform function for `Employer`. As part of future work, the system could provide *default transform functions* for such simple transforms. Systems like O_2 [10] and OTGen [17] already provide such facilities.

3.1.2 Type Incorrect Assignments

Now let us consider the transform function for `Employee`. One might want to write it as follows:

```
NewEmployee (NewEmployee e) {
    name = e.name;
    yearly_salary = e.monthly_salary * 12;
    employer = e.employer; // type incorrect assignment!
}
```

The first assignment copies the `name` field from the old `Employee` object and the second assignment derives the yearly salary from the monthly salary. However, the assignment,

```
employer = e.employer;
```

is not type correct. The type of the left-hand side variable is `NewEmployer` whereas the type of the right-hand side variable is `Employer`.

So how do we initialize the `employer` field? Here are some approaches that do not work:

1. Let the transform code do the assignment. This does not work not only because of type inconsistency, but also because the transform function might call a method on the object referred to by the field later as part of the transform.

2. Assign a default value to the uninitialized field. This does not work because the resulting new object might not satisfy its representation invariant leading to problems when methods are called on it.

3. Transform the field before assigning to it. This does not work because in some cases it might cause a cycle of transforms that never terminates. As an example, consider a circular list whose interface and implementation are being changed: where there used to be just one value at each link, now there are two. The representation of `List` used to be

   ```
   int val;
   List next;
   ```

   The upgrade changes it to

   ```
   int val1;
   int val2;
   NewList next;
   ```

   As we transform some list `x` whose `next` field points to some list `y`, we cannot transform `y` as part of the transform of `x` because this would cause a cycle of transforms, (i.e., as part of running the transform on `y`, we need to first transform `x`.)

Our solution is to delay the assignment until the very end of the transform, when no more transform code is running. To achieve this, we extend the syntax of transform functions as follows:

```
NewClass (OldClass x) {
   // initialize new object of NewClass using x
    ...
} assign {
    field_1: expr_1;
    field_2: expr_2;
    ...
}
```

The `assign` clause is optional. It identifies the type incorrect assignments that need to be done at the end of the transform. The body of the `assign` clause is a list of ⟨`field, expr`⟩ pairs. For each ⟨`field, expr`⟩ pair, `field` is a field of the new object being constructed, belonging to some new class C_{new}, `expr` is of type C_{old}, and the class upgrade $\langle C_{old}, C_{new}, TF \rangle$ is part of the upgrade.

Using the extended syntax with type incorrect assignments, we can now write the transform function for `Employee` as follows:

```
NewEmployee (Employee e) {
   name = e.name;
   yearly_salary = e.monthly_salary * 12;
} assign {
   employer: e.employer;
}
```

Allowing type incorrect assignment solves the problem we had at the beginning of this section in a way that ensures that the code of the transform functions can not use the object incorrectly. Furthermore, the type inconsistency is never visible to the application code because those fields would have been transformed by the system before any application accesses them.

3.1.3 Type Checking

Transform functions are fed into a preprocessor for type checking. This processor has relaxed typing rules to cope with the type incorrect assignments. The transform functions can refer to both old and new classes in the upgrade and must be type consistent with respect to the old and new schema.

We would also like to check the following properties of the transform function:

- Well-definedness, i.e., a transform function should not modify an object besides the one it transforms.
- Termination. In particular, creating objects belonging to old classes affected by the upgrade or calling other transform functions might cause cycles of transforms.

Unfortunately, checking the above properties is impossible in general. Some algorithm can be used to *conservatively* approximate this checking and warn the programmer if anything might be wrong, but it cannot be precise. A good programmer should keep the above restrictions in mind when he defines an upgrade.

3.2 Simulation Methods

In Section 2.6.5, we mentioned two ways to deal with the rep exposure problem caused by the transaction snapshot approach. The first solution is to define "upgrade-aware" transform functions that deal with this problem; the second solution is to have a way to impose an ordering on transforms. This section discusses how to support upgrade-aware transform functions.

"Upgrade-aware" transform functions goes hand-in-hand with *simulation methods*, which are not needed in the general query solution. Simulation methods have the same interface as the methods in the old classes affected by the upgrade but their implementation is different. Before we get into the details for simulation methods, we first use the following example to illustrate why upgrade-aware transform functions alone does not solve the rep exposure problem.

Suppose that at time t1, object x and object y are both affected by the current upgrade but have not been transformed yet. y refers to x and x refers to another object z, which is not affected by the upgrade but is read by the transform function of x, i.e, it is a tf-read object. Furthermore, x is not encapsulated within y but z is encapsulated within x. This situation is shown in Figure 3-1. At time t2, x is accessed by an application transaction and therefore is transformed to x' via transform function Tx as shown in Figure 3-2. Suppose that x and x' still both refer to object z. Another application transaction modifies z to z_mod at time t3 via x', as shown in Figure 3-3. At time t4, object y is accessed by an application and transformed to y' by transform function Ty, as shown in Figure 3-4.

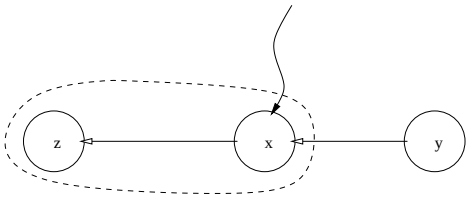

Figure 3-1: At t1, y refers to x, x refers to z

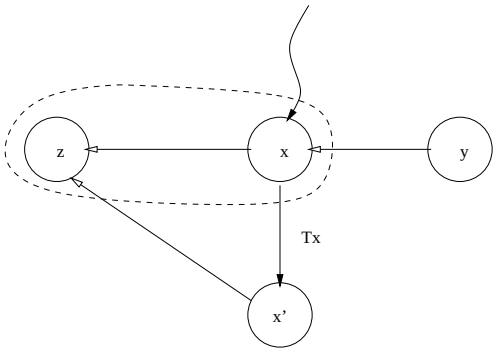

Figure 3-2: At t2, x is transformed to x'

Suppose that Ty calls some method m of x. Furthermore, suppose m returns some information about z. In the upgrade snapshot approach, z's unmodified state is

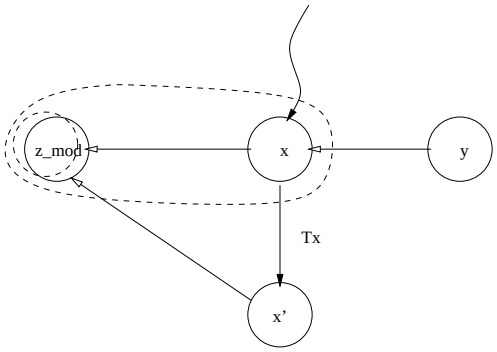

Figure 3-3: At t3, z is modified to z_mod

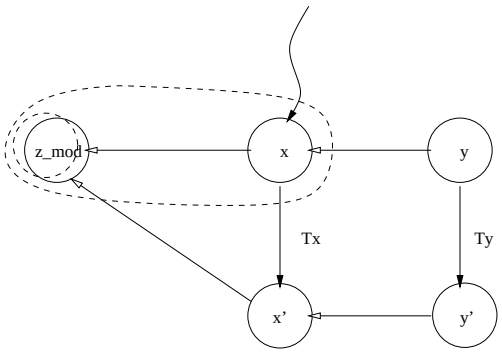

Figure 3-4: At t4, y is transformed to y' using x

preserved. Figure 3-5 shows what it would look like in the upgrade snapshot approach at time `t4`. Here, method `m` of `x` uses `z` instead of `z_mod`. However, in the transaction snapshot approach, since the state of `z` is not saved before it is modified, `m` uses `z_mod`. This might break some rep invariants of `x` that is assumed by the implementation of `m`. Note that in this case, making `Ty` upgrade-aware does not solve the problem. This is because `Ty` relies on the proper returning of `m` and `m` may not return properly because its original implementation cannot cope with the broken invariant.

Our solution is to provide another implementation of `m` that deals with the possible broken invariant. We call this additional implementation the simulation method of `m`. In a way, simulation methods are the analog of upgrade-aware transform functions:

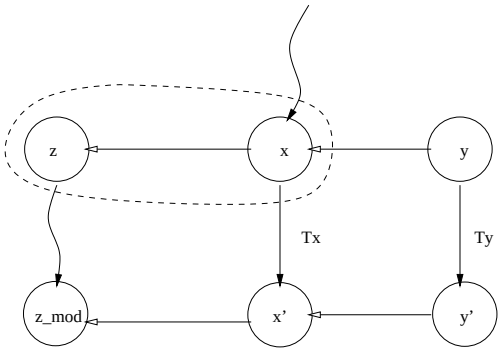

Figure 3-5: In upgrade snapshot approach, z's state is preserved when it is modified

the return value of a simulation method is used by the calling transform function whereas the result of an upgrade-aware transform function is used by applications. Both simulation methods and upgrade-aware transform functions are needed for the system to function correctly in the transaction snapshot approach.

The definition of upgrades must be extended to include simulation methods. More specifically, the class upgrade tuple now becomes

$$\langle C_{old}, C_{new}, TF, SM \rangle \tag{3.1}$$

where SM is the code for simulation methods.

