Lazy Type Changes in Object-Oriented Databases

by

Shan Ming Woo

B.A., University of Cambridge, UK (1997)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2000

Author ....
Department of Electrical Engineering and Computer Science
January 14, 2000

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

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

Lazy Type Changes in Object-Oriented Databases

by

Shan Ming Woo

Submitted to the Department of Electrical Engineering and Computer Science
on January 14, 2000, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Object-oriented databases are gaining importance among modern applications. They provide powerful execution environments for applications to safely share persistent objects. However, maintaining applications built on top of object-oriented databases has long been a challenge.

Since the behavior of object-oriented database applications is partly composed of the behavior of the persistent objects they access, to update these applications, it is sometimes necessary to modify types that govern the behavior of the persistent objects and to transform objects to reflect the type changes. But it has been difficult to carry out type changes correctly without affecting the availability of the underlying database. This is because related type changes have to be executed atomically with respect to application transactions to avoid run-time type errors. However, transforming the potentially large number of persistent objects affected by the type changes takes time. As large-scale databases and mission-critical databases become more common, the need for a mechanism to carry out type changes without sacrificing database availability becomes more urgent.

This thesis demonstrates that type changes can be carried out correctly without affecting the availability of the underlying database. It proposes a general model to do so and presents an efficient implementation design of the model on a distributed object-oriented database management system. The model is based on the technique of transforming objects lazily when they are about to be accessed. Thus, rather than paying for the cost of executing a related set of type changes all at once, the model spreads the cost effectively over time and distributes it fairly among applications. Therefore, disruption to database availability is avoided.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

Acknowledgments

I would like to thank my thesis supervisor, Professor Barbara Liskov, whose patient guidance has been crucial to the completion of this thesis. Barbara's insightful advice and comments have been indispensable. I am greatly impressed by her ability to identify important issues quickly and accurately.

I am also indebted to my office-mate, Miguel Castro. The complexity of Thor frustrated me in countless occasions. Very fortunately, Miguel was there to help every time. Other members of the Programming Methodology Group have also given me much-needed support and encouragement. They have made it enjoyable to be part of the group. Thank you.

Although I did most of the work related to this Thesis under an MIT research assistantship, I would not have been able to work on it without prior support from the Dah Sing Bank 50th Anniversary Fellowship. The fellowship allowed me to explore various thesis topics so that I could pick one that interested me the most. Also, it gave me the opportunity to meet Mr. David S.Y. Wong, Chairman of Dah Sing Financial Holdings, Ltd. of Hong Kong. Mr. Wong's entrepreneurial spirit and his determination to develop Hong Kong's information technology industry deserve high respect.

Before I came to Cambridge, Massachusetts, where MIT is located, I have studied at Cambridge, United Kingdom. This was made possible by the Sir Edward Youde Memorial Scholarship for Overseas Studies. The scholarship not only allowed me to study computer science at the University of Cambridge, UK, but it also let me travel outside Hong Kong and southern China for the first time.

The academic progress of my brother and myself has always been our parents' highest priority. I hope we have lived up to their expectation.

I am grateful to my fiancee, Vivian Pan, whose love and care have given me energy to finish this thesis. Gratitude is also extended to my numerous friends in the Boston area. They have added color to my life outside the computer laboratory.

Contents

Chapter 1

Introduction

1.1 Background

Object-oriented databases provide modern applications with powerful execution environments. They offer applications the benefits of the object-oriented programming paradigm as well as the ability to share persistent data. An application can store data as objects into a database for access in a later session. Other applications can also access the stored objects concurrently or at a later time.

The sharing of data among applications is safe because of concurrency control and data encapsulation. In a database environment, the operations carried out by each application are grouped into transactions. Concurrency control ensures that concurrent transactions do not interfere or conflict with each other. Data encapsulation guarantees that the data stored in an object are always accessed in a correct way by restricting the access to the object's methods. Thus, the only way for an application to access the data is to call the object's methods.

Despite the benefits mentioned in the above, maintaining applications built on top of an object-oriented database has long been a challenge. These applications differ from other applications in that their behavior is partly composed of the behavior of the persistent objects they access. Therefore, to update these applications, it is sometimes necessary to change the behavior of objects in the database.

The behavior of an object is defined by its type. Besides behavior, the type of an object also defines its representation. To change the behavior of the objects of a type, the definition of the type has to be changed and the objects have to be transformed from the old representation to the new representation. This operation is called a *type* *change* in this thesis. The term is also used to refer to the codes that carry out the operation.

Changing the definition of one type may affect other types. Thus, one type change can lead to other type changes. In this thesis, a set of related type changes is called an *upgrade*. When an upgrade is executed, all its type changes are carried out simultaneously.

Executing an upgrade is not a trivial task. The execution has to be atomic with respect to application transactions. This ensures that the objects accessed by an application transaction belong to a consistent type configuration.

One simple way to achieve atomicity in the execution of an upgrade on a database is to shut down the database before changing any type definitions and transforming any objects. The shutdown aborts all running transactions. Therefore, no application transactions will be concurrent with the upgrade execution and thus the execution is atomic.

However, the above execution model is not suitable for all databases because it disrupts the availability of a database during an upgrade execution. For large-scale databases, an upgrade can affect a large number of objects and take a long time to execute. This implies that the disruption to database availability can be long. Therefore, the above execution model is not desirable. For mission-critical databases, any disruption to database availability is simply not acceptable. Therefore, the above execution model cannot be used.

1.2 Thesis Contributions

This thesis presents an alternative execution model called the *Lazy Transformation Model*. The Lazy Transformation Model delays the transformation of an object until it is about to be used. Therefore, the workload of an upgrade is spread over time and distributed among running applications. This efficient distribution of workload allows database availability to be preserved. Besides the above advantage, the Lazy Transformation Model is also generally applicable. Unlike some previous work, it imposes only a weak restriction on upgrades to guarantee type safety during their execution.

This thesis also describes how the Lazy Transformation Model can be implemented without unduly affecting the performance of a database management system and applications. The implementation design is based on Thor [LAC+96, LCSA99], a distributed object-oriented database management system designed to support large-scale and mission-critical databases.

1.3 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 explains the static aspects of type changes and upgrades.
- Chapter 3 presents three upgrade execution models: the Snapshot Model, the Regular Transaction Model, and the Lazy Transformation Model. Studying the Snapshot Model and the Regular Transaction Model makes it easier to understand the Lazy Transformation Model.
- Chapter 4 gives an overview of Thor and describes the implementation design of the Lazy Transformation Model on Thor. The design highlights several issues that have to be addressed by any implementations of the model.
- Chapter 5 discusses previous work related to this thesis.
- Chapter 6 concludes this thesis with a summary of its contributions and a discussion of future research topics.

Chapter 2

Type Changes and Upgrades

As discussed in Chapter 1, the behavior of applications built on top of an object-oriented database is partly composed of the behavior of the persistent objects they access. Therefore, to update these applications, it is sometimes necessary to change the behavior of objects in the database. This is achieved by defining and executing *type changes* and *upgrades*.

This chapter discusses the static aspects of type changes and upgrades. Their execution is covered in Chapter 3. But first, the notions of *objects* and *types* used in this thesis have to be defined.

2.1 Objects and Types

In an object-oriented database, data are stored in *objects*. Each object has a *unique identity* that distinguishes it from other objects in the database. The unique identity of an object is used when references to the object are made.

Besides a unique identity, each object also has a *type* defined in an object-oriented programming language supported by the database management system. The type of an object defines a set of *methods* that can be invoked on the object and a set of *fields* in which the object's data are stored. These methods and fields are part of the object. The methods determine the object's behavior and the fields determine the object's representation. Besides methods and fields, each type also defines a set of *constructors* that are used to create new objects of the type. However, unlike methods and fields, constructors do not belong to any objects. They belong to their type.

From the above discussion, each object σ can be regarded as a triple $\langle id, T, d_T \rangle$

where id is the object's unique identity, T is the object's type, and d_T is the object's data stored in the fields defined in T. It is an invariant that an object's representation always matches its type. Therefore, $\langle id, T, d_{T'} \rangle$ is not a legal object if $T \neq T'$.

This thesis assumes that objects are *strictly encapsulated*. This means that the fields of an object can only be accessed by its methods. Thus, applications or the methods of other objects have to access the fields of an object indirectly by calling its methods. The indirection protects the integrity of the object's data. Such protection is especially important when the type of an object can be changed at run-time.

The notion of encapsulation used in this thesis is stronger than that used in most object-oriented programming languages. In these languages, encapsulation means that the fields of an object can only be accessed by the methods of its type. Thus, the fields of an object can be accessed directly by the methods of another object of the same type. This kind of access is not allowed when strict encapsulation is imposed.

Assuming strict encapsulation may seem to be a limitation. However, the ideas proposed in this thesis are still applicable when types are defined in a programming language that does not enforce it. This is achieved in two different ways.

First, if a field of an object can be accessed directly by codes other than the object's methods, implicit methods can be added to the object's type to account for any access. For example, consider a type `student`. Suppose that each `student` object has a field called `address` that can be read or modified directly by codes other than the object's methods. By regarding any access of the field in a `student` object as being made through a `get_address` method or a `set_address` method defined implicitly in `student`, the object becomes "strictly encapsulated". Implicit methods have to be considered together with actual methods when type changes are defined. In the above example, if a type change removes `address` from `student`, it effectively removes the two implicit methods.

Second, programmers can refrain from writing codes that violate strict encapsulation even when the programming language does not enforce it. The loss in run-time efficiency of their codes due to extra method calls will be compensated by increased ease when they define type changes. Strict encapsulation decouples object representation and object behavior. Thus, it is possible to change object representation without changing existing object behavior. This makes the lives of programmers easier when they need to change object representation. This point will be apparent when the compatibility of type changes and the completeness of upgrades are discussed later in this chapter.

This thesis also assumes that types are defined in a *strongly typed* object-oriented programming language. Therefore, in the definition of a type, each identifier has a declared type. In particular, if an identifier represents a reference to an object, the declared type of the identifier is called the *apparent type* of the reference and the type of the object referred to by the identifier at run-time is called the *actual type* of the reference.

In this thesis, a type S is said to *depend on* a type T if the definition of S contains identifiers with declared type T.

Types can be related to each other to form a *type hierarchy* or *supertype/subtype relation*. This thesis uses the behavioral notion of subtyping proposed in [LW94]. A type S is a *subtype* of another type T if S exhibits all the behavior of T. This implies that S has all the methods of T and that S's methods have the same syntax and compatible specifications with their counterparts in T. When S is a subtype of T, T is a *supertype* of S. The supertype/subtype relation is transitive. Therefore, when R is a subtype of S and S is a subtype of T, R is also a subtype of T. The relation is also reflexive, i.e. a type is both a supertype and a subtype of itself.

Because a subtype exhibits all the behavior of a supertype, an object of the subtype can be used wherever an object of the supertype is needed. Therefore, the actual type of an object reference may be different from the apparent type of the reference—the former may be a subtype of the latter. It is an error if the actual type of an object reference is not a subtype of its apparent type.

The purpose of using a strongly typed programming language is to enable compile-time type checking. Compile-time type checking detects possible errors in codes before the actual execution. Codes that pass compile-time type checking have the following *static type correctness property*:

> *On an identifier with declared type T, only type T methods are invoked.*

They also maintain the following *dynamic type correctness invariant*:

> *An object reference of apparent type T always refers to an object of type T or a subtype of T.*

Together, the static type correctness property and the dynamic type correctness invariant guarantee that whenever a method is invoked on an object at run-time, the object will actually have such a method, i.e. there is no *run-time type errors*.