Simulation methods have the same signatures as the methods they simulate. However, they have the following properties:

- The implementation of a simulation method has access to the old object of the class being upgraded, e.g., to `x` in Figure3-5. Note that a simulation method does not need to access the new state of the transformed object to work properly. Using the previous example, if the new implementation of `y'` depended on the new state of `x'`, `y'` would have been modified in the same application transaction that modified `x'`. Since it was not included in that transaction, it is not important for the simulation method for `m` to reflect the new state of `x'`. Therefore, it does not have to access `x'`.

- The simulation methods are *hidden* from applications and are only accessible to transform functions.
- After an object is transformed, the old snapshot of the object is accessed through its simulation methods instead of the original methods.

3.3 Completeness of Upgrades

As mentioned in Section 2.2, upgrades should be complete to guarantee the consistency of the database. Complete upgrade provides replacement code for all classes affected by the upgrade. In this section, we discuss what it means for a class to be affected by an upgrade.

Class upgrades are either compatible or incompatible. A class upgrade

$$\langle C_{old}, C_{new}, TF, SM \rangle \tag{3.2}$$

is *compatible* if C_{new} is a subtype of C_{old} ([20]). For example, C_{new} may provide a different implementation but preserve the same interface for C_{old}. Classes that use C_{old} do not needed to be upgraded in the case of a compatible upgrade.

However, the subclasses of C_{old} may be affected by the upgrade. For example, suppose a `protected` field of C_{old} is used by a subclass; then if this field is removed in C_{new}, the subclass must be changed to to cope with this.

A class upgrade is *incompatible* if C_{new} is not a subtype of C_{old}. The incompatibility includes changes in behavior even if there is no change in the interface.

When C_{old} is changed incompatibly, subclasses of C_{old} are affected even if all fields are private. Here are some possible ways to upgrade the subclasses of C_{old}:

1. They could be changed to become subclasses of C_{new}.
2. They could be changed to become subclasses of the superclass of C_{old}.
3. They could be changed to become subclasses of some other class.

In all of these cases, the behavior of the subclasses must be changed to be the same as those of their new superclass ([20]).

Furthermore, all code using C_{old} and its subclasses should be checked to see if it is affected by the upgrade. A complete upgrade provides the recompiled code for all code affected by the upgrade.

Chapter 4

Thor

In this chapter, we first give an overview of JPS [6], a version of Thor [3] that our implementation is based on. We then describe various components of Thor focusing on the ones that are most relevant to our implementation.

4.1 Overview

Thor provides persistent storage for a large number of objects. It is intended to be used in a distributed environment with many servers and clients. Thor has a client-server architecture. Objects are stored on reliable servers called *Object Repositories* (OR's). Each server has a persistent root. All objects reachable from the persistent root or from an application are persistent and the rest are garbage.

Applications run on client machines using cached copies of persistent objects. The part of Thor that runs on the client machines is called the *front end* (FE).

An application interacts with Thor by starting a *session*. Each session consists of a sequence of *transactions*; a new transaction starts after the previous one ends. A transaction invokes methods on Thor objects and ends with a request to commit or abort. A commit request may fail and cause an abort; if it succeeds, Thor guarantees that the transaction is serialized with respect to all other transactions and that all its modifications to the persistent objects are recorded reliably. If the transaction aborts, Thor discards all its modifications and guarantees that it has no effect on the

persistent state of the system.

Each object in Thor has a unique identity, a set of methods, and a state. The state of an object is encapsulated and is accessed only through its methods. An object may contain references to other objects, even those in other OR's. Thor objects are implemented using a type-safe language. This language is Java [14] for the version of Thor that we are using. Previous versions used Theta [5, 21].

In the following sections, we describe the most important components of Thor. FE and client, OR and server are used interchangeably. We emphasize the FE side of the system, which is where our implementation runs, while keeping the description of the rest of the system brief.

4.2 Object Format

Servers store objects on disk in pages. To improve performance on both servers and clients [28, 13], objects in Thor are kept small.

4.2.1 Object Reference

Objects are small because object references (*orefs*) are 32 bits in Thor. Orefs refer to objects at the same OR; objects point to objects at other ORs indirectly via *surrogates*. A surrogate is simply a small object containing the OR number of the target OR and the oref inside that OR.

The first 22-bit *pid* of an oref identifies the page containing the object and the next 9 bits (*oid*) is an index into in the *offset table* of that page. The offset table maps oids to addresses within that page. This indirection is important because it allows the servers to compact objects within a page independently from other servers and clients, e.g., during garbage collection. The last bit of an oref is used to indicate if an object is *swizzled* in the client cache. This is explained below.

4.2.2 ROT

It is not practical to use orefs as object references in the client cache. This is because each pointer deference would require a table lookup. Therefore, Thor uses *pointer swizzling*: an oref is translated to a pointer to an entry in an indirection table (ROT, for resident object table) and the entry points the target object. Indirection allows the FE to move and evict objects from client cache with low overhead.

ROT entries are as depicted in Figure 4-1. Conceptually, each object's ROT entry contains three parts: a pointer to the dispatch vector (DV) of the object's class, a pointer to the fields of the object and some header information including the object's oref and a reference count for garbage collection. The dispatch vector is shared by all objects of the same class whereas the fields are unique to each object. Fields are represented as an array of slots of either object references or actual data.

Object references are orefs when the page containing the object is first fetched into the persistent cache. An object reference is *swizzled*, i.e., replaced by a pointer to the ROT entry corresponding to this object, when it is first traversed. Thus later pointer traversal is sped up. The last bit of the oref is set to 0 to indicate that the object reference is swizzled.

The ROT may contain persistent objects as well as non-persistent objects. A non-persistent object is indicated by a zero oref in the ROT entry.

4.2.3 Class Objects

For each class in Thor, there is a class object. The class object has an oref, just like a regular object. Every non-surrogate object has a field containing its class oref. The class object contains all the information about the class, such as orefs of superclasses, dispatch vectors, etc. When an object is installed into the ROT, its class object is looked up from its oref and a pointer to the dispatch vector of the class object is put into the ROT entry[1].

[1]In the current implementation of Thor, class objects are non-persistent objects initialized at boot time and *pinned* in the client cache.