This thesis is concerned with changing type definitions and transforming objects from one type to another at run-time. To avoid run-time type errors, it is essential that the dynamic type correctness invariant be maintained. How to do so is explained in the latter part of this chapter.

2.2 Basics of Type Changes and Upgrades

A type change changes the definition of a type and transforms objects of the type from the representation of the old definition to that of the new definition. Conceptually, a type change C is a triple $\langle T, T', f^T \rangle$ where T is the original type definition, T' is the new type definition, and f^T is a transform function that transforms objects of T into objects of T'. In this thesis, T is called the *pre-type* of the type change and T' is called the *post-type* of the type change.

A set of related type changes form an upgrade. An upgrade U is a tuple $\langle n, \{C_i\} \rangle$ where n is its serial number and $\{C_i\}$ is a non-empty set of type changes that constitute the upgrade. Upgrades are totally ordered by their serial numbers. The pre-types of the type changes in an upgrade are collectively known as the pre-types of the upgrade and similarly for the post-types.

Type changes are total. This means that each type change affects all objects of its pre-type. Therefore, the pre-types of an upgrade should all be distinct. Also, it does not make sense for an upgrade to change a type and transform its objects from one definition to another and then back to the original definition or from one definition to another and then to yet another definition. Therefore, the pre-types and the post-types of an upgrade should be disjoint.

An upgrade is a unit of execution. This means that all the type changes in an upgrade are carried out simultaneously when the upgrade is executed. The execution of type changes and upgrades is the subject of the next chapter. For the purpose of this chapter, it is sufficient to assume that the execution of an upgrade transforms all objects of its pre-types instantaneously and at the end of the execution, there are no more objects of its pre-types in the database.

After an upgrade has been executed, its pre-types become obsolete. This means that the post-types of an upgrade should not depend on or create objects of its pre-types. However, the reverse does not have to hold true. Sometimes the pre-types of an upgrade may depend on or create objects of its post-types if some of the post-types have been defined and used before the definition of the upgrade.

Upgrades are executed in the order specified by their serial numbers. Ordering upgrades frees programmers from worrying about the dynamics of type changes. When defining new upgrades, types, or applications, programmers can assume that all previously executed upgrades have taken effect. Therefore, if there is a series of changes for a certain type, the new upgrades, types, or applications have to handle only objects of the latest type definition but not any previous definitions.

2.3 Examples

This section describes a few example type changes. These examples are based on a type called `movie`. Here's a fragment of the definition of `movie` in a pseudo object-oriented programming language:

```
movie = type    %movie is declared to be a type
   int length;
      %length is an integer
      %it stores the length of the movie in milliseconds
   channel data;
      %data is a reference to a channel object
      %the object stores the movie data in some format
   int length()
   {  %effects: return the length of the movie in milliseconds
      return length;
   }
   stream output()
   {  %effects: create and return a stream of multimedia data
      ......
   }
   movie copy()
   {  %effects: create and return a copy of the current object
      ......
   }
   movie(int l, channel d)
   {  %this is a constructor
```

```
        %effects: initialize a new movie object
        length = l;
        data = d;
    }
end movie
```

Assume that `int` is a built-in type and that the `channel` type and the `stream` type have been defined appropriately. The `movie` type has two fields, three methods, and one constructor. It is an immutable type. The data stored in a `movie` object are passed as arguments to the constructor when the object is created.

There are a number of plausible type changes programmers may want to make to `movie`:

1. *Alteration of the object representation.* Programmers may want to store the movie data in another format to make use of the latest data compression technology or to comply with a new industrial standard.

2. *Addition of some methods and/or fields.* Programmers may want to store the description of a movie alongside the movie data and provide methods in the type to set and retrieve the description.

3. *Removal of some methods.* Programmers may want to remove the `copy` method to make sure that the copyright law is not violated.

4. *Amendment of some methods.* Programmers may want `output` to accept an argument of type `account` so that a fee can be charged to the account every time a stream is produced.

2.4 Transform Functions

The transform function is arguably the most important component of a type change. After all, it is the transform function that transforms objects from the pre-type to the post-type. During the execution of the type change, the transform function is used as a subroutine. This section explains how the transform function works.

A transform function acts on one object at a time. Consider a type change $\langle T, T', f^T \rangle$. The transform function f^T takes an object of T as its argument and returns an object of T' as its result. More precisely, when f^T is applied to an object

$\sigma_T = \langle id, T, d_T \rangle$, it returns an object $\sigma_{T'} = \langle id, T', d_{T'} \rangle$. Note that σ_T and $\sigma_{T'}$ have different types and can have different representations but they have the same identity. Before the execution of the type change terminates, $\sigma_{T'}$ will be used to replace σ_T. Chapter 3 discusses when the replacement happens and how it happens. After the replacement, any objects that referred to σ_T originally will refer to $\sigma_{T'}$. Since $\sigma_{T'}$ corresponds to σ_T, object references are "preserved" across type changes.

Strict encapsulation is imposed on transform functions as well as other codes. When the transform function f^T is applied to σ_T to produce $\sigma_{T'}$, f^T is regarded as both a method of σ_T and a method of $\sigma_{T'}$. This means that f^T can access the fields of σ_T and those of $\sigma_{T'}$ directly. However, f^T does not have this kind of privilege on any other objects. To access an object other than σ_T and $\sigma_{T'}$, f^T has to call its methods.

In a typical situation, when given the object σ_T, f^T will carry out the following steps to produce $\sigma_{T'}$:

1. Pre-process the fields of σ_T if necessary to retrieve the data stored in σ_T. This step may involve the creation of some auxiliary objects and the invocation of some methods on the auxiliary objects and objects referred by the fields of σ_T.

2. Create the object $\sigma_{T'}$ by calling a constructor of T' using some of σ_T's data, i.e. the fields of σ_T and the auxiliary objects, as arguments.

3. Store the rest of σ_T's data into the fields of $\sigma_{T'}$.

4. Return $\sigma_{T'}$.

To provide more concrete examples, let's consider the first two type changes in Section 2.3.

In the first type change, suppose that the post-type is called `pact_movie` and that the new format stores the audio data and the video data separately in two different channels. Thus, `pact_mov` is defined as follows:

```
pact_mov = type
    int length;
    a_channel audio;
    v_channel video;
    int length() {return length;}
    stream output() {......}
    movie copy() {......}
```

```
   pact_mov(int l, a_channel a, v_channel v)
   {  length = l;
      audio = a;
      video = v;
   }
end pact_mov
```

Assume that the types a_channel and v_channel are defined appropriately to store audio data and video data respectively. The same as movie, pact_mov is also an immutable type. The data stored in a pact_mov object are passed in as arguments to the constructor when the object is created.

So, when given an object σ of type movie, the transform function does the following:

1. Invokes some methods on the object referred by the field data of σ to retrieve the movie data.

2. Creates an a_channel object and a v_channel object by calling their constructors using the retrieved movie data as arguments.

3. Creates a pact_mov object σ', passing the field length of σ and the newly created a_channel and v_channel objects to the constructor as arguments.

4. Returns σ'.

In the second type change, assume that the post-type is called desc_mov and there is a built-in type text that is suitable for storing the description of a movie. Here's the definition of desc_mov:

```
desc_mov = type
   int length;
   channel data;
   text desc;
   int length() {return length;}
   stream output() {......}
   movie copy() {......}
   text getDescription() {return desc;}
   setDescription(text d) {desc = d;}
```

```
    desc_mov(int l, channel da, text de)
    {  length = l;
       data = da;
       desc = de;
    }
  end desc_mov
```

Since existing `movie` objects do not have descriptions, their `desc` field will be set to `null` when they are transformed. The transform function does the following when given a `movie` object σ:

1. Creates a `desc_mov` object σ' using `length` of σ, `data` of σ, and a null value of type `text` as arguments to the constructor. (In this case, the constructor simply assigns its arguments to the corresponding fields of σ'. Note that the objects referred by the arguments are not accessed.)

2. Returns σ'.

There is a distinction between the two transform functions described above: the first transform function accesses an existing object besides the one it is transforming (the `channel` object referred by the `data` field) but the second transform function does not. The first transform function is called a *complex transform function* while the second transform function is called a *simple transform function*. This way of classifying transform functions has been proposed by Ferrandina et al. in [FMZ94, FMZ$^+$95]. Note that the classification excludes any auxiliary objects created during the execution of a transform function. The importance of this classification is discussed in Chapter 3.

When defining a transform function that creates persistent auxiliary objects (e.g. those in the first transform function described above), programmers have to be careful about the types of the auxiliary objects created. For a type change $\langle T, T', f^T \rangle$, if a new object of type T is created whenever f^T transforms an existing object from T to T', the execution of the type change can never terminate because the number of objects of type T in the database never decreases. In general, for an upgrade $U = \langle n, \{C_i\} \rangle$, the transform function of any C_i should not explicitly create any objects of the pre-types of U, i.e. it should not contain direct calls to the constructors of U's pre-types. Whenever programmers feel the need to do so, they should consider creating objects of the corresponding post-types instead.

However, in some situations, transform functions may have to call methods that invoke the constructors of the pre-types of their upgrades. Therefore, when compiling an upgrade, the compiler has to determine if the upgrade may lead to non-terminating execution and warn programmers about it. Here's the procedure to do so for a given upgrade U:

1. For each type change in U, analyze its transform function to determine if it can create objects of any pre-types of U directly or indirectly.

2. Establish the relation $\mathcal{R}$ such that $\mathcal{R}(P, Q)$ if P and Q are pre-types of U (not necessarily distinct) and P is the pre-type of a type change in U whose transform function can create an object of type Q.

3. Detect any cycle in $\mathcal{R}$. If there is a cycle in $\mathcal{R}$, the execution of U may not terminate. Thus, generate a warning message and ask programmers if they want to continue with the compilation process. Otherwise, proceed directly to the next stage of the process.

Note that in order to determine if a transform function can create objects of the pre-types of its upgrade indirectly through method calls, the compiler will need the definitions of all types (i.e. including the definitions of types not affected by U) in general. However, this need of the compiler can easily be met in an object-oriented database environment because types are stored in the database together with objects. Therefore, the compiler can read the definitions from the database.

Transform functions require special treatment when they are type-checked at compile-time. Normally, an assignment in which the type of the right-value is not a subtype of the type of the left-value should generate an error during compile-time type checking. However, if the assignment is inside a transform function and it is *supported* by a type change in the transform function's upgrade, it is safe to let the assignment pass the checking. For example, consider an upgrade U that contains $C_T = \langle T, T', f^T \rangle$ and $C_S = \langle S, S', f^S \rangle$. Suppose that there is a field of type S in T and the corresponding field has type S' in T'. When f^T assigns the value of the field in its argument to the field in its return object, the assignment will have a left-value of type S' and a right-value of type S. In this case, the assignment can pass the type checking even if S is not a subtype of S'. The assignment is said to be *supported* by C_S. (Because of this, f^T and C_T are also said to be *supported* by C_S.) The assignment will not lead to any run-time type errors because any object of type S represented by the right-value will be transformed by f^S to type S' before it can be accessed through the left-value due to the atomicity of upgrade execution.

2.5 Classification of Type Changes

2.5.1 Effect on Object Representations

Type changes can be classified according to their effect on object representations. A *rep-preserving type change* preserves the object representation of its pre-type—the fields in the post-type are syntactically and semantically the same as those in the pre-type. A *rep-modifying type change* modifies the object representation of its pre-type—the fields in the post-type are different from those in the pre-type.