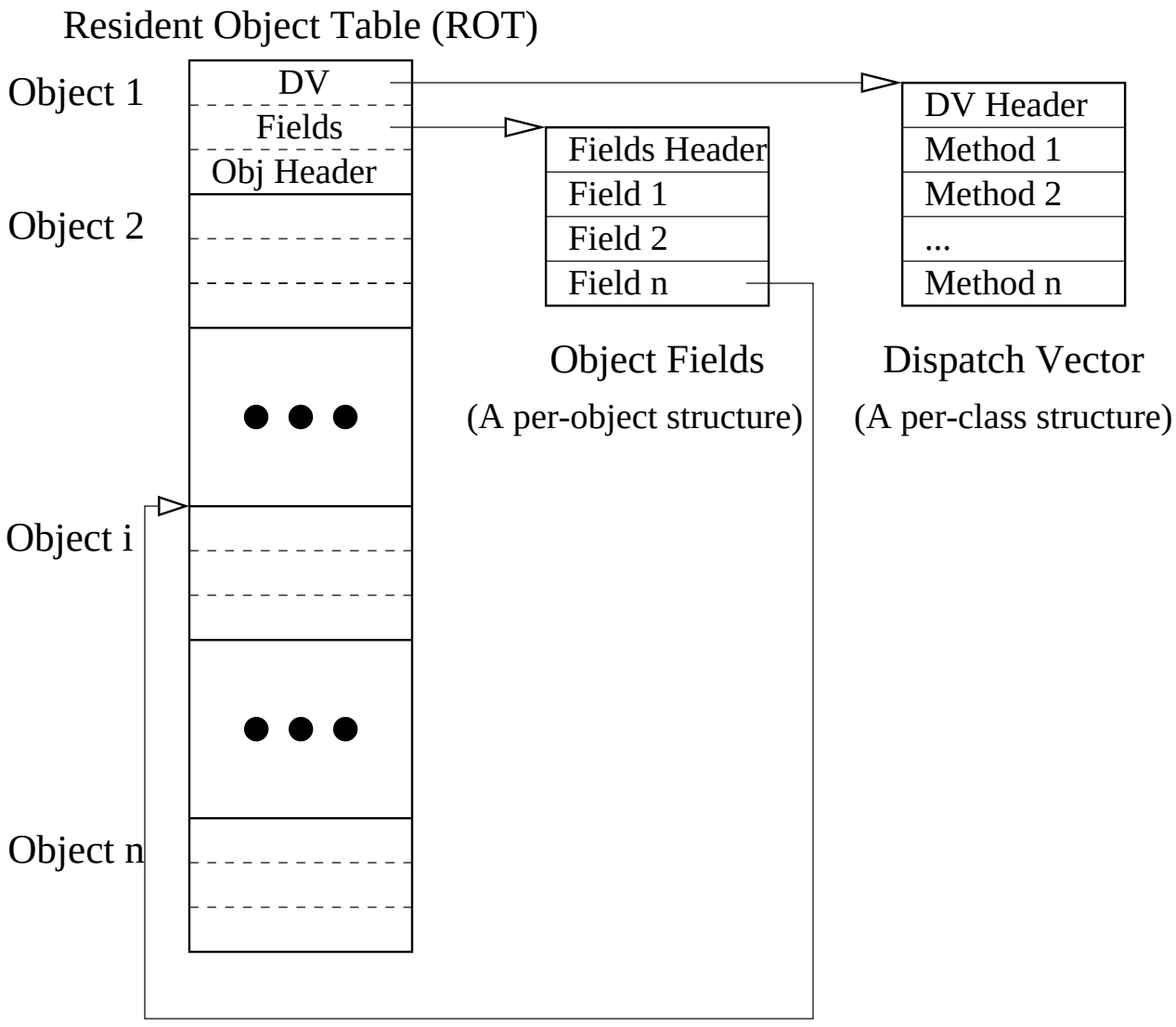

Figure 4-1: ROT Object Layout

4.3 Transaction Management

Thor uses *optimistic concurrency control* to avoid communication between clients and servers. The client runs a transaction assuming that reads and writes of the cached objects are permitted. When the transaction is ready to commit, the client informs the OR about reads and writes of the transaction. The server determines if the transaction can commit by checking if it is serializable with respect to other transactions. If the transaction used objects from multiple servers, a two-phase commit protocol is used. If the transaction can commit, modifications to objects are made persistent; otherwise, the client is told to abort. *Invalidation messages* are sent to the clients that cache stale copies of modified objects. Clients evict those objects from the cache upon receiving the message and abort their current running transaction if it used any invalidated objects. Invalidation messages are piggybacked on other messages sent to the client to reduce overhead.

The *transaction manager* (TM) at an FE keeps track of the ROS (read object set) and the MOS (modified object set) when an application is running. It also keeps an *undo log*, which records the states of objects when they are first modified. When a commit is requested, the TM sends the ROS, MOS and NOS (new object set, the set of newly persistent objects) to an OR. If the commit request is granted, the TM makes the newly created objects persistent and prepares for the next transaction. Otherwise, the TM undoes the effect of the transaction using the undo log.

The TM also processes invalidation messages. It discards invalid objects from the cache and determines if the current transaction used those objects (by looking at its ROS and MOS). If it did, the TM aborts the transaction.

The OR performs validation checks and sends invalidation messages. Details can be found in [1], [12], and [2].

Experimental results in [1, 12] show that this scheme out-performs adaptive call-back locking, which is considered to be the strongest competitor, in all reasonable environments and almost all workloads.

4.4 Client Cache Management

Thor uses *hybrid adaptive caching*(HAC) [22] for managing its client cache. HAC combines page caching with object caching to achieve both low overheads and low miss rates. HAC partitions the client cache into page frames and fetches entire pages from the server. To make room for an incoming page, HAC does the following:

- selects some page frames for *compacting*,
- discards the *cold* objects in these frames,
- compacts the *hot* objects to free one of the frames.

HAC selects page frames for compaction based on the current statistics: pages in which locality is high remain untouched whereas pages with low locality are compacted.

Experimental results in [22] show that HAC out-performs other object storage systems across a wide range of cache sizes and workloads. It performs substantially better on the expected workloads, which have low to moderate locality.

4.5 Server Storage Management

Thor uses *object shipping* because we use HAC and also because we discard invalid objects from client cache. At commit time, the clients send the modified objects to the servers that must eventually write them back to their containing pages to preserve clustering. To reduce the overhead of installation of new objects, Thor uses the MOB [11]. The MOB (for modified object buffer) is a volatile buffer used to store recently modified objects. The modifications are written back to disk lazily as the MOB fills up and space is required for new modification.

The MOB architecture combines the benefits of both object caching and page caching while reducing the overhead by avoiding installation reads and taking up less storage than what is required by page caching. Furthermore, Thor guarantees

reliability by writing committed transactions to persistent transaction logs before inserting them to MOB.

Experimental results in [11] show that for typical object-oriented database access patterns, the MOB architecture out-performs the traditional page-based organization of server memory that is used in most databases.

4.6 JPS

In the original Thor, objects are implemented using the Theta [5, 21] language. JPS [6], a later version of Thor, replaces Theta with Java. Like Theta, Java is type-safe and object-oriented.

To preserve the good performance of the original Theta-based system, we did not replace the runtime system with a standard Java interpreter. Rather, a modified bytecodes-to-C translator (TOBA) is used to generate code that interfaces appropriately with the Thor runtime environment. We then run an optimizing C compiler on the generated C code to produce native machine code. This is illustrated in Figure 4-2.

Figure 4-2: JPS compiler structure

JPS provides a clean functional interface between its runtime system and the code produced by TOBA. The translator-generated code uses this interface, while the JPS runtime system implements this interface with a set of C macros. The generated code does not need to know all the details of how the underlying system is implemented. For example, the generated code does not need to know how to check if an object is in cache or how to fetch it from an OR if it is not. The macros take care all these details.

Experimental results in [6] show that JPS performs almost as well as the original Theta-based Thor.

Chapter 5

Implementation

This chapter describes how we extended Thor to support schema evolution.

5.1 Basic Approach

Our implementation is based on the lazy transaction snapshot approach described in Section 2.6.4. In this approach, the system keeps upgrade snapshots of transformed objects and transaction snapshots of other objects. Transform functions run on these snapshots. Objects are transformed on demand and applications only operate on the most up-to-date objects.

Since the OR is the bottleneck of the system, modification to the OR is minimized. Therefore, most changes are done at the FE side of Thor. Our implementation assumes there is only one OR that is connected to all FEs. This OR is also the *upgrade OR*, which stores and processes all the upgrades.

The rest of this chapter goes into more detail about how upgrades are supported by the implementation of Thor.

5.2 Upgrade Check-in

We assume that a programmer defines an upgrade and saves the code in some files. These files contain the new class definitions, transform functions, and simulation

methods for the old classes. The classes and simulation methods are defined in Java whereas the transform functions are written in the Java-like syntax that we described in Section 3.1.1.

A user *checks in* or *initiates* an upgrade by starting an *upgrade transaction* with the name of the directory where the upgrade files are stored. As part of this transaction, the system finds the upgrade code contained in the files and compiles it.

Recall from Section 4.6 that Thor uses the standard Java compiler to compile Java source into bytecode and then uses a modified bytecode-to-C translator, TOBA, to get C code. An optimizing C compiler is then used to produce native machine code.

This structure cannot be used directly as it is because the Java compiler takes regular Java code whereas our transform functions are written in the special syntax that allows type incorrect assignments. To get past Java compiler without type errors, we can use a translator to translate the special syntax into regular Java code. The TOBA translator is then modified to recognize and undo the changes that the Java translator made so that the type incorrect assignments are made. Figure 5-1 illustrates this approach.

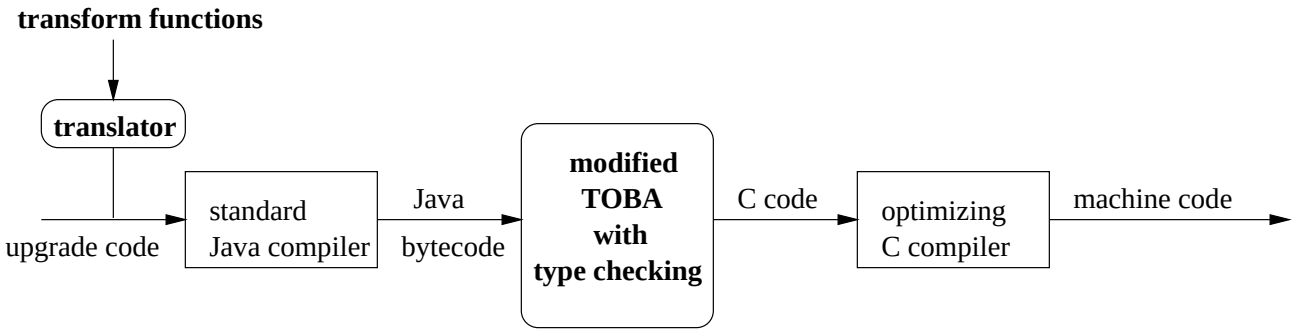

Figure 5-1: Modified JPS compiler to support upgrades

The modified TOBA translator is also modified to type check the upgrade code to ensure the type consistency, and to some extent, completeness of the upgrade.

If the upgrade code fails the type check, the system rejects the upgrade by aborting the upgrade transaction. Otherwise, the system assigns the upgrade a unique version

number. The system keeps a global counter. This counter is incremented every time an upgrade is checked in. This counter is used as the version number for that upgrade.

Class objects are initialized as part of the upgrade transaction. Recall from Section 4.2.3 that in Thor a class object contains all the information about a class, such as pointers to dispatch vector and superclasses, etc. To support schema upgrades, a class object is enlarged to contain the following information:

version_num is the version number of the upgrade that this class is part of.

next_class is a pointer to the class object of the next version.

tf is a pointer to the code of the transform function that transforms objects of this class to the class pointed to by `next_class`.

sim_dv is the simulation dispatch vector for the simulation methods of the class.

The last three fields are initialized to `NULL` when a class object is first created. They are changed to point to the non-`NULL` values when the next upgrade that affects this class is installed as described in Section 5.3. Because a class object is a per-class data structure, increasing its size to contain upgrade information only imposes small space overhead.

Eventually, the FE running the upgrade makes an *upgrade object* and commits it at upgrade OR, which stores information about the all upgrades. An upgrade object includes the following items:

- version number;
- initialized class objects;
- compiled code; and
- a manifest of all class upgrades in the form of a list of $\langle ClassOref_{old}, ClassOref_{new}, TF \rangle$ pairs.

Upon receiving an upgrade commit request, the upgrade OR serializes the upgrade with respect to all previous upgrades. Once an upgrade is checked in at the upgrade

OR, it needs to be propagated to the other FEs. In a single-OR implementation, the upgrade OR is the only OR. It can piggyback the current upgrade version number to all the FEs connected to it. Section 7.2.1 discusses upgrade propagation in a multiple-OR system.

5.3 Upgrade Installation

When an FE learns about a version number that is higher than the highest number it knows so far, it needs to install the current upgrade. After fetching the upgrade object from the upgrade OR using its version number, the class objects affected by the upgrade are identified via the list of class upgrades. Their `next_class`, `tf` and `sim_dv` fields are initialized to point to their counterparts in the upgrade object.

Next, the FE checks if the current transaction used any objects whose classes are affected by the upgrade. If it did, the transaction is aborted; otherwise, the FE performs a *ROT scan* (described below) and the transaction is continued. A simpler approach to implement this is to abort the current transaction even if it did not use any objects affected by the upgrade. Assuming that upgrades occur rarely, this approach is also acceptable. In either case, a ROT scan is performed immediately.

During a ROT scan, each entry in the ROT is examined and entries pointing to objects affected by the upgrade are changed to point to NULL. The purpose of the ROT scan is to ensure the *ROT Invariant*:

> *When an application runs, all objects pointed at by ROT entries reflect all upgrades that have been installed at the FE.*

In later sections, we will show how the ROT Invariant is preserved during the execution of a transaction.

The ROT scan is an expensive operation because the ROT is likely to contain many entries. However, we assume that upgrade installation is rare and thus the cost of the ROT scan is negligible. After the ROT scan, the installation of the upgrade is complete and the FE can start running applications conforming to the new schema.

5.4 Version Check

The essence of lazy upgrades is to transform objects on-demand. Therefore, the system needs a mechanism to detect an object that needs to be transformed when a transaction is running and "trap" for a transform, if necessary. We call this the *version check*. As the name suggests, the version check examines the version number of an object to determine if it is the right object to use for the running transaction and if not, appropriate actions are taken.

The first question we need to answer in the implementation is *when* to do version checks. The naive approach of checking version on every object access obviously poses too much runtime overhead. One key observation is that an object must be in the ROT in order to be used. Thus, we can fold the version check into the installation of ROT entries, which occurs whenever an object not in the ROT is accessed. However, checking versions *only* on ROT installation is not enough. Since the ROT is shared by both applications and transforms, an object in the ROT that is the right version for an application might not be the right version to use for a transform. For example, an application transaction might trigger a transform of object `o` to the current version `o'`. When the transform finishes, `o'` is installed in the ROT. A later transform on some other object `t` might want to read the untransformed version of `o`. In this case, `o'` would be the wrong version to read for this transform function.

One approach to solving this problem is to have a ROT scan (described in Section 5.3) before running any transaction of a different version than the previous one. However, this approach is not practical for performance reasons. As mentioned before, a ROT scan is a time-consuming operation and an application may access many objects and trigger many transforms. If a ROT scan must be performed at the beginning of each transform, ROT scans may need to be done very frequently. This would severely slow down the system.

Our approach is to check the version of an object even if it is already installed in the ROT, if we are running a transform. Our rational for this choice is: 1) the version check is only done during the running of transforms. The cost of the version

check is small compared to certain overheads already associated with transforms; and 2) checking if a transaction is a transform or an application can be done quickly by moving the mode bit into a register.

The last point about the version check concerns where version numbers of objects are stored. Recall from Section 5.2 that class objects contain version numbers. Given an object, we can get to its class object and hence its version number. The alternative approach of adding a version number field to every object was considered but was rejected for its impact on the space overhead.

5.5 Transaction Management

This section describes how the transaction manager (TM) at the FE side of Thor is modified to support schema evolution.

5.5.1 Data Structures

Below are the main data structures that we added to the TM to support upgrades. Other auxiliary structures are introduced later as we go into more details.

is_tf is true if the FE is running a transform transaction, and false if it is running an application transaction. Our FE is single-threaded which means the TM is either running a transform or an application.

version_num is the version number of the transaction that the FE is currently running. If a transform is running, this number is equal to the version number of the upgrade that the transform is part of. Otherwise, the version number is the version number of the most recent upgrade that this FE has installed.

curr_log ROS, MOS and undo log of the currently running transaction, which could be either an application or a transform.

a_curr holds the state (see below) of an interrupted application.

t_stack is a stack of states of suspended transform transactions.

t_log records state of all completed transform transactions. They are indexed in the order they are completed.

The state information in the `a_curr`, `t_stack` and `t_log` structures is captured in the data structure, `t_record`, which contains the following fields:

ver_num is the version number of the upgrade this transaction is part of.

tlog_index is the index of the last entry in `t_log` before this transaction was interrupted.

post_state contains the new states of objects modified by the interrupted transaction.

ros, mos, undo log are the ROS, MOS and undo log of transaction.

Figure 5-2 illustrates the initialization of a TM at the beginning of an application.

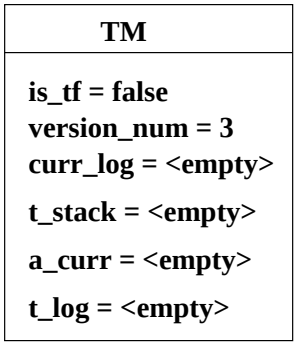

Figure 5-2: Initial state of TM at the beginning of an application transaction when the system is running at version 3

5.5.2 Interleaving Transforms and Applications

Unlike in the original Thor, applications may interleave with the transforms. The implementation of managing interleaving transactions follows the design described in Section 2.6.1 and 2.6.4.

Triggering Transforms

As an application is running, the TM keeps track of the objects that the application reads and writes in `curr_log` as illustrated by Figure 5-3. A transform is triggered if an object is found to belong to an earlier schema during a version check, as described in Section 5.4.

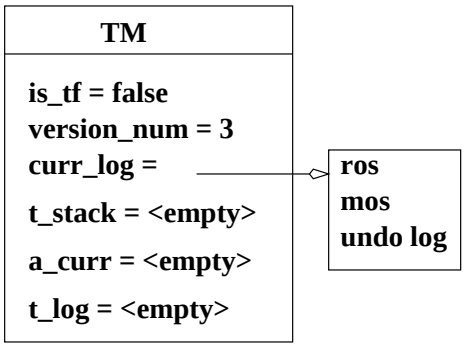

Figure 5-3: State of a TM running an application transaction

Before the transform starts to run, the state of the application transaction must be saved. In particular, the state recorded in `curr_log` is saved in `a_curr`. Because we are using the transaction snapshot approach, transforms should run on the transaction snapshot of the enclosing application. Therefore, for all objects modified by this application, we save pointers to their new states in `post_state` and restore their original states by pointing their ROT entries to the corresponding values in the undo log. If `t_log` is non-empty, the index of the most recent entry is saved in the `tlog_index` field of `a_curr`.

Finally, the TM is initialized to run the transform. Figure 5-4 illustrates the state of the TM right before a transform function of version number 2 is about to run. Note that the state of the application is saved in `a_curr`; and the `curr_log`, `is_tf` and `version_num` are all initialized to run the transform function.

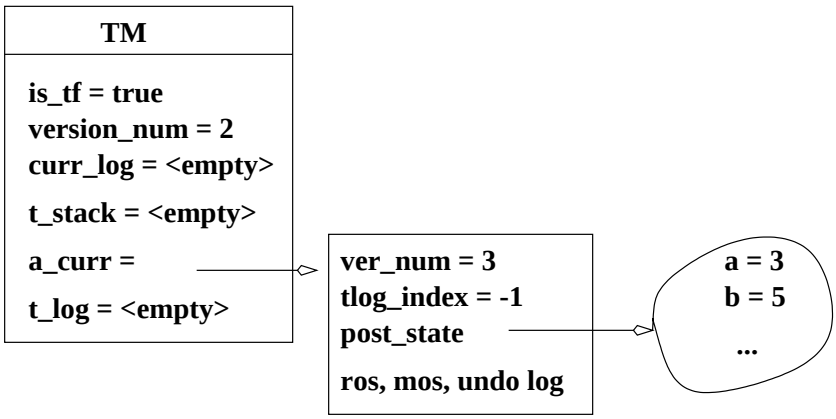

Figure 5-4: State of a TM with application suspended to run a transform function

Running a Transform Function

Given an object to be transformed, the system finds the code for its transform function from its class object and starts a transform transaction to run it. A transform transaction runs just like an application transaction except for more frequent version checks as described in Section 5.4.