Among the example type changes in Section 2.3, the first two are rep-modifying. The last two are rep-preserving.

2.5.2 Effect on Object Behavior

Type changes can also be classified according to their effect on object behavior. A *compatible type change* preserves or extends the behavior of objects of its pre-type while an *incompatible type change* modifies the objects' behavior. In a compatible type change, the post-type is a subtype of the pre-type. In contrast, an incompatible type change does not have this property.

Among the examples in Section 2.3, the first two are compatible while the last two are incompatible. In the first type change, the object representation is changed but the methods are preserved. Therefore, the post-type is a subtype of the pre-type. In the second type change, the object representation and the set of methods are extended. However, since the methods of the pre-type are preserved, the post-type is still a subtype of the pre-type. In the third type change, since the `copy` method is removed, the post-type is not a subtype of the pre-type. Finally, in the fourth type change, the `output` method is changed. Although the method is still called `output`, it has a different specification and a different signature. Therefore, the post-type is also not a subtype of the pre-type.

2.6 Completeness of Upgrades

A type change affects an object-oriented database in two ways. First, it makes the pre-type obsolete. Second, it breaks the dynamic type correctness invariant if it is incompatible. These are explained below using a type change $C_T = \langle T, T', f^T \rangle$.

First, since T becomes obsolete after the execution of C_T ends, no more objects of T should be created. Therefore, calling a constructor of T after the execution of C_T ends should be regarded as a run-time type error.

Second, if C_T is incompatible (i.e. T' is not a subtype of T), when objects of T are transformed to T', references to these objects may no longer be type correct—the actual type (i.e. T') of these references may no longer be a subtype of their apparent types (i.e. T or supertypes of T). Any such violations of the dynamic type correctness invariant can lead to run-time type errors when the transformed objects are accessed. For example, suppose that T' does not have a method m that is originally present in T. A run-time type error will occur if m is called on a transformed object.

To prevent run-time type errors, types that are affected by a type change have to be changed as well. Thus, one type change may induce other type changes. These type changes may in turn induce other type changes and so on. A related set of type changes or an *upgrade* is formed. The question is: given a type change $C_T = \langle T, T', f^T \rangle$, which types are affected by C_T?

Obviously, any types that invoke the constructors of T are affected. If C_T is compatible, transforming objects of T into T' will not break the dynamic type correctness invariant on references to these objects. Therefore, these are all the types that are affected.

However, if C_T is incompatible, it will affect other types as well:

1. *Types that depend on T are affected.* Types that depend on T contain identifiers with declared type T. Thus, C_T breaks the dynamic type correctness invariant on these identifiers. To restore the invariant, types that depend on T should be changed so that the declared type of these identifiers is changed to T' and method calls on these identifiers are changed to respect the definition of T'.

2. *Subtypes of T* may *be affected.* Objects of the subtypes of T can be used wherever objects of T are needed. Because of C_T, objects of T' are now needed wherever objects of T are needed before. However, objects of the subtypes of T are still expected to be valid substitutes for objects of T'. Therefore, if any

of these subtypes are not subtypes of T', they should be changed so that they are.

3. *Supertypes of* T may *be affected.* Objects of T can be used wherever objects of the supertypes of T are needed. After C_T changes T to T', objects of T' are expected to be valid substitutes for objects of the supertypes of T. Therefore, any supertypes of T that are not supertypes of T' should be changed so that they are.

An upgrade is said to be *complete* if its type changes do not affect any types other than its pre-types and do not require the support of any type changes not included in the upgrade to pass compile-time type checking. To avoid run-time type errors, this thesis requires that all upgrades are complete. In subsequent chapters, only complete upgrades will be considered.

2.7 Chapter Summary

This chapter defines the notions of objects and types and studies the static aspects of type changes and upgrades. It uses an example to illustrate the various uses of type changes. Since transform functions are the most important components in any type changes, this chapter covers them separately. This chapter also classifies type changes according to their effect on object representations and object behavior. Finally, it discusses the completeness of upgrades.

Chapter 3

Execution of Upgrades

This chapter discusses the execution of upgrades. Three different execution models are studied. The first model is the simplest. It provides the semantics in which all affected objects are transformed instantaneously. The second model modifies the semantics of the first model to eliminate its space overhead. Despite this improvement, there remains a common problem with these two execution models: they severely affect database availability. Thus, they are not suitable for large-scale databases and mission-critical databases. The third model—the Lazy Transformation Model—modifies the second model to preserve database availability. This model is an important contribution of this thesis. An implementation design of this model is described in the next chapter.

3.1 Common Execution Semantics

This section presents the semantics common to the three execution models discussed in this chapter. According to Chapter 2, an upgrade consists of a set of related type changes. When an upgrade is executed, all its type changes are executed simultaneously.

A type change transforms objects of its pre-type into its post-type and makes its pre-type obsolete once all affected objects have been transformed. The effect of executing an upgrade is the aggregated effect of executing its type changes.

Upgrades are executed in the order determined by their serial numbers. During the execution of an upgrade, the complete effect of all previous upgrades is observable but the effect of any later upgrades is not.

The execution of an upgrade is atomic with respect to application transactions. This means that no partial effect of the upgrade can be observed by application transactions. If an application transaction accesses an object of a pre-type of the upgrade, it should not be able to access any objects that have been transformed by the upgrade. If an application transaction accesses an object that has been transformed by the upgrade, it should not be able to access any objects of the upgrade's pre-types. The atomicity of upgrade execution ensures that the completeness of upgrades is reflected in objects accessed by application transactions.

Objects affected by an upgrade are transformed in an arbitrary order. Because objects are not naturally ordered, it will be unreasonable to prescribe any particular order. An implementation can pick any order that works well for itself.

3.2 Snapshot Model

The first execution model is called the Snapshot Model. This model provides the *snapshot semantics* in which objects of the pre-types of the executing upgrade are transformed to their corresponding post-types instantaneously, i.e. the execution is based on a single snapshot of the database.

Under the Snapshot Model, each upgrade is executed as an atomic transaction. At the beginning of this *upgrade transaction*, a *temporary store* is created. The temporary store is initially empty and is separated from the database, i.e. there are no references from objects in the database to objects in the temporary store. During the upgrade transaction, transform functions are applied to objects affected by the upgrade. Objects created by the transform functions are added to the temporary store. When all affected objects have been transformed, the objects in the temporary store are copied to the database and the temporary store is deleted.

Objects that are copied to the database from the temporary store fall into two categories. The first category is objects that are the transformed version of objects in the database. These objects have the same identities as their counterparts in the database. When these objects are copied from the temporary store to the database, they replace their counterparts so that any objects that originally referred to their counterparts now refer to them. The second category is auxiliary objects that are created by the transform functions when they are transforming objects in the database. Auxiliary objects do not have counterparts in the database. These objects are simply added to the database when they are copied from the temporary store.

The use of a temporary store is necessary to provide the snapshot semantics because of complex transform functions. Simple transform functions access only the objects they are transforming. Therefore, if an upgrade contains only simple transform functions, no object will be accessed in more than one application of transform functions during the execution of the upgrade. Thus, objects created by the transform functions can be written directly to the database because they will not be accessed by the upgrade transaction again. However, if an upgrade contains a complex transform function and objects created by its transform functions are written directly to the database, the transformed version of an object may be accessed by the complex transform function. If so, the upgrade transaction will observe two different versions of the object and violate the snapshot semantics. Therefore, if an upgrade contains a complex transform function, objects created by its transform functions have to be written to a temporary store to provide the snapshot semantics during its execution.

However, a temporary store alone is not sufficient to guarantee the snapshot semantics in the presence of complex transform functions. One more requirement is needed: no transform functions, be they simple or complex, can modify objects in the database. Otherwise, the upgrade transaction can violate the snapshot semantics by observing two different versions of an object: one before the object is modified and one after the modification when the object is later transformed or is accessed by a complex transform function.

The snapshot semantics are desirable for two reasons. First, the semantics shield complex transform functions from the intermediate states of the execution of their upgrades. When a complex transform function is applied to transform an object, other objects affected by its upgrade it accesses are guaranteed to be in their pre-transformed versions even if transformed versions of the objects exist in the temporary store. Therefore, programmers can safely use the pre-types of the objects when they define the complex transform function.

Second, object transformations are deterministic. They always yield the same results regardless of the order in which objects are transformed. This is because all object transformations are based on the same snapshot of the database. If a transformation is allowed to observe the effect of another transformation, its result may be different from it would be if the order of the two transformations were reversed.

Despite the advantages described above, the Snapshot Model also has two drawbacks. First, the temporary store can be a huge space overhead. It has to be large enough to store all the objects created by transform functions during the execution of an upgrade. For an upgrade that affects a large portion of objects in the database, the size of the temporary store can approach the size of the entire database.

There is a way to reduce the size of the temporary store: only objects that can be accessed by complex transform functions have to have their transformed versions written to the temporary store. The types of these objects can be determined by a static analysis of the complex transform functions and other type definitions. However, the effectiveness of this optimization depends on the upgrade and the database. There are worst cases in which the size of the temporary store cannot be reduced by much.

The second drawback of the Snapshot Model is that it disrupts database availability. An upgrade transaction can be long and can involve a large number of objects. Therefore, it is likely to have conflicts with other concurrent transactions. But it is also very costly to abort. Therefore, the database management system has to give it concurrency control priority. (One simple way to do so is to shut down the database as described in Chapter 1 so that no application transactions will be concurrent with the upgrade transaction.) Favoring the upgrade transaction essentially keeps the objects affected by the upgrade unavailable to other transactions for as long as the time needed to complete the upgrade transaction. Thus, the Snapshot Model is undesirable for large-scale databases and cannot be used on mission-critical databases.

3.3 Regular Transaction Model

The second execution model is called the Regular Transaction Model. It eliminates the space overhead of the Snapshot Model by abandoning the snapshot semantics.

As in the Snapshot Model, each upgrade is executed as an upgrade transaction in the Regular Transaction Model. Within the upgrade transaction, transform functions in the upgrade are applied to affected objects in the database. However, instead of writing to a temporary store, objects created by the transform functions are written to the database directly. This means that the transformed version of an object will replace its original version and will be available for access as soon as the transformation of the object completes.

Abandoning the snapshot semantics introduces two problems. Both of them concern upgrades that contain complex transform functions. First, complex transform functions are no longer shielded from the intermediate states of upgrade execution. Second, object transformations can now be non-deterministic. The rest of this section explains these two problems and proposes solutions.

3.3.1 Intermediate State Exposure

Under the Regular Transaction Model, when a complex transform function is applied to transform an object, other objects affected by its upgrade it accesses may or may not have been transformed. If the complex transform function is defined assuming that an object has been transformed but it has not, there will be a run-time type error if the transform function invokes a method that is supported only by the object's post-type on it. On the other hand, if the complex transform function is defined assuming that an object has not been transformed but it has, there will be a run-time type error if the transform function invokes a method that is supported only by the pre-type of the object on it. Therefore, special care is needed when dealing with complex transform functions.

To avoid run-time type errors, a complex transform function can check the types of the objects it accesses to determine if they have been transformed and access them accordingly. However, this technique is only suitable for objects that are accessed by the complex transform function directly. In order to apply this technique to an object that is accessed indirectly through a method called by the complex transform function, the check has to be included in the definition of the method. If the method belongs to a pre-type of the current upgrade, the type change that affects the object would not have been defined when the method was defined. Therefore, it is impossible to include the check in the definition of the method. If the method belongs to a post-type of the current upgrade, it is possible to include the check in the definition of the method. However, it is not desirable to do so. Besides being called by the complex transform function, the method can be called by application transactions as well. Since upgrades are complete and atomic with respect to application transactions, the check is redundant when the method is called by application transactions. The redundant check affects the performance of application transactions though.

Another technique to avoid run-time type errors is to transform objects before they are accessed by complex transform functions. Therefore, complex transform functions can safely access objects affected by their upgrades according to their post-types. However, this technique may lead to non-terminating recursions. There are two flavors of such recursions. First, when a complex transform function is applied to transform an object, it may call a method on the object directly or through a series of method calls. If an attempt is made to apply the complex transform function on the object again, a recursive loop will be created. Second, when a complex transform function is applied to transform an object and triggers the transformation of another object, the transformation of the second object may be performed by a complex transform function as well. If the second complex transform function accesses the object being transformed by the first complex transform function, it will trigger its transformation. In this way, a recursive loop is created.

This thesis proposes another technique that relies on the notion of *base methods*. Base methods are defined with respect to upgrades. The base methods of an upgrade represent the common behavior of its pre-types and its post-types. Each base method is supported by a pre-type as well as its corresponding post-type. Also, the base methods of an upgrade call only base methods on objects affected by the upgrade. This applies to objects accessed directly by the base methods as well as objects accessed indirectly through other method calls.

The base methods of an upgrade can be identified by an elimination algorithm as follows:

1. Add all methods common to the pre-types and the post-types of the upgrade to the set of candidate base methods $\mathcal{B}$.

2. For each candidate in $\mathcal{B}$, analyze its definition to determine if it can call any methods not in $\mathcal{B}$ on objects affected by the upgrade. If so, remove the candidate from $\mathcal{B}$.

3. If any candidate has been removed, repeat step 2 above. Otherwise, the remaining methods in $\mathcal{B}$ are the base methods of the upgrade.

Note that the analysis in step 2 above requires the definitions of all types. However, since types are stored in the database, this requirement can be met.

To avoid run-time type errors, complex transform functions are restricted to call only base methods on objects affected by their upgrades. This *base method restriction* applies to objects accessed by complex transform functions directly as well as objects that are accessed indirectly through a series of method calls. Then, no matter whether an object accessed by a complex transform function has been transformed or not, there will not be any run-time type errors. Note that a complex transform function can still access the fields of the object it is transforming directly. But when it calls a method on the object, the method has to be a base method to make sure that it invokes only base methods on other objects. Again, determining if a complex transform function conforms to the base method restriction requires the definitions of all types. These are read from the database.

The base method restriction is presented above as a restriction on complex transform functions due to base methods. However, it can also be viewed as a restriction on base methods due to complex transform functions. Sometimes a complex transform function may need to call a non-base method in order to transform objects correctly. In this case, the post-types of its upgrade have to be changed to make the method a base method. Thus, the base method restriction is a mutual restriction between complex transform functions and base methods.

The base method restriction limits the variability of an upgrade—its post-types have to be sufficiently similar to its pre-types so that there are enough base methods for its complex transform functions to use. However, this is a weak limitation. It causes problems only in an upgrade that satisfies the following conditions:

1. The upgrade contains a complex transform function that accesses objects affected by the upgrade.

2. The pre-types and the post-types of the upgrade are so different from each other that the set of base methods is too small for the complex transform function to correctly transform objects.

Such upgrade rarely exists. Even when such an upgrade is needed, methods can be added to its post-types to provide all base methods needed by its complex transform functions. By preventing application transactions from accessing these added methods (e.g. by not including them in the type interface), the post-types remains unchanged to them.

3.3.2 Non-deterministic Transformations

The other problem introduced by abandoning the snapshot semantics is that object transformations can be non-deterministic for upgrades that contain complex transform functions.

Due to complex transform functions, a later transformation can observe the effect of an earlier transformation. Therefore, if the order of the two transformations is reversed, they may produce different results. However, since objects are transformed in arbitrary order, it is not possible to predict which set of results will be produced before the objects are transformed.

Non-deterministic transformations do not make sense. An object should always be transformed in the same way no matter when it is transformed. To prevent non-deterministic transformations, transform functions cannot leave any effect behind, i.e. transform functions cannot modify any objects in the database. The same requirement is used in the Snapshot Model to guarantee the snapshot semantics. However, what constitutes a modification in the Regular Transaction Model needs special interpretation.

In the Regular Transformation Model, the behavior and/or the representation of an object is modified as soon as it has been transformed. However, this kind of modifications will not lead to non-deterministic transformations as long as calling a base method on the pre-transformed version of an object always returns the same result as calling the method on the transformed version of the object.

Note that the notion of equality used above is an unusual one. Calling a base method on a transformed object and the pre-transformed version of the object may return objects with different identities. However, the returned objects can be considered the same if substituting one of them for the other does not affect the results of any transform functions that access them. This notion of equality may not be checkable by any mechanical means. Programmers should be aware of this when defining upgrades. However, it is unlikely that programmers will define upgrades that transform objects in different ways depending on the order in which they are transformed. After all, upgrades are defined to change type definitions. It is not defined to modify any data.

3.4 Lazy Transformation Model

The Regular Transaction Model is an improvement over the Snapshot Model since it eliminates its space overhead. However, database availability is still disrupted. Thus, it is still not suitable for large-scale databases and mission-critical databases. The Lazy Transformation Model solves this last problem.

The Lazy Transformation Model is a major contribution of this thesis. It is based on the technique of transforming objects lazily, i.e. just before they are used. Not only is it space efficient, but it also preserves database availability. Therefore, it can be used on any databases including those that are large-scale and those that are mission-critical. This section will first describe the general scheme of the Lazy Transformation Model. Then, it will analyze the model's space requirement and its impact on database availability. Finally, it will discuss the details of the model.

3.4.1 General Scheme

Under the Lazy Transformation Model, upgrades are not executed as transactions of their own. Instead, objects are transformed in application transactions.

When an application transaction accesses an object, the database management system first checks if the object's type has been changed by any upgrades. If so, it transforms the object accordingly. The transformation occurs before the object is accessed by the application transaction. Therefore, to the application transaction, the object *appears* to have been transformed in an earlier upgrade transaction.

Object transformations within an application transaction are regarded as part of the transaction by the database management system. For concurrency control purposes, a transformation is just like a modification. Therefore, only one application transaction can transform an object at any time and once an object has been transformed, its transformed version is observed in all later transaction.

3.4.2 Space Analysis

Unlike the Regular Transaction Model in which upgrade transactions are given concurrency control priority and always commit, the Lazy Transformation Model does not favor the execution of upgrades. Object transformations are part of application transactions and can be aborted when application transactions abort.

Because object transformations can be aborted, a transformed object cannot replace the original object in the database until the application transaction that is responsible for the transformation commits. Instead, the transformed object has to be stored aside in memory or on disk temporarily. The temporary storage space does not constitute a space overhead if the application transaction intends to modify the object. This is because the space is required for the modification even without the transformation. Only when the intension of the application transaction is a pure read does the temporary storage space constitute an overhead.

However, the space overhead described above is unlike the one in the Snapshot Model. Typically, an application transaction accesses only a tiny portion of objects that have to be transformed. Even when there are a lot of concurrent application transactions, the total space overhead will still be tiny when compared to the total size of objects that have to be transformed in the worst case scenario, i.e. when the database is large and a large portion of objects in the database have to be transformed. Therefore, the Lazy Transformation Model is space efficient.

3.4.3 Database Availability Analysis

In the Snapshot Model and the Regular Transaction Model, the work of executing an upgrade is done by a single upgrade transaction. Therefore, the availability of the objects affected by the upgrade is disrupted during the execution of the upgrade transaction. In the Lazy Transformation Model, the work of executing an upgrade is spread over time and distributed among running applications. Therefore, large-scale disruption to database availability is avoided.

Under the Lazy Transformation Model, the availability of an object affected by an upgrade is still affected when the object is being transformed. When two concurrent application transactions try to read an object not affected by any upgrade, both of them will succeed. However, if the object is affected by an upgrade and has to be transformed before it is read, one of the transactions have to wait or fail because concurrent transformations of the object will be regarded as conflicting modifications, i.e. the object is not available to one of the transactions.

However, the kind of unavailability described above is limited to objects subjected to concurrent access. They represent only a tiny portion of objects affected by an upgrade. Also, the length of the unavailability of an object is limited to the time its transformation takes or to the length of the application transaction that transforms it depending on the concurrency control scheme used. This is insubstantial when compared to the length of an upgrade transaction in the previous two execution models. Therefore, the overall availability of the database is not affected.

3.4.4 Details

The details of the Lazy Transformation Model are discussed here. In the Lazy Transformation Model, more than one upgrade can be under execution at any time. When the execution of an upgrade starts, the upgrade is said to be *installed*. When the execution of an upgrade ends, the upgrade is said to *retire*. An upgrade that is still under execution is said to be *active*.

Upgrades retire in order. This is to make sure that when an earlier upgrade transforms an object to a pre-type of a later upgrade, the later upgrade will still be active to transform the object. An upgrade can retire when all earlier upgrades have retired and all objects of its pre-types have been transformed to its post-types.

At any time, the set of active upgrades may contain a series of type changes, e.g. one from T_1 to T_2 and one from T_2 to T_3. Therefore, when an application transaction wants to access an object that is currently of type T_1, the database management system has to transform the object from T_1 to T_2 and then from T_2 to T_3 before the application transaction can access the object.

In general, the database management system has to transform an object iteratively until there is no pending transformation on the object before it lets an application transaction access the object. It is possible that a long chain of transformations have to be performed on the object before it can be accessed by the application transaction. The transformations will affect the throughput of the application transaction. However, this situation only happens on objects that are accessed rarely and thus accumulate long chains of pending transformations. Therefore, the transactions that are affected in this way are equally rare.

In the Snapshot Model and the Regular Transaction Model, because each upgrade is executed as a single transaction, the atomicity of the upgrade is guaranteed. However, in the Lazy Transformation Model, extra care must be taken to guarantee that every upgrade is atomic with respect to application transactions.

Since all pending transformations are applied to every object before the object is accessed by an application transaction, each application transaction is guaranteed to observe the full effect of all upgrades that are active at the time the transaction starts. However, a problem arises when a new upgrade is installed in the middle of an application transaction. Before the new upgrade is installed, the application transaction may have accessed an object affected by the new upgrade. Therefore, to guarantee that the new upgrade is atomic with respect to the transaction, the application transaction cannot access any objects that are transformed due to the new upgrade. This not only means that objects affected by the new upgrade should not be transformed when they are accessed by the application transaction, but it also means that the application transaction has to be aborted if it tries to access an object that has been transformed according to the new upgrade by another application transaction.

Doing the above requires extra processing by the database management system. An alternative approach to guarantee the atomicity of the new upgrade is to simply abort the application transaction if it has accessed objects of the new upgrade's pre-types before the upgrade is installed. This approach may cause unnecessary aborts but it does not require so much processing by the database management system.

Similar to the Regular Transaction Model, the Lazy Transformation Model has to deal with the problems introduced by abandoning the snapshot semantics when there are complex transform functions. However, the problems are more difficult in this model. They are discussed below.

Intermediate State Exposure

In the Lazy Transformation Model, complex transform functions are exposed to the intermediate states of the execution of their upgrades. This problem is the same as the one in the Regular Transaction Model. Therefore, it is solved by the base method restriction.

However, in the Lazy Transformation Model, complex transform functions are not only exposed to the intermediate states of the execution of their upgrades, they are also exposed to the intermediate states of the execution of earlier upgrades as well as those of later upgrades. This is a violation of the semantics that upgrades be executed in order. The problem introduced is discussed below. Then, a solution will be proposed.