During the version check in a transform function, there are two scenarios where the object being checked is not the right version for the transform:

1. The version of the object is greater than the version needed by the current transform. This means that the object has been transformed earlier by some other transform function. In this case, we find the earlier version of the object (see Section 5.5.4). Since this object has already been transformed beyond that version, simulation methods must be used to access the object. This is achieved by pointing the DV pointer in the ROT entry to the DV of the simulation methods, which can be found via the class object.

2. The version of the object is earlier than the version needed by the current transform. In this case, we invoke a *nested* transform transaction to transform this object. The transition from the current transform to the nested transform is similar to the transition from an application to a transform. The only difference

is that the suspended transform is pushed onto the transform stack, `t_stack`, instead of `a_curr`. Figure 5-5 shows the state of a TM with nested transforms. As shown in the figure, there are two completed transforms (T0 and T1) and a stack of transforms on `t_stack`. The TM is currently running a transform of version 0.

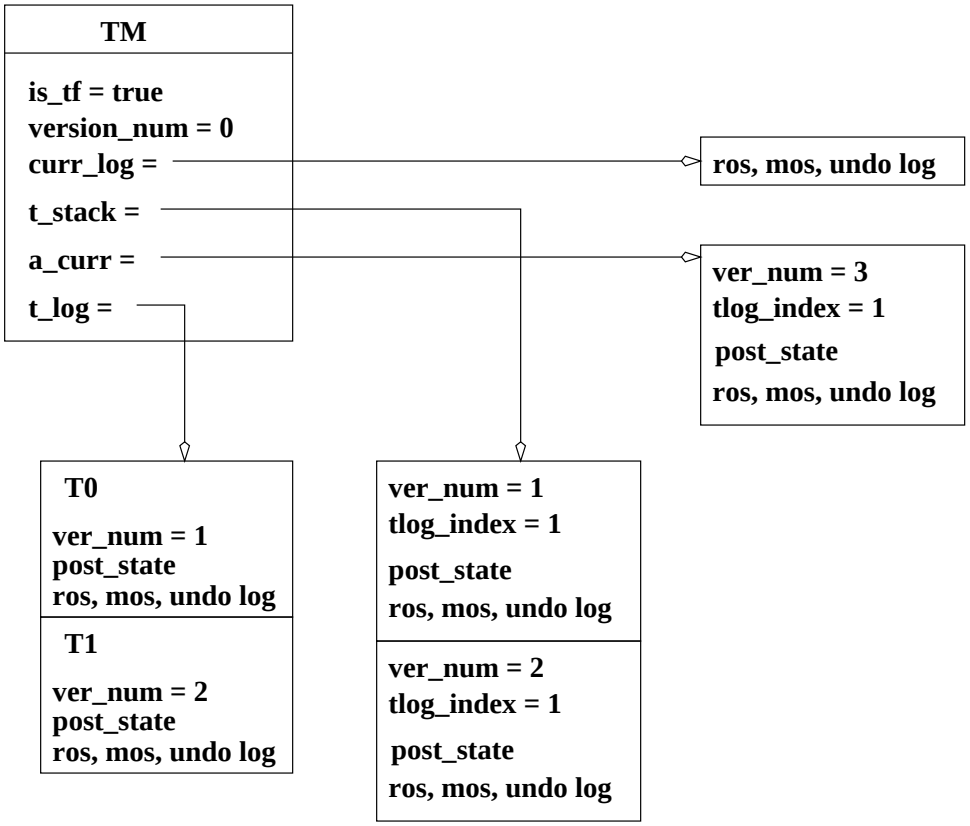

Figure 5-5: An example of TM with multiple, nested transforms

In both of the cases above, we may install objects of versions earlier than the most up-to-date version into the ROT. To preserve the ROT Invariant stated in Section 5.3, we discard these entries before the application is restored.

When a transform transaction finishes running, its state is appended to the end of the `t_log` and the suspended transform or application transaction that triggered it is restored.

Restoring a Suspended Transaction

If `t_stack` is not empty, the transaction on top of the stack is popped off and restored; otherwise, it is the suspended application in `a_curr` that is to be restored. Before the transaction can be restored, the TM checks if it conflicts with any completed transforms, using the following rules. Let A be the transaction to be restored and T1 a completed transform:

- A.ros $\cap$ T1.mos $= \emptyset$

  If this is non-empty, it means that A read an earlier value of an object modified by T1. Note that if our transform functions are well-defined, this intersection is always empty because T1 would only modify the object that it transforms and A could not have read the untransformed version of that object.

- A.mos $\cap$ T1.mos $= \emptyset$

  If this set is non-empty, it means that A modified an object that is later modified by T1. Again, this intersection is always empty if the transform functions are well-defined.

Note that A does not need to be compared with all the completed transforms in the `t_log`, but only those that were last triggered by A since it was suspended (the index of the first such transform is stored in `A.tlog_index`). This is because the last restoration of A has already validated A against transforms in the `t_log` up to the point of `tlog_index`.

If the transaction can be restored, we swing the ROT entries of modified objects to point to the values pointed to by `post_state` of this transaction when it was interrupted. The rest of the system (`version_num`, `is_tf`, etc.) is initialized from the saved information and the interrupted transaction continues to run.

If the transaction cannot be restored, we need to undo its effects. For each object modified by this transaction, we overwrite its state pointed to by its `post_state` with its current value. We then swing the ROT entry of the object back to that pointed by `post_state`. This abort cascades to all suspended transactions until there is no

transaction left. Note that at the end of this abort, we still commit the transforms left in `t_log`, if any.

5.5.3 Transaction Commit

The original Thor TM on the FE side only handles commit of one transaction at a time. Now it still commits at the end of each application transaction, but the application might have triggered transforms, which also need to be committed. According to our lazy semantics described in Chapter 2, all transforms are committed before the application that triggered them. This means that all transforms recorded in `t_log` should be committed in the order that they completed and before the application transaction.

A naive approach is to commit each transform in `t_log` as a separate transaction. This approach is not very practical because the commit process is time consuming. Our more efficient approach is to commit all completed transforms as one big transaction, called T. Depending on the result of committing T, the application transaction, A, is committed or aborted as usual. The disadvantage of this approach is that if there are many transforms in T, it is more likely for T to conflict with other FEs and therefore abort. One possible solution for this problem is to divide T into smaller chunks and commit them separately if T gets too big. This is part of our future work. For now, all transforms are committed as one big transaction, T, followed by application transaction, A.

In the following sections, we assume that `t_log` is non-empty at the end of the transaction; if it were empty, it means that no transforms were triggered during the application transaction and therefore the normal commit protocol of Thor is used to commit or abort the application.

Commit Request

The TM computes the commit set for T as follows:

$$T.ros \;=\; \bigcup t\_log[i].ros \tag{5.1}$$

$$T.mos = \bigcup t\_log[i].mos \tag{5.2}$$

$$T.post = \begin{cases} (x\_oref, x\_value), & \text{if } A \text{ aborts} \\ (x\_oref, x\_value'), & \text{where } x\_oref \in T.mos \text{ and} \\ & x\_value' = \begin{cases} A.undo(x\_oref), & \text{if } x\_oref \in A.mos \\ x\_value, & \text{if } x\_oref \notin A.mos \end{cases} \end{cases} \tag{5.3}$$

T.mos and T.ros are sets of orefs containing the write set and read set, respectively. T.post contains the values of the write set. If we know that A must abort before we commit T, T.post contains the current values of objects in T.mos, because A has already been aborted. Otherwise, if A modified any object after it was transformed by T, T.post should contain the pre-modified state of that object.

The commit request also includes a new object set (NOS). The NOS is generated during the process of recursive unswizzling[1] of all modified objects. Objects in this set are made persistent if the commit succeeds.

After the commit request for T is generated, it is sent to the OR. The transaction snapshot approach requires that we save the snapshot of the old objects after transforming them. Section 5.5.4 shows a way to do this using surrogates.

Commit Response

The OR validates T against transactions from other FEs as described in Section 4.3 and sends back a commit response. Depending on the response, there are several scenarios:

- If T cannot commit, both A and T should abort. T.undo is used to undo the effect of T and A.undo is used to undo the effect of A, if it has not already been aborted.

- If T can commit, send commit request of A to the OR. If A must abort, undo the effects of A but install the new objects of T. Otherwise, install new objects of both T and A.

[1] Unswizzling is the opposite of swizzling. It replaces a reference to a ROT entry with its oref. All fields of an object are unswizzled before it is committed at the OR.

There are a couple of ways to make the implementation more efficient. They are part of our future work.

- Rather than sending the commit request for T and then A, we could have sent both in one request if A has not aborted. This approach saves time on commit; but it requires modifying the implementation of the OR.
- Instead of aborting T when the commit response is negative, we could try committing entries in `t_log` one by one, starting from the first entry, until an abort occurs.

5.5.4 Upgrade Surrogates

The Transaction snapshot approach requires that we save the snapshots of all transformed objects. This means that after an object is transformed, we cannot replace it with the new object; instead, the old and new objects must remain distinct. In addition, there must be a way to go from one version to the other. There are two cases:

- We must be able to go to the old object from the new object if an earlier version is needed by a transform function.
- We must be able to go to the new object from the old because we want to "snap" the pointer that points to the old object to point to the new one if the old object has already been transformed.