When a complex transform function accesses an object during the transformation of another object, it will access the object according to a certain type, say T. However, the object may need to be transformed, maybe more than one time, before it is of type T or a subtype of T because of earlier upgrades. Also, the object may already have been transformed from T or a subtype of T to another type due to later upgrades. Therefore, the access may cause run-time type errors.

Consider a series of upgrades $U_i = \langle i, \{C_i\} \rangle$ in which each $\{C_i\}$ contains the type

	1	$\xrightarrow{U_1}$	2	$\xrightarrow{U_2}$	3	$\xrightarrow{U_3}$	4	$\xrightarrow{U_4}$	5
R	σ_R								
S					σ_S				
T									σ_T

Figure 3-1: The configuration of three objects under a series of upgrades.

change $\langle R_i, R_{i+1}, f_i^R \rangle$, $\langle S_i, S_{i+1}, f_i^S \rangle$, and $\langle T_i, T_{i+1}, f_i^T \rangle$ where $i = 1, 2, 3, 4$. Suppose that there are three objects: σ_R of type R_1, σ_S of type S_3, and σ_T of type T_5. The situation is depicted in Figure 3-1. Suppose further that an application transaction accesses σ_S and therefore f_3^S is applied to transform σ_S. During the transformation, f_3^S calls a base method of U_3 on σ_T and then calls a base method of U_3 on σ_R. Let's consider these method calls.

Obviously, the first method call will fail if T_5 does not have the called method. Therefore, it would be nice if the notion of base methods can be extended beyond one upgrade. For example, if the base methods of U_4 are required to include the base methods of U_3, the first method call made by f_3^S will succeed because it will be calling a base method on σ_T. However, extending base methods beyond one upgrade has an unacceptable consequence.

As explained in Section 3.3.1, the base method restriction limits the variability of upgrades. It requires the post-types of an upgrade to be sufficiently similar to the pre-types so that enough base methods are provided for the complex transform functions of the upgrade to correctly transform objects. This limitation is acceptable when it only affects one upgrade as is the case in the Regular Transaction Model. A major change to a type can still be made through a series of upgrades. However, if the limitation is carried across multiple upgrades, it can outlaw major type changes. Therefore, extending base methods beyond one upgrade is not acceptable.

Assume that the first method call succeeds. When f_3^S makes the second method call, the database management system can transform σ_R from R_1 to R_2 and then from R_2 to R_3 first before the method is executed. Since f_3^S calls a base method of U_3, σ_R will have the method when it is of type R_3. Thus, the call will succeed. However, a problem arises when f_1^R is applied to σ_R to transform it from R_1 to R_2. If f_1^R calls a base method of U_1 on σ_S or σ_T, the situation is similar to the one described above. Therefore, a run-time type error can still occur.

The run-time type errors described above are caused by a complex transform function of an upgrade trying to access an object that has been transformed by a later upgrade. To prevent the run-time type errors, no such objects should exist. Therefore, this thesis proposes a *complex transform function protection rule*:

> *New upgrades affecting objects that can be accessed by the complex transform functions of* active *upgrades cannot be installed.*

This rule delays the installation of a new upgrade until it no longer affects objects that can be accessed by the complex transform functions of any active upgrades. Thus, it is impossible for a complex transform function of an upgrade to access an object that has been transformed by a later upgrade and cause a run-time type error as a result.

Because of the complex transform function protection rule, whenever a complex transform function of an upgrade tries to access an object, the database management system can safely apply all transformations due to *earlier* upgrades to the object before it is accessed by the complex transform function. Doing so restores the semantics that upgrades be executed in order. Combined with the base method restriction, it guarantees that the access will be free of run-time type errors. Note that only transformations due to earlier upgrades are applied to the object accessed by the complex transform function. If the upgrade of the complex transform function also affects the object, the object should not be transformed according to it yet. This transformation should be delayed until the complex transform function terminates or else a recursive loop may form as discussed in Section 3.3.1.

To enforce the complex transform function protection rule, it is necessary to analyze the definitions of complex transform functions to determine the types of objects they can access. Again, the analysis requires access to the definitions of all types. This is not a problem for an object-oriented database because types are stored in the database.

The complex transform function protection rule should not affect many upgrades. After all, upgrades are rare. Upgrades that satisfy the condition so as to require a delay in their installations are even rarer. By the time such an upgrade is defined, earlier upgrades that would be affected by the new upgrade are likely to have retired. However, to further ensure that no installations of upgrades are delayed for too long, a mechanism is needed to accelerate the execution of active upgrades. The mechanism should trigger the transformation of objects that are rarely accessed by application transactions. Such a mechanism is presented in the next chapter.

Non-deterministic Transformations

The other problem introduced by abandoning the snapshot semantics when there are complex transform functions is that object transformations become non-deterministic due to the arbitrary order in which objects are transformed. However, because the semantics that upgrades be executed in order are enforced, the non-deterministic problem is no different from the one in the Regular Transaction Model. Therefore, the solution in the Regular Transaction Model applies to this model as well.

3.5 Chapter Summary

This chapter studies three different upgrade execution models. The Snapshot Model is simple but it has a large space overhead and disrupts database availability. The Regular Transaction Model eliminates the space overhead of the Snapshot Model but it still disrupts database availability. Finally, the Lazy Transformation Model solves both the space overhead problem and the database availability disruption problem. However, it is more complicated than the other two models. The next chapter demonstrates how the Lazy Transformation Model can be implemented. It presents an implementation design of the model in a distributed object-oriented database management system.

Chapter 4

Implementation Design

Three different upgrade execution models are discussed in the previous chapter. Among them, the Lazy Transformation Model is the most versatile. It is suitable for regular databases as well as large-scale databases and mission-critical databases. Yet, the Lazy Transformation Model is also the most complicated.

This chapter describes an implementation design of the Lazy Transformation Model on Thor [LAC+96, LCSA99]. It highlights several issues that have to be addressed by any implementations of the model and provides efficient solutions to address them on Thor.

Thor is a distributed object-oriented database management system developed by the Programming Methodology Group at the MIT Laboratory for Computer Science. Thor is designed with scalability and availability in mind. Therefore, it can handle large-scale and mission-critical databases. This chapter will first give an overview of Thor. Then, it will explain how the Lazy Transformation Model can be implemented on Thor.

4.1 Overview of Thor

Thor [LAC+96, LCSA99] is based on a client/server architecture. Thor servers are called object repositories (ORs). Thor clients are called front-ends (FEs).

ORs provide persistent storage for Thor objects. A Thor database consists of one or more ORs connected by a LAN/WAN. A large-scale database can span thousands of ORs and provide storage for billions of objects.

FEs act as interfaces to Thor databases. They shield applications from the dis-

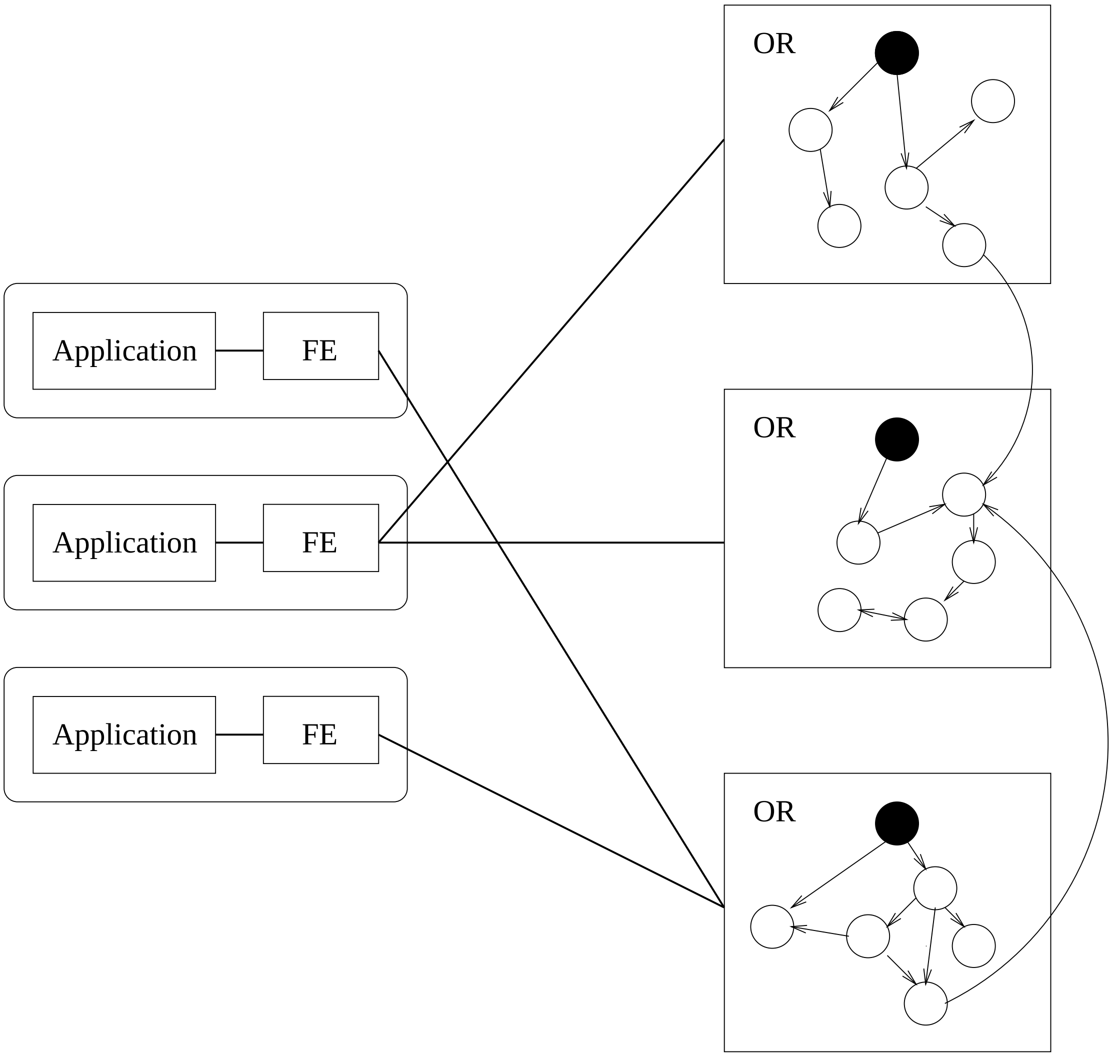

Figure 4-1: The Architecture of Thor.

tributed nature of Thor. Each running application uses a dedicated FE to manage its database access. When an application needs to access an object, its FE will open a connection with the OR where the object is stored. Therefore, to the application, the database appears to reside locally. Other FE functions will be described later in this section.

Figure 4-1 depicts the architecture of a typical Thor database. The small circles represent Thor objects while the arrows between them represent object references. Objects that reside in an OR can refer to objects in the same OR as well objects in other ORs.

Each OR contains a special root object. This is represented by a filled circle in Figure 4-1. Persistence in a Thor database is determined by the reachability from the root objects of its ORs. Objects that cannot be reached from any root objects or from the data structures of any running applications are garbage-collected [ML94, ML97a, ML97c, ML97b].

Thor uses optimistic concurrency control [AGLM95]. Each FE maintains a cache of objects using a hybrid adaptive caching scheme [CALM97]. Operations on objects carried out by an application are executed on the object copies in the cache of its FE. These operations are grouped into atomic transactions for concurrency control purposes. At the end of each transaction, the FE will send a *commit-request message* to each of the ORs that stores objects accessed in the transaction. The message summarizes the transaction and contains the following information:

1. the identities of the objects that were read,
2. the identities and the new values of the objects that were modified, and
3. the identities and the values of the objects that have been made persistent by the transaction.

The ORs then validate the transaction using a two-phase commit protocol [Ady94]. The transaction can pass the validation only if it does not conflict with other concurrent transactions. In this case, it is committed and its effect is made permanent. However, if the transaction fails the validation, it is aborted and its effect will be discarded.

Once an FE successfully commits a transaction that modified objects, the objects' ORs will inform other FEs that may have cached the objects about the modifications by sending them *invalidation messages*. These messages help to keep FE caches up-to-date. They are piggybacked on other messages and are flushed at some interval. When an FE receives an invalidation message, it will remove any stale objects from its cache and abort the running transaction if it has used any stale objects. When these objects are accessed the next time, the FE will fetch their up-to-date versions from their ORs.

In Thor, each object has a type that defines its behavior and its representation. Types are defined in Theta [LCD^{+}94], which is a strongly typed, strictly encapsulated, sequential, object-oriented programming language designed especially for Thor.

According to the original design of Thor, each type is represented by a *type object* stored in the database. This type object contains the dispatch vector of the type and other data that describe the type. Each object of the type contains a reference to this type object. At run-time, type objects are fetched and linked by FEs on demand. Therefore, it is easy to add a new type to a database. However, in the current implementation of Thor, types are simply linked statically with applications together with FE codes. Adding a new type to a database requires re-compilation of all applications. Nevertheless, in the rest of this chapter, the original Thor design will be assumed.

Each object in Thor has a fixed size. For a type that has a constant-sized representation, all objects of the type have the same size. This size is recorded in the type object. For a type that has a variable-sized representation, objects of the type can have different sizes. The sizes of these objects are fixed at the time they are created and are stored in their fields.

In an OR, objects are stored in pages. The header of each page contains an offset table. Each occupied table entry contains the offset of an object in the page. Therefore, within an OR, each object is uniquely identified by the identity of its page and the index of its entry in the page's offset table. This combination is known as the *oref* of the object. Combining the oref of an object with the identity of its OR produces its *xref*. This xref uniquely identifies the object within the database. In the current implementation of Thor, orefs are 32 bits long and xrefs are 64 bits long. However, these can be changed to fit different machine architectures and database requirements.

Thor has excellent execution performance [LAC+96, LCSA99]. One of the reasons is that it has a compact object format. Thor assumes that most object references are local, i.e. they refer to objects in the same ORs as the objects that contain the references. Therefore, space is allocated in the fields of regular objects to store local references in the form of orefs only. If an object needs to refer to an object in another OR, it makes a local reference to a *surrogate*. A surrogate is a small object that contains an xref to a remote object. Surrogates are the smallest possible objects in Thor. Any other objects are either of the same size as surrogates or are bigger.

Xrefs and orefs are used as object references within objects stored at ORs. However, when objects are fetched into an FE cache and operations are carried out on them, object references in the form of xrefs and orefs are inconvenient and slow the operations down. Therefore, object references in FE caches are translated into memory pointers. The translation process from xrefs and orefs into memory pointers is called *swizzling* [Mos90]. The translation process in the reverse direction is called

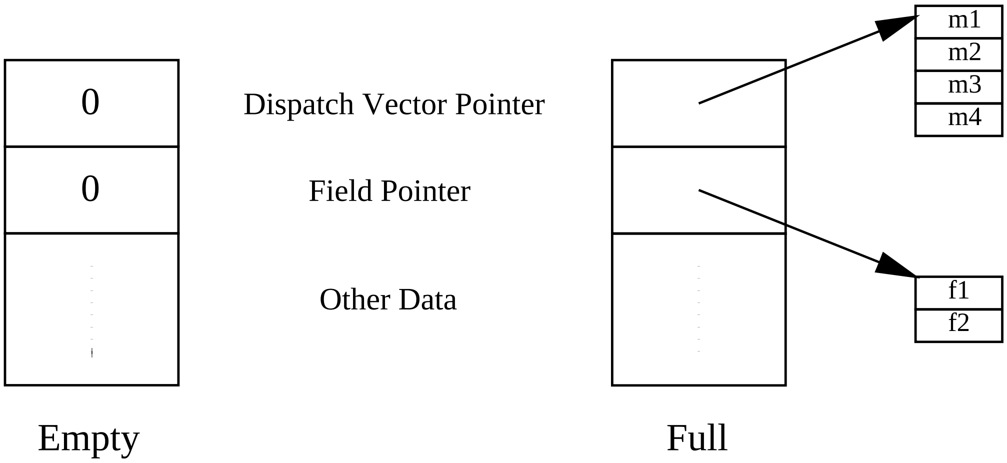

Figure 4-2: An empty ROT entry vs a full ROT entry.

unswizzling. Swizzling of an object reference is done lazily the first time the reference is loaded into a register. A reference is unswizzled when the object that contains the reference is shipped back to its OR in a commit-request message. To facilitate swizzling and unswizzling, each FE maintains a *swizzle table* to record the mapping between xrefs and memory pointers.

Thor uses indirect swizzling. Each FE maintains a *resident object table* (ROT). Each ROT entry represents an object. When an object reference is swizzled, it is translated into a pointer to the ROT entry representing the object being referenced. If the object has not yet had an entry in the ROT, a new entry is created for it. Therefore, every object that can be accessed has an ROT entry.

The ROT entry of an object provides space to hold a pointer to the dispatch vector of the object's type, a pointer to the object's fields, and other data specific to the object. As illustrated in Figure 4-2, a ROT entry can be in one of two states: in an *empty* entry, the dispatch vector pointer and the field pointer are null; in a *full* entry, the pointers point to the dispatch vector and the fields of the represented object. Newly created ROT entries are all empty. Empty ROT entries have to be *filled* to become full entries before the objects they represent can be accessed. Empty ROT entries are filled lazily, i.e. just before the objects they represent are accessed. When objects represented by full ROT entries are discarded from the FE cache, their ROT entries will be *emptied*. When these objects are subsequently accessed, they will be fetched from their ORs again if necessary when their empty ROT entries are filled.

Besides the swizzle table and the ROT, each FE also maintains a *surrogate table*.

Unlike regular objects, surrogates are not represented by ROT entries. Instead, they are stored in the surrogate table. The surrogate table enables the FE to reuse existing surrogates when it unswizzles remote references. This reduces the number of surrogates at an OR that all refer to the same remote object.

When an FE makes an object persistent, it is responsible for finding unoccupied storage space for the object in an OR page. This process also determines the xref of the newly persistent object. For concurrency control purposes, only one FE can allocate space in an OR page at a time. Each OR page has an associated *allocation right*. Only the FE with the allocation right to a page can allocate space on that page. ORs guarantee that the allocation right of each page is only granted to a single FE at any time.

The above description of Thor is sufficient for understanding the rest of this chapter. However, many details have been omitted. Interested readers are encouraged to refer to [LAC+96] and [LCSA99] for a more complete description of Thor.

In the next two sections of this chapter, the implementation design of the Lazy Transformation Model on Thor will be presented. The first section will describe the design for databases that reside on a single OR. The second section will extend the design to support databases that span multiple ORs.

4.2 Single OR Implementation

This section describes the implementation design of the Lazy Transformation Model on single OR databases.

Upgrades are stored persistently in the OR in an *upgrade table*. This ensures that once installed, they will survive failures and their execution can be resumed after a recovery. Each upgrade is represented by an *upgrade object* in the database. The upgrade object contains references to the *type change objects* that represent the upgrade's type changes. Each type change object contains references to the type object of its pre-type, the type object of its post-type, and its transform function. The upgrade table is indexed by the serial numbers of its upgrades. Each entry of the table refers to the upgrade object of its upgrade.

Upgrades are not executed at the OR however. This is because the OR is the bottleneck of the system. Instead, the OR pushes active upgrades to FEs. FEs transform objects in their caches when they are accessed by applications.

The rest of this section is organized as follows:

1. It describes a number of FE data structures related to the execution of upgrades.
2. It discusses the activation of new upgrades.
3. It covers the details of object transformations.
4. It explains how transactions that include object transformations are committed.
5. It presents the retirement of upgrades.
6. It ends with an optimization for rep-preserving type changes.

4.2.1 FE Data Structures

Each FE maintains a *type change table* to store the type changes of the upgrades it has received from the OR. Objects affected by the type changes in the type change table will be transformed by the FE before they are accessed. The type change table is indexed by the pre-types of its type changes. Each of its entries represents a type change and records its post-type, its transform function, and the serial number of its upgrade. For space and bandwidth efficiency, transform functions are fetched from the OR and entered into the table lazily the first time they are used.

The type change table is created during the start-up process of its FE. During the process, the OR generates a manifest of all active upgrades from its upgrade table and sends it to the FE. The manifest contains the serial number, the pre-types, and their corresponding post-types of each active upgrade. The FE then uses this manifest to initialize its type change table.

The *resident object table* of an FE plays a crucial role in upgrade execution. According to the Lazy Transformation Model, an object affected by an upgrade is transformed when it is accessed, i.e. when a method is called on it. However, the performance of the FE will be badly affected if it has to check against its type change table on every method call to determine if the call is being made on an object that has to be transformed. Therefore, the FE remembers which objects have been checked and thus do not have to be checked again by maintaining the following *ROT-invariant*:

> *All full ROT entries represent objects that are not affected by type changes in the type change table.*

With the ROT-invariant in place, no checking and transformation is necessary when a method is called on an object represented by a full ROT entry. When a method is called on an object represented by an empty ROT entry, the FE has to fill the entry before executing the method call. This is when the FE checks the object against the type change table and transforms the object when needed. Because empty ROT entries are filled lazily, this transformation mechanism agrees with the Lazy Transformation Model.

Each FE also creates an *upgrade stack* and a dummy *transient entry set* during its start-up process. A new transient entry set will be created whenever a transform function is applied. The upgrade stack stores upgrade serial numbers. Transient entry sets store pairs each of which consists of a reference to a ROT entry and a reference to a transform function.

To avoid recursive transformation loops, the Lazy Transformation Model requires that whenever a complex transform function accesses an object that is affected by its own upgrade, the object is not transformed and is accessed as is. However, before the object can be accessed, its empty ROT entry has to be filled first. This breaks the ROT-invariant. Therefore, the object has to be transformed as soon as the complex transform function terminates to restore the invariant. The upgrade stack is used to determine if an object accessed by a complex transform function should be transformed before the access or not. The most recent transient entry set is used to record the ROT entries of objects that have to be transformed after the current transform function terminates and the transform functions to be used on the objects. The use of these data structures will be explained later when the details of object transformation are discussed.

4.2.2 Upgrade Activation

When a new upgrade is installed at the OR, the OR activates it by sending a manifest of the new upgrade to each of the running FEs. The manifest contains the upgrade's serial number, its pre-types, and its post-types. Once an upgrade is activated at the OR, the OR will include it in the manifest of active upgrades it sends to every newly started FE.

When a running FE receives the manifest of a new upgrade, it activates it by adding its type changes to its type change table. Doing so breaks the ROT-invariant. Therefore, the FE restores the invariant as soon as it has finished adding entries to its type change table by scanning the ROT and emptying any full entries that represent objects affected by the new upgrade. Thus, when these objects are next accessed, their entries will be filled first. As a result, the FE will check its type change table and transform the objects according to the new upgrade if they have not yet been transformed by other FEs.