A doubly-linked list seems to be the natural solution. One way to implement this is shown in Figure 5-6. Each object has two additional fields: `next`, which points to the next object in the upgrade chain; and `prev`, which points to the previous version of the same object. The `prev` of the first version and the `next` of the last version are NULL. The chain is extended when the object of the most recent version is transformed. Given an external object reference to any of the object in the chain, one can navigate to any other object in the chain. This approach requires major changes to the current implementation of Thor, which assumes an object format without the

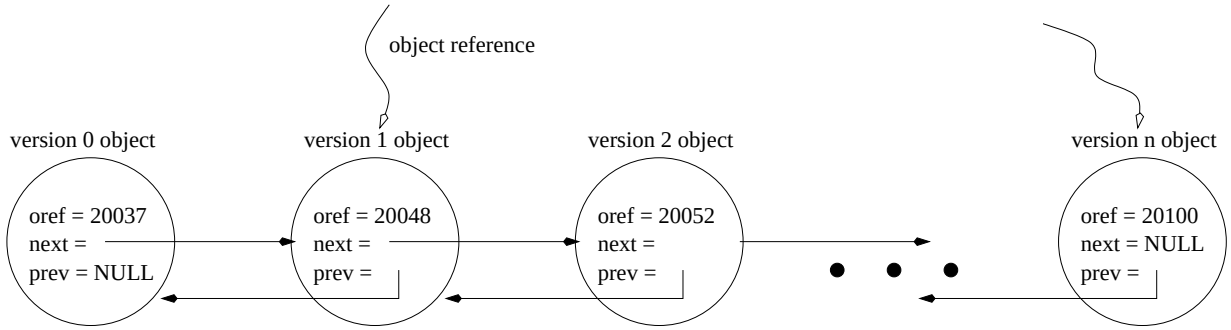

Figure 5-6: Doubly-linked chain structure of transformed objects

additional pointers. A "smart" garbage collection scheme is needed to remove the older versions in a chain of objects.

To minimize the impact to the existing system, we propose the structure of doubly-linked chain of *upgrade surrogates*, as shown in Figure 5-7. As mentioned in Section 4.2.1, a surrogate is a small object that is used to point to an object at a different OR. It contains an OR number and the oref inside that OR. An upgrade surrogate is a surrogate that is extended to contain pointers to its neighbors. Once an object is transformed, a surrogate is created in its place to point to both the old and the new object. An external object reference that previously pointed at the old object now points at its surrogate.

This approach does not need to change the format of non-surrogate Thor objects and therefore requires the minimal changes. Like the first approach, we need an efficient garbage collection to get rid of the old objects. The main disadvantage of this approach is that the locality of objects is affected. As a remedy, we try to commit the old and new objects on the same page as the surrogate wherever possible.

The linked chain structure in Figure 5-7 is how the OR stores the old and new versions of objects. At the FE side, traversing the surrogate chain can be inefficient. Instead, a pointer that initially points at an upgrade surrogate is "short-circuited" to point to the snapshot directly when the pointer is traversed, and we maintain a map between the surrogates and their corresponding snapshots. Thus, one can

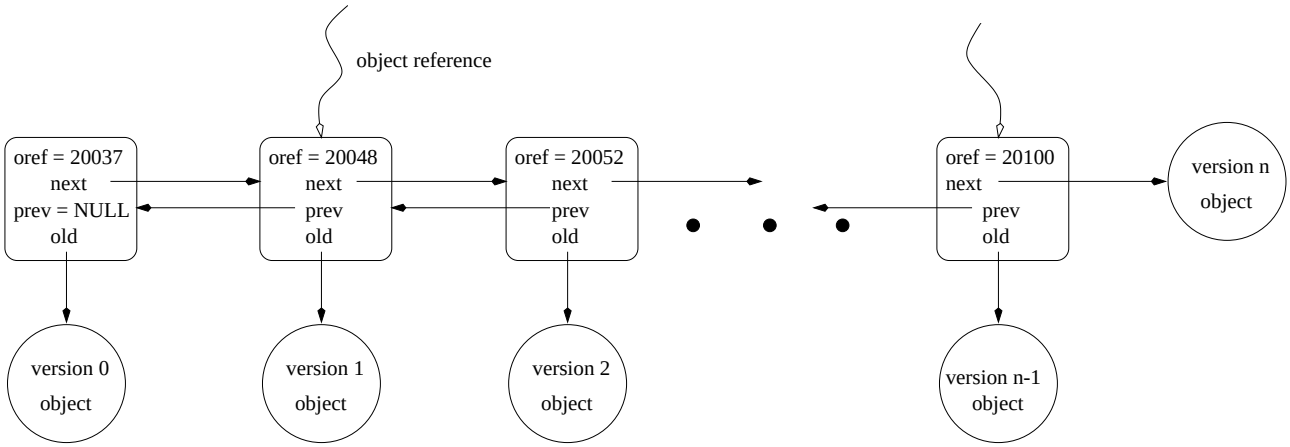

Figure 5-7: Doubly-linked chain structure of surrogates

still navigate on the chain of surrogates and get to any version of an object. For objects that are newly transformed, we delay creating upgrade surrogates for them until commit time; a data structure is therefore needed to save these mappings.

Extra work needs to be done at commit time. First, upgrade surrogates are created and initialized to point to all objects that are transformed during the application transaction. Existing surrogates are changed to point to these new surrogates. Secondly, during unswizzling, any short-circuited reference is changed to contain the oref of the corresponding upgrade surrogate.

Chapter 6

Related Work

The topic of schema evolution has been heavily studied in object-oriented database research [4, 25, 10, 24, 17, 18, 29, 27, 7, 23, 26]. Roughly speaking, previous research extends along the following dimensions: program compatibility, types of schema changes, schema consistency and the implementation of database transformation. In the following sections, we will discuss how previous work and our work approach these issues.

6.1 Program Compatibility

Depending on whether old programs are supported after a new schema is in place, all schema evolution systems take either the *conversion* or *versioning* approach.

In a conversion approach, a new schema replaces the old schema. After the installation of the new schema, only programs conforming to the new schema are allowed to run. Orion [4], GemStone [25], O_2 [10], ObjectStore [24], OTGen [17], Woo [29], and our system belong in this category. The main disadvantage of this approach is the lack of support of legacy code.

In contrast, a versioning system keeps versions of all previous schemas and old programs can still run after a new schema is installed. Systems like Encore [27], Clamen [7], COLSQL [23] and Coast [26] take this approach. The disadvantage of versioning is the complexity of the implementation and the runtime overhead. For

the rest of this section, we give an overview of four prominent versioning systems.

Encore [27] is one of the earliest versioning systems. It keeps a *version set* for each class. An object is accessed through the *version set interface* of its class, which is a virtual class definition that contains the union of all attributes of the versions of the class. A handler is defined for every attribute of a class version that appears in the version set interface, but not in that version of the class. These handlers return default values when the attributes are accessed. There are two serious limitations with this approach: 1) if an evolution adds a new attribute to a class, an instance that was created by an earlier version of this class cannot store any new value associated with this attribute (unless it coincides with the default value provided by the handler); 2) attributes of the same name in different versions are assumed to represent the same information, thus it is impossible to represent a change in the semantics of the attribute between versions.

Clamen [7] proposed a scheme where the creation of a new version of a class results in a new version (or *facet*) of every instance of the class. Thus, an instance can have multiple facets and different programs access the object using different facets. This approach is similar to ours, where versions of objects are kept for the use of transform functions. However, in our system, versions are garbage collected when they are no longer needed by transform functions, whereas in his system, versioned objects persist forever. To reduce the storage requirement, he allows attributes to be shared among facets. Clamen's approach does not have Encore's limitations mentioned above, but now updates to one facet must be reflected in all other facets. Clamen suggested a delayed propagation scheme but since this system was never implemented, the exact mechanism of update propagation between facets is not clear.

COLSQL [23] lets users define *update* and *backdate* methods between class versions. Instances are not versioned as in Clamen's approach; instead, they are converted to the right version using the update or backdate methods when they are accessed. To address the problem of lost information during conversions, the system saves the value of an attribute that was dropped. When the object is converted back to the original version, this value, rather than some default value, is restored to that attribute.

Compared with Clamen's approach, COLSQL sacrifices time for space.

In Coast [26, 15], each application works on top of exactly one schema version and through each schema version, *sv*, a certain set of objects, called the *instance access scope* (*IAS(sv)*) of the schema version, is available. Forward and backward conversion functions can be defined between schemas to make objects created under one schema version visible to application on top of another schema version. Coast is similar to Clamen's system in that multiple versions of instances are kept in different *IAS*s; it is also similar to COLSQL because conversion functions are provided at the class level. The most innovative feature of Coast is its *propagation flags*. These flags are associated with each conversion function and can be switched on or off independently. They give the designer fine-grained control over how creation, deletion and modification of an instance in one *IAS* should be propagated to another *IAS*.

Since our system adopts the conversion approach, we focus our attention on systems that take the same approach in the rest of this chapter.

6.2 Types of Schema Changes

Different schema evolution systems offer different sets of allowed changes to an existing schema. Most changes can be put into two categories: 1) changes to class definition, i.e., adding, deleting and modifying attributes or methods; and, 2) changes to the inheritance graph, i.e., adding, deleting or changing classes and inheritance links. Table 6.1[1] lists schema changes offered by some of the systems we are going to discuss.

In the following discussion, we will distinguish between simple and complex transform functions. A *simple* transform function can only access the state of the object being transformed whereas a *complex* transform function can access objects other than the object being transformed.

Orion [4] offers a wide variety of operations that one can do to change the schema. However, the application of each operation is governed by a set of rules to preserve the structural invariants of the schema (e.g., class lattice, domain compatibility in-

[1] The format of this table is borrowed from [16]

variants, etc.). Since the set of rules is fixed, it necessarily restricts the power of schema modifications. For example, the domain of an instance variable can only be generalized. In addition, a new attribute can only be initialized with a default value. In contrast, our system allows users to provide transform functions that can be used to define arbitrary changes; in particular, we allow complex transforms.

GemStone [25] only offers a few schema operations, for example, it does not support any method or inheritance link manipulations. This limitation might have come from its lack of any invariant preserving rules, such as those defined in Orion, or invariance checking mechanism, as in O_2 [8]. GemStone inherits the limitations of Orion that we mentioned above.

O_2 [10, 30] does not allow modifications to the attributes or methods of a class, i.e., the name and domain of an attribute cannot be changed and the signature of a method cannot be changed. However, like our system, it allows users to define conversion functions where the initialization of attributes can use other attributes or their sub-objects. This feature is lacking in Orion and Gemstone. However, it is not clear from the literature if the conversion functions can invoke method calls in O_2, a feature that is supported by our system.

O_2 also supports *object migration*. There are two kinds of object migration. Note that O_2 only allow migrations to subclasses to avoid runtime errors.

- any particular object can be migrated to belong to any of its subclasses. This is done via a system method `migration()` associated with the root class `Object`, which takes the as argument the name of the target class.

- all or some of the objects of a class can be changed to belong to one of its subclasses. This is done via *migration functions*. A migration function (like a conversion function) has access to the state of the object being migrated; it is not clear if a migration function can access objects other than the one it migrates.

Our system does support the transform of objects of one class to belong to another class (not necessarily a subclass); but we do not provide the "filtering" mechanism

of migration functions to transform only a subset of objects of a class to different classes.

In addition to basic operations, ObjectStore [24] allows user-defined *transformer functions* to transform objects. However, transformer functions can only access the values of the old attributes of the object being transformed and cannot invoke method calls on them. ObjectStore also provides a convenient way to transfer objects of a class to one of its subclasses, called *instance reclassification*.

OTGen [17] automatically constructs a *transformation table*, where a default *transformation function* is provided for each modified class by comparing the old and new schemas. The default transformation functions can be modified by the user. As seen from Table 6.1, OTGen does not support any method modifications. However, OTGen provides the following general operations:

Initialization of variables By allowing users to modify the default transform function, OTGen allows arbitrary computation to be used to initialize attributes of the transformed objects. In particular, transformation functions can invoke methods on attributes as part of initialization. This is similar to our system.

Context-dependent changes OTGen allows boolean expressions to be attached to a transformation table entry. For instance, an object can be transformed to class C or class D, depending on the value of one of its attributes. This is a more generalized form of "object migration" of O_2, where there has to be a subclass relationship between C and D.

Sharing of information OTGen uses *shared expression* to support sharing of information among objects when attributes are initialized. A shared expression is evaluated once for each distinct set of argument values it is instantiated with. A table that is indexed by the argument values is used to look up the shared expressions. This functionality is convenient but not essential; our system can simulate it at the application level.

Note that OTGen provides the above functionalities on top of the core schema evolution operations via some special constructs. Our system currently does not support

Table 6.1: Types of changes supported by schema evolution systems

	Attribute			Method			Class			Inheritance Link		
	add	del	mod[a]	add	del	mod[b]	add	del	mod[c]	add	del	mod[d]
Orion	•	•	•[e]	•	•	•	•	•	•	•	•	
GemStone	•	•	•[f]				•	•[g]				
O_2	•	•		•	•		•	•	•	•	•	
ObjectStore	•	•	•	•	•	•	•	•	•	•[h]	•[i]	•[j]
OTGen	•	•	•				•	•	•	•	•	

[a] modify name and domain
[b] modify name and signature
[c] modify name
[d] modify superclass
[e] domain can only be generalized
[f] domain can only be generalized
[g] only when there is no instance of the class
[h] add only base classes
[i] remove only base classes
[j] changing between virtual and nonvirtual inheritance

either context-dependent changes or sharing of information. Extending our system to support context-dependent changes would increase our expressive power; supporting sharing of information is mainly for convenience.

In Woo [29], transform functions are defined in similar fashion as ours and therefore provide the same expressive power.

6.3 Schema Consistency

Different systems have different notions for what a *consistent* or *valid* schema is. In general, there are two aspects of schema consistency: structural and behavioral ([4, 30, 17]).

6.3.1 Structural Consistency

Structural consistency refers to the static invariants that a schema evolution should preserve. Most data models in the literature require the following invariants:

Class Lattice Invariant The class lattice (inheritance graph) is a rooted and connected directed acyclic graph (DAG).

Unique Name Invariant All class names must be unique; within a class definition, names of all attributes and methods must be unique, no matter whether they are defined or inherited.

Full Inheritance Invariant A class inherits all attributes and methods from all its superclasses.

Type Compatibility Invariant The type of an inherited attribute must be the type or the subtype of the attribute it inherits. Also the type of an attribute must correspond to a class in the class lattice.

Orion [4] and GemStone [25] ensure the above invariants by defining a set of rules for each schema update operation that they support. A particular rule is selected from this set when the operation is applied, so that the invariants would hold. For example, when the domain of an instance variable is changed, the rule says that it can only be generalized.

In OTGen [17], the checking for these invariants is built into the automatic system that generates default transform functions, although the transform functions can be overridden by the user.

O_2 [10, 30] uses an integrity checker [8] to check these invariants. It is similar to our approach where the checking is done as part of the type checking.

ObjectStore [24] and Woo [29] do not provide a description of how to check for the structural consistency.

6.3.2 Behavioral Consistency

Behavioral consistency refers to the runtime behavior of the schema evolution. Informally, it means that each method respects its signature and its code does not result in runtime errors or unexpected results. It is equivalent to the "completeness" of upgrades that we discussed in Section 3.3 of this thesis.

Behavioral Consistency was not addressed by Orion, GemStone, OTGen or ObjectStore. In our system and Woo's thesis, the completeness of upgrades is discussed. Some global analysis can be used to approximate the completeness of an upgrade.

Zicari [30] outlined an algorithm to check behavioral consistency in O_2. A method dependency graph is extracted by looking at the code of each method. In addition, extra information is kept about which portion of a class a method actually uses. The systems uses this dependency information to determine if a method might become invalid or need to be recompiled when a new upgrade is installed.

6.4 Database Transformation

As discussed in Chapter 2, there are two approaches to transform existing objects in the database once the upgrade is installed: immediate and lazy. If a system only allows simple transforms, whether the upgrade is propagated immediately or lazily does not affect the semantics; however, if a system allows complex transform functions, it must be done carefully to ensure the correct behavior.

This thesis identifies two problems that arise when implementing complex transform functions lazily. First, an object affected by the upgrade might be accessed by another complex transform function. In this case, the problem is that the object might be of an earlier or later version depending on when it is accessed. Secondly, an object not affected by the upgrade might be accessed by a complex transform function (we call these objects tf-read object in this thesis). In this case, the problem is that the object might have been modified by some application and the transform function might observe broken invariants.

Our thesis is the first to address both problems. To solve the first problem, we save versions of objects when they are transformed, so that we can find an earlier version if needed; in addition, we developed a mechanism to trigger nested transforms if a later version is needed. To solve the second problem, we proposed a solution that is a combination of "upgrade-aware" transform functions and "simulation methods". Now we discuss how other systems implement database transformation.

First, we look at systems that do not support complex transforms, such as Orion and GemStone. In Orion, deletion or addition of attributes is delayed until an object is fetched from disk to memory. In GemStone, all objects of the affected classes are enumerated, transformed and put back to the database. In both systems, since transforms are not complex, doing the transform in-place does not create any semantics problems.

Next, we look at the systems that do support complex transforms. ObjectStore takes an immediate approach, which is divided into two steps: instance initialization and instance transformation. During the first phase, a new object is created and initialized for each affected object. All pointers to the old object are swung to point to the newly created object. During the second phase, user-defined transformed functions are applied to the newly created objects. The system provides access to the values of data members of the corresponding old object for transform functions. At the end of the second phase, all old objects are discarded.

The pointer swinging of the initialization phase described above is problematic because it changes all pointers, including those contained within attributes of some untransformed objects to point to the new instances. These objects and their attributes may be accessed by some transform function during the transform phase later. Thus, these transform function would access the new objects instead the old. ObjectStore mentioned the use of exception handlers to handle illegal pointers during the initialization phase, but it is not very clear how it handles the problem we described here during the transformation phase.

In OTGen, objects are partitioned into collections. 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. This collection of objects is transformed the first time it is accessed with the new version of the server. For each collection of objects, the transform is immediate; from the point of view of the whole database, the transform is lazy. It is not clear from the paper [17] how exactly the transformation of objects is done, however.