Scanning the ROT takes time. On a DIGITAL Personal Workstation with a 433MHz Alpha 21164 processor running OSF1 V4.0, it takes an order of magnitude of 0.1 second to scan a ROT of 786432 entries occupying 12MB of memory space. However, an FE has to carry out the scan only once for every upgrade. Also, its effect on the performance of the FE is not worse than that of a small network congestion when the FE fetches an accessed object from the OR. Thus, the scanning is acceptable.

Before a running FE activates a new upgrade, the running transaction may have accessed objects of the upgrade's pre-types. If, after the activation, it accesses objects transformed according to the new upgrade, the atomicity of the new upgrade will be broken. Therefore, to guarantee the atomicity of the new upgrade, the FE will check if the running transaction has used any objects of the pre-types of the new upgrade before the activation. If so, the FE aborts the running transaction. Otherwise, it resumes the transaction after the ROT scan.

One may worry that emptying the ROT entries of objects affected by the new upgrade may affect the running transaction if it is resumed by the FE after the ROT scan. However, this is not possible because the transaction will be resumed only if it has not accessed any objects affected by the new upgrade. Therefore, when restoring the ROT invariant, the FE will not modify any entries that are relevant to the running transaction.

As discussed in the previous chapter, there is an alternative way to guarantee the atomicity of the new upgrade when the running transaction has accessed objects affected by the upgrade. The FE can delay adding the type changes of the new upgrade to the type change table until the running transaction terminates. Thus, objects in the FE cache will not be transformed according to the new upgrade during the execution of the running transaction. However, the FE still has to abort the running transaction if it accesses objects that are transformed according to the new upgrade by other FEs. This is achieved as follows: whenever the running transaction accesses an object the first time after it has been fetched from the OR (i.e. when its empty ROT is filled), the FE checks if the type of the object is any of the new upgrade's post-types. If so, the object may have been transformed according the new upgrade by another FE and thus the running transaction is aborted.

This alternative method has the advantage of avoiding unnecessary transaction aborts. However, it also complicates the logics of FEs. The method requires FEs to have two different modes of operation: a regular mode and a mode in which there is one or more upgrades waiting to be activated. An FE has to determine which mode it is in every time an empty ROT entry is filled and acts accordingly. This slows down every transaction. Since upgrades are rare and avoiding some rare transaction aborts does not justify slowing down every transaction, this thesis proposes the simpler method discussed earlier to guarantee the atomicity of new upgrades.

4.2.3 Object Transformations

Because of the ROT-invariant, objects affected by type changes in the FE type change table are represented by empty ROT entries. When these objects are accessed, their ROT entries have to be filled first. This is when they are transformed.

The process of filling an empty ROT entry can be divided into two stages. During the first stage, the object represented by the entry will be fetched from the OR if it is not in the FE cache yet. Then, the null dispatch vector pointer and the null field pointer in the entry are updated to the correct values. In the second stage, the type of the object represented by the entry is compared against the pre-types in the type change table. If there is a type change for the object, an attempt to transform the object will be made. There may be more than one pending type change for the object. So, after a transformation, the current type of the object is compared against the pre-types in the type change table again and if there is a type change for the object, an attempt will be made to transform the object accordingly. Thus, the second stage has the form of a loop.

An object is not transformed every time an attempt is made to transform it. When a complex transform function accesses an object affected by an upgrade earlier than its own, the object is transformed before it is accessed. However, when a complex transform function accesses an object that is affected by its own upgrade, the object is not transformed and is accessed as is. This is to avoid recursive transformation loops and is explained in Section 3.4.4. (Note that it is impossible for a complex transform function to access an object affected by an upgrade later than its own. This is due to the complex transform function protection rule.)

Whether to transform an object or not is determined by the upgrade stack. When an attempt is made to transform an object according to a type change, the upgrade stack will be consulted. If it is empty or if the serial number of the type change's upgrade is strictly smaller than the serial number on the top of the upgrade stack, the transform function of the type change will be applied to the object. Otherwise, a transformation will not be made. If the object is to be transformed, the serial number of the type change's upgrade will be pushed onto the upgrade stack before the transform function is applied. This is to make sure that the transform function will not trigger the transformation of objects according to its own upgrade in case it is complex. When the transform function terminates, the upgrade stack will be popped to remove the serial number.

No matter whether an object affected by a type change is transformed or not as determined by the upgrade stack, its ROT entry will have been filled in the first stage of the fill-entry process. Thus, not transforming the object breaks the ROT-invariant. Therefore, the object will have to be transformed as soon as the complex transform function that accesses it terminates to restore the invariant. This is achieved by using a transient entry set. Before a transform function is applied, a transient entry set will be created. During the execution of the transform function, this transient entry set will be used to collect references to ROT entries of objects that are affected by type changes but are not transformed accordingly and the transform functions of the type changes that affected these objects. Once the execution terminates, objects of the ROT entries stored in the transient entry set will be transformed one by one using their matching transform functions. This restores the ROT-invariant.

The object transformation controlling mechanism described above is summarized in Figure 4-3. The figure shows the pseudo code of a function `fill_entry(_e)` that fills an empty ROT entry `_e` and a function `transform(_f, _n, _e)` called by `fill_entry(_e)`. The code requires a transient entry set at the top-level. This is why a dummy one is created during the FE start-up process. However, this dummy set will never be used because whenever it is the most recent transient entry set, the upgrade stack will be empty. Also, note that when objects represented by entries in a transient entry set are transformed, the upgrade stack does not have to be tested. This is because these objects are all affected by the upgrade of the complex transform function just executed. Since the complex transform function can be executed (i.e. it has passed the upgrade stack test), these objects can be transformed.

```
let _us = upgrade stack
let _tes = most recent transient entry set

fill_entry(_e)
{
   %requires: _e is a valid and empty ROT entry
   %modifies: _e
   %effects: fill _e and transform its object when needed.
   %         if the object cannot be transformed yet, add
   %         (_e, _f) to _tes where _f denotes the appropriate
   %         transform function for the object
   1. fetch the object represented by _e if necessary
      and update _e with the field pointer and the
      dispatch vector pointer of the object
   2. while there is a type change for the object's current type
      {
         1. let _f be the transform function of the type change
            and _n be its serial number
         2. if _us is not empty and _n >= top of _us
               add (_e, _f) to _tes
            else
               transform(_f, _n, _e)
      }
}

transform(_f, _n, _e)
{
   %requires: _e is a full ROT entry, _f and _n are the transform
   %          function and the serial number of the type change
   %          that affects the object represented by _e
   %modifies: _e
   %effects: transform the object represented by _e using _f
   1. push _n onto _us
   2. let _tes' = new transient entry set
   3. let _oldtes = _tes
   4. let _tes = _tes'
   5. apply _f to the object represented by _e
   6. pop from _us
   7. for each (_e', _f') in _tes
         transform(_f', _n, _e')
   8. delete _tes
   9. let _tes = _oldtes
}
```

Figure 4-3: The pseudo code to fill an empty ROT entry.

The execution of a transform function can be divided into two stages:

In the first stage, the ROT entry of the object being transformed has just been filled. It points to the dispatch vector of the pre-type of the transform function's type change and the fields of the object in the format defined by the pre-type. This is when the transform function retrieves data from the object, creates an object of the type change's post-type on its heap, and stores the retrieved data into the new object.

In the second stage, the ROT entry will be updated with pointers to the dispatch vector of the post-type and the fields of the new object.

4.2.4 Transaction Commit

When a transaction terminates, its FE will send a commit-request message to the OR where the transaction is validated. A summary of the transaction is included in the message. The summary informs the OR about which objects have been read, modified, or made persistent during the transaction so that the OR can determine if the transaction has any conflicts with other concurrent transactions. Also, the new values of the modified objects and the newly persistent objects are included in the summary so that the OR can write them to disk in case the transaction commits.

Although object transformations are regarded as modifications for concurrency control purposes, the FE cannot simply add the identities and the new values of all objects transformed during the transaction to the summary as if these objects have been modified. This is because modifying an object does not change its size but transforming it may.

Since all Thor objects have fixed sizes, when a transaction commits, the OR will simply overwrite any modified objects on disk with their new values. Doing the same will not cause any problem for transformed objects that are smaller than or of the same size as their original versions. For these objects, the FE can indeed add their identities and their new values to the transaction summary as if they have been modified. However, if an object becomes bigger after being transformed, overwriting it with the transformed version may overwrite its neighboring object as well. Therefore, the object has to be treated specially.

First, the FE has to find unoccupied space in an OR page to store the transformed object. The page of the original object will be tried first. If it has enough space and the FE has the right to allocate storage in this page, the FE will allocate space for the transformed object on that page. In this case, the FE will inform the OR that the object has been transformed and has migrated to a new offset within its page in the transaction summary. Thus, the OR can update the object's entry in the page's offset table and write it to the new location if the transaction commits. Since the page and the offset table entry of the object have not changed, the object's identity is preserved.

If the FE cannot allocate space for the transformed object in the page of the original object, it will have to find space in another page. Doing so changes the identity of the object and invalidates previous references to the object. The FE solves this problem by turning the original object into a surrogate that refers to the transformed object. Therefore, previous references to the object will refer to its transformed version through the surrogate and will still be valid. In this case, the FE will inform the OR that the original object has been modified, its new value is that of the surrogate, and the transformed object has been made persistent in the transaction summary. Thus, the OR will overwrite the original object with the surrogate and write the transformed object to the allocated space if the transaction commits.

Surrogates created as described above point to objects in the same OR. (This section only considers single OR databases.) These surrogates are called *local surrogates* and those that represents remote references as described in Section 4.1 are called *remote surrogates*.

Local surrogates, unlike remote surrogates, do not create long-term space overhead. They are shortcut when references through them are swizzled and unswizzled. When all references through a local surrogate are shortcut, the local surrogate will be garbage collected. The shortcutting happens as follows:

Suppose that an object σ_1 refers to an object σ_2 through a local surrogate. When the reference to the local surrogate in σ_1 is swizzled at an FE, the local surrogate will be skipped and the reference will be translated into a memory pointer to the ROT entry of σ_2. Thus, when the memory pointer is unswizzled, it will be translated into a direct reference to σ_2, i.e. the local surrogate is shortcut.

No matter whether the transformed object overwrites the original object, the transformed object migrates within the same page, or the transformed object moves to a new page and the original object is overwritten with a local surrogate, the OR will regard the original object as being modified during the validation process. Therefore, if the transaction commits, the OR will send invalidation messages to all FEs that may have cached the original object. Thus, the original object will be discarded from FE caches and running transactions will be aborted if they have accessed the original object. When an FE needs to access the object later, it will fetch the transformed version from the OR.

4.2.5 Upgrade Retirement

Since upgrades retire in order, the *oldest active upgrade* (OAU) is the candidate for retirement. When all objects of its pre-types have been transformed, its entry can be removed from the OR's upgrade table and the entries of its type changes can be removed from FEs' type change tables.

The garbage collector is responsible for determining if the OAU can retire. The garbage collector has to visit each object in the database during each phase of garbage collection. Therefore, it can easily find out if there are persistent objects of the OAU's pre-types in the database.

Besides determining if the OAU can retire, the garbage collector also helps to trigger the transformation of objects affected by the OAU. This accelerates the execution of the OAU and expedites its retirement. The acceleration minimizes the impact of the complex transform function protection rule discussed in the previous chapter and is achieved as follows:

When the garbage collector comes across an object that should be transformed by the OAU, it passes a reference to the object to a special FE called the *transformer*. Unlike other FEs, the transformer does not serve any application. Instead, it receives object references from the garbage collector and runs a series of transactions that transform the objects according to the type changes in its type change table.

The transformer has an integral parameter n. As soon as the transformer has transformed n objects, it will commit the current transaction and start a new one. The value of n has to be chosen carefully. If n is set too high, conflicts between transformer transactions and application transactions are likely. If n is set too low, the OR will be flooded with a large number of commit requests from the transformer. Also, the appropriate value for n changes with the age of the OAU. When the OAU gets older, objects of its pre-types remaining in the database are the ones that are less likely to be accessed by application transactions. Therefore, n can be set higher without increasing the chance of conflicts between transformer transactions and application transactions. An adaptive mechanism can be used to adjust the value of n on the fly.

The OAU is special with respect to other active upgrades. Because all upgrades earlier than the OAU have retired, objects accessed by the complex transform functions of the OAU must be affected by the OAU itself if they are affected by any upgrade at all. Thus, no objects will ever be transformed during the execution of the complex transform functions. Therefore, the transformer does not need to have an upgrade stack. However, the transformer still uses transient object sets to keep track of the accessed objects so that they can be transformed after the complex transform functions terminate.

4.2.6 Rep-preserving Type Change Optimization

A type change is rep-preserving if the representation of its pre-type and that of its post-type have the same form and the same meaning. So, when transforming objects in the database according to a rep-preserving type change, the only thing that has to be done to the objects is to make them refer to the type object of the post-type instead of the type object of the pre-type. However, there is a better way to carry out the type change. Instead of modifying many affected objects, the type object of the pre-type can be modified to reflect the post-type. In this way, all affected objects in the database are automatically "transformed". The details of this optimization are explained below.

When the OR activates an upgrade that contains a rep-preserving type change, it makes a copy of the type object of the type change's pre-type first. Then, it modifies the original type object to reflect the type change's post-type. Existing type change objects may contain references to the original type object. These references are updated so that they refer to the copy. Finally, the OR includes the type change in the manifest of the upgrade and sends the manifest to all running FEs as usual.

After a running FE has received the manifest of the upgrade from the OR, added its type changes to its type change table, and aborted the running transaction if it has accessed any objects affected by the upgrade, it processes the rep-preserving type change as follows:

First, it fetches the modified type object and the copy of the original type object from the OR. Then, it updates the entries in its type change table to reflect any updates to type change objects made by OR. Finally, instead of emptying the ROT entries of objects affected by the type change, it updates their dispatch vector pointers during the ROT scan. Doing so "transforms" all objects affected by the type change in the FE cache.

4.3 Multiple OR Implementation

This section extends the implementation design of the Lazy Transformation Model presented in the previous section to support multiple OR databases.

When a database spans more than one OR, type objects, type change objects, and upgrade objects are replicated at each of the ORs. Each OR has its own upgrade table to keep track of its upgrades. One of the ORs is chosen to store the master copy of all upgrades. It is where any new upgrades are installed. This OR is called the *master OR* (MOR). After a new upgrade has been installed at the MOR, the MOR will propagate it to other ORs.

If the database consists of only a handful of ORs, the MOR can send a copy of the new upgrade to each of the ORs. However, if the database consists of a large number of ORs, a hierarchical distribution scheme is used to reduce the workload of the MOR and the bandwidth consumption on the network connection out of the MOR. Under the hierarchical distribution scheme, the ORs are organized into a tree structure with the MOR at the root. When a new upgrade is installed at the MOR, it sends the upgrade only to its children. The children in turn send the upgrade to their own children and so on until all the nodes of the OR tree have received the new upgrade.

An OR may receive an upgrade from the database administrator if it is the MOR or from another OR as described above. However, it may also be informed of the upgrade from an FE. This form of upgrade propagation will be discussed later in this section. But no matter how the OR receive the upgrade, it activates it by updating its upgrade table and sending a manifest of the upgrade to each running FE that has connected to it according to the description in the previous section. The OR can also carry out the rep-preserving type change optimization as described.

When an FE starts up, it will connect to a pre-selected OR and receive the manifest of active upgrades from the OR. During the execution of the FE, if a new upgrade arrives at an OR to which it is connected, it will receive the manifest of the new upgrade from the OR as well. Also, when the FE makes a new connection to an OR, it will receive the serial number of the latest upgrade at the OR. The FE can then determine if it has missed any upgrades stored at the OR and request their manifests if so. The above guarantees that an FE will always be aware of all upgrades stored at the ORs that it has connected to.

When an FE receives the manifest of an upgrade from an OR, it processes the upgrade in much the same way as described in the previous section. The difference in this case is that the FE may receive the manifest multiple times each from a different OR. It processes the upgrade the first time it receives it. Also, when allocating space for a transformed object in a different OR page from the one that holds the original object, the FE will prefer a page on the same OR so that the original object can be turned into a local surrogate instead of a remote surrogate. This is because a local surrogate can be shortcut and garbage-collected but a remote surrogate cannot.

When an FE receives the manifest of an upgrade the first time, it guarantees the atomicity of the new upgrade by aborting the running transaction if it has used any objects affected by the upgrade. This mechanism relies on the condition that the FE is informed of the upgrade in a timely manner. If the FE is not informed of the upgrade, it will not have the chance to abort the running transaction. If the transaction has already accessed objects affected by the upgrade, it may then access objects transformed accordingly by another FE and break the atomicity of the upgrade. Of course, an OR will always inform every FE that has connected to it of any new upgrades. But what if the OR itself does not know about a new upgrade because of communication delays or network failures? Consider the following situation:

There are two ORs: OR1 and OR2. OR1 has received a new upgrade U but OR2 has not been informed about U yet. There are two FEs: FE1 and FE2. FE1 is connected to OR1 as well as OR2. It is informed about U from OR1 and has transformed some objects from OR2 according to U. When the transaction at FE1 commits, these transformed objects are stored persistently in OR2 while it is still unaware of U. FE2 is connected to OR2 only. Therefore, it is also unaware of U. It is running a transaction that has accessed objects affected by U. If the transaction accesses any objects transformed by FE1, the atomicity of U will be broken.

To avoid such a situation, the following *propagation invariant* is used:

> *Whenever an OR or an FE receives an object that has been transformed according to an upgrade, it will be aware of the upgrade.*

If the propagation invariant is maintained, it will not be possible for OR2 in the above to store objects transformed according to U without being aware of it. Also, FE2 will be aware of U when it tries to access any transformed objects. Thus, it will be able to abort the running transaction if it has used any objects affected by U so as to guarantee U's atomicity.

Trivially, the propagation invariant is maintained at all time when no upgrades have been installed. Also, installing a new upgrade at the MOR, sending an upgrade from one OR to another OR, and sending the manifest of an upgrade from an OR to an FE will not break the invariant.

Sending objects from an OR to an FE will not break the propagation invariant assuming that the invariant holds at the OR. This is because the FEs that have connected to the OR will always be aware of the upgrades at the OR.

However, sending objects from an FE to an OR can break the propagation invariant. This happens when the FE sends the OR an object transformed according to a new upgrade that has not arrived at the OR yet. Objects are only sent from an FE to an OR in commit-request messages. Therefore, to avoid the above situation, an FE will include the serial number of its latest upgrade in every commit-request message. When an OR receives a commit-request message from an FE, it can determine from the serial number if it has missed any upgrades at the FE. If so, it can request the upgrade from the FE, the MOR, or other ORs. Therefore, if the propagation invariant holds at the FE, the commit-request message will not contain any objects transformed according to upgrades that the OR is not aware of.

Because objects are not sent between ORs and between FEs, by having the above mechanism in place, the propagation invariant holds at all time by simple induction.

4.4 Chapter Summary

This chapter demonstrates how the Lazy Transformation Model can be implemented. It presents an implementation design of the model on Thor [LAC+96, LCSA99], a distributed object-oriented database management system. The design highlights several issues that have to be addressed by any implementations of the model:

1. An implementation has to remember which objects are up-to-date so that the system does not have to test them to determine if they have to be transformed every time they are accessed.

2. It has to deal with objects changing their sizes across transformations.

3. It has to trigger the transformation of objects that are rarely accessed by applications.

4. It has tolerate communicate delays and network failures when the database is distributed.

This chapter has presented efficient techniques to address all these issues on Thor.

Chapter 5

Related Work

This chapter discusses previous work related to this thesis.

5.1 Orion

Banerjee et al. [BKKK87] were among the pioneers who studied type changes in object-oriented databases. They defined and classified a set of type change primitives (e.g. adding a field to a type, dropping a field from a type, adding a method to a type, removing a method from a type, etc.) and presented an implementation of the primitives on an object-oriented database management system called Orion.

However, unlike the implementation design presented in Chapter 4, Orion lacked the ability to transform objects according to the type change primitives. Instead, various tricks were used to make them appear to have been transformed. For example, if a field was dropped from a type, the corresponding field of an object of the type would be masked out before the object was accessed. If a field was added to a type, a field with a default value would be added to an object of the type in memory accordingly after the object had been fetched from disk. Any changes made to the added field would be discarded when the object was written back to disk. Orion's inability to transform objects made it impossible to meaningfully carry out any primitives that affect object representations.

5.2 GemStone

More complete support for type changes was implemented in GemStone [MSOP86, BOS91] by Penney and Stein [PS87]. GemStone supported a set of type change primitives similar to those in Orion. However, unlike Orion, GemStone was able to transform objects according to the primitives. To carry out a type change, all affected objects were first exclusive-locked, meaning that no transactions that accessed (read or modified) the objects could commit. Then, the objects were transformed and the locks were released. Finally, a report was generated. The report contained a list of objects that had been transformed and a list of objects that had been deleted during the execution of the type change. Obviously, this approach affected database availability.

Despite the above shortcoming, the work of Penney and Stein raised an interesting issue that was not covered by Thor or by this thesis. This was the issue of access control. In GemStone, objects were stored in segments and each segment had an owner. No one except the owner of a segment could access objects stored in the segment unless permission to do so was granted by the owner. Therefore, an interesting situation occurred when a user carried out a type change that affected objects in other users' segments. In this case, GemStone would still transform every affected object in every segment. However, to guarantee that no user was able to infer the existence of objects and their values that the user had no right to access, all owners of the segments that contained the affected objects would receive a report at the end of the type change execution. Each report would only contain information about the objects owned by its receiver.

5.3 OTGen

A system called Object Transform Generator (OTGen) [LH90] was developed by Lerner and Habermann. As its name suggested, OTGen was built to automate the generation of transform functions to transform objects according to user defined type changes. By having the concept of transform functions, OTGen was able to support more general type changes than Orion and GemStone did.

OTGen generated a default transform function by comparing the definitions of the pre-type and the post-type of a type change. Programmers could modify the default transform function to implement exactly what they wanted to do.

OTGen supported atomic execution of related sets of type changes as well as some degree of laziness in object transformations. Unlike the kind of lazy object transformations described in this thesis, OTGen required that each collection of connected objects be transformed together when a member of the collection was accessed. Thus, type change execution could still significantly affect the availability of a database if objects in the database were highly connected.

5.4 O_2

Lazy object transformations at object-level granularity were implemented on O_2 by Ferrandina et al. [FMZ94, FMZ+95]. Similar to OTGen, O_2 automatically generated default transform functions to transform objects according to type changes. Programmers could define their own transform functions to augment the default transform functions generated by the system. When an object was to be transformed, the default transform function would be applied first. If a programmer-defined transform function was available, it would be applied next.

As mentioned in Section 2.4, Ferrandina et al. classified transform functions into two kinds:

- A *simple transform function* accesses only the object it is transforming.
- A *complex transform function* needs to access objects other than the one it is transforming in order to carry out the transformation.

They pointed out that complex transform functions required special attention