In O_2, the user can choose whether to transform the database lazily or imme-

diately. O_2 physically retains the deleted or modified information in the object in a so-called *screened part*. Applications do not have access to the screened parts of objects, but conversion functions use the screened parts to perform the correct transformations. By dependency analysis of transform functions, the storage overhead of screened parts can be reduced by not saving fields that are not going to be accessed by transform functions. Furthermore, whenever an immediate transformation is launched, O_2 transforms all the objects in the database to conform to the most recent schema and screened parts of objects are dropped. By having screened parts, O_2 addressed the problem of complex transform functions accessing transformed objects. However, it did not address the problem with the tf-read objects.

Like O_2, objects are transform in-place in the work of Woo [29]. He recognized the same problem that O_2 solved. Rather than saving screened fields for objects, his system forbids the installation of upgrades affecting objects that can be read by the complex transform functions of any previous upgrades that has not retired. This method saves the space overhead but restricts the installation of upgrades. Furthermore, he did not address the tf-read problem either.

Chapter 7

Conclusions

7.1 Summary

The requirement for both good performance and sound semantics has made schema evolution a long-standing challenge for object-oriented database systems. The challenge primarily concerns when and how to transform existing objects in the database in the case of an incompatible upgrade. This thesis presents our solution to this problem.

First, we provide a simple interface for users to specify upgrades as a collection of class upgrades. Each class upgrade contains a transform function that converts an object of the old class into an object of the new class.

Next, the question of when to apply these transform functions is addressed. In order to preserve the availability of the system, we proposed a lazy upgrade model where object transformations are delayed until they are accessed by applications. To guarantee the consistency of the database, our model requires that the upgrades be complete and the transform functions well-defined.

The interleaving of applications and transforms poses some problems. More specifically, an object might belong to either an earlier or later version when it is encountered by a transform function. To cope with the former case, our model provides a mechanism for running nested transforms; to deal with the latter case, our model keeps snapshots of objects when they are transformed. Keeping snapshots of trans-

formed objects is not enough to prevent complex transform functions from observing broken rep invariants, however. For this problem, we proposed a solution where "upgrade-aware" transform functions and "simulation methods" are defined.

Finally, we implemented our approach in Thor, an object-oriented database system. The implementation was done with efficiency and performance in mind.

7.2 Future Work

There are many ways our work can be improved or extended. This section suggests a few paths for future work.

7.2.1 Multiple-OR Implementation

Our current implementation assumes a single OR system where the upgrade OR is the only OR. As part of the future work, this implementation can be extended to support multiple ORs. An interesting research problem is how to propagate the upgrade information from the upgrade OR to the FEs in an efficient and timely fashion. If a lazy upgrade snapshot approach were taken to transform the database, all ORs used need to be informed about the upgrade when it is committed at the upgrade OR because of the tf-read objects. With the transaction snapshot approach an epidemic algorithm can be used to propagate the upgrades among ORs and FEs.

7.2.2 Upgrade Limitations

The lazy upgrade model that we proposed in this thesis has some limitations. In particular, our model does not handle two kinds of transforms:

Multiple-object Transforms There are transforms where a *group* of objects must be transformed all at once. Such transforms are not supported by our current model because our transform functions are limited to transforming only one object at a time. As part of future work, we could extend the transform functions to transform multiple objects at once.

Ordered Transforms Since transforms run independently in the current model, there is no way we can impose any order to them. As mentioned in Section 2.6.5, sometimes we want to run transforms in a certain order to get the correct semantics. One of the main reasons for our proposal of "upgrade-aware" transform functions and "simulation methods" in this thesis is to get around this problem. Allowing ordered transforms may eliminate the need for this.

A general query mechanism as described in Section 2.6.5 could give the user more control over the order of transforms. An efficient implementation of this idea is worth exploring.

7.2.3 Garbage Collection

As mentioned in Section 5.5.4, a good garbage collector is essential to the performance of our system. Normal GC does not help in removing the old versions in our system because we use doubly-linked chain of surrogates in our implementation.

One possible approach is to perform some global analysis of the code and detect which classes of the previous upgrade are not used. Objects belonging to those classes can then be garbage collected. To speed up the upgrade of the database, the garbage collector can also proactively transform objects by sending them to the upgrade OR when the system is not busy. The garbage collector used need to be incremental and distributed in order to work well in a large, distributed system.

Bibliography

[1] Atul Adya. Transaction Management for Mobile Objects Using Optimistic Concurrency Control. Master's thesis, MIT, Cambridge, January 1994.

[2] Barbara Liskov Atul Adya, Robert Gruber et al. Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks. In *Proceedings of ACM SIGMOD*, 1995.

[3] L. Shrira B. Liskov, M. Castro and A. Adya. Providing Persistent Objects in Distributed System. In *Proceedings of the European Conference for Object-Oriented Programming*, 1999.

[4] J. Banerjee and W. Kim. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proceedings of the ACM SIGMOD 1987 Annual Conference*, pages 311–322, 1987.

[5] Mark Day Barbara Liskov, Dorothy Curtis et al. Theta Reference Manual. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, Feb 1994. Available at `http://www.pmg.lcs.mit.edu/papers/thetaref/`.

[6] Chandrasekhar Boyapati. JPS - A Distributed Persistent Java System. Master's thesis, MIT, Cambridge, September 1998.

[7] S. Clamen. Type Evolution and Instance Adaptation. Technical Report CMU-CS-92-133, Carnegie Mellon University, 1992.

[8] C. Delcourt and R. Zicari. The Design of an Integrity Constraint Checker (ICC) for an Object-Oriented Database System. In *Proceedings of European Conference on Object-Oriented Programming*, 1991.

[9] T. Meyer F. Ferrandina and R. Zicari. Implementing Lazy Database Updates for an Object Database System. Technical Report 9, Goodstep, 1994.

[10] F. Ferrandina and G. Ferran. Schema and Database Evolution in the O2 Object Database System. In *Proceedings of the 21st VLDB Conference*, 1995.

[11] S. Ghemawat. *The Modified Object Buffer: a Storage Management Technique for Object-Oriented Databases.* PhD thesis, MIT, Cambridge, MA, 1995.

[12] Robert Gruber. *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases.* PhD thesis, MIT, Cambridge, MA, 1997.

[13] R. N. Mayo J. C. Mogul, J. F. Barlett and A. Srivastava. Performance Implications of Multiple Pointer Sizes. In *USENIX 1995 Tech. Conf. on UNIX and Advanced Computing Systems*, 1995.

[14] Bill Joy James Gosling and Guy Steele, 1996.

[15] S. Lauteman. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proceedings of the 5th International Conference on Database Systems for Advanced Applications*, pages 323–332, 1997.

[16] T. Lemke. Schema evolution in OODBMS: A selective overview of problems and solutions. *Intelligent Database Environment for Advanced Applications IDEA*, 1994.

[17] B. Lerner. Beyond Schema Evolution to Database Reorganization. In *Proceedings of ECOOP/OOPSLA*, 1990.

[18] B. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. Technical Report 96-044, Computer Science Department, University of Massachusettes, 1996.

[19] Barbara Liskov and John Guttag. *Program Development in Java.* Addison-Wellesley, 2001.

[20] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[21] B. Liskov M. Day, R. Gruber and A. C. Myers. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. In *Proceedings of Object-Oriented Programming, Systems, Languages and Applications*, pages 156–168, 1995.

[22] Barbara Liskov Miguel Castro, Atul Adya and Andrew C. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *16th ACM Symposium on Operating System Principles*, 1997. Available at `ftp://ftp.pmg.lcs.mit.edu/pub/thor/hac-sosp97.ps.gz`.

[23] S. Monk. Schema Evolution in OODBs Using Class Versioning. *SIGMOD RECORD*, 22(3), 1993.

[24] Object Design Inc. *ObjectStore Advanced C++ API User Guide Release 5.0*, 1997. Also available at `http://www.unik.no/~mdbase/OS_doc_cc/user1/9_strcnv.htm`.

[25] A. Otis Robert Bretl, D. Maier and J. Penney. The GemStone Data Management System. In *Object-Oriented Concepts, Applications and Databases.* Ed. W. Kim and F. Lochovsky, Addison-Wellesley, 1989.

[26] P. Eigner S. Lautemann and C. Wohrle. The COAST Project: Design and Implementation. In *Proceedings of the 5th International Conference on Deductive and Object-Oriented Databases*, pages 229–246, 1997.

[27] A. Skarra and S. Zdonik. The Management of Changing Types in an Object-Oriented Database. In *Proceedings of Object-Oriented Programming, Systems, Languages and Applications*, 1986.

[28] S. J. White and D. J. Dewitt. Quickstore: A high performance mapped object store. In *ACM SIGMOD Int. Conf. on Management of Data*, 1994.

[29] Shan Ming Woo. Lazy Type Changes in Object-Oriented Databases. Master's thesis, MIT, Cambridge, February 2000.

[30] Roberto Zicari. A Framework for Schema Updates in an Object-oriented Database System. In *Proceedings of the 7th International Conference on Data Engineering*, 1991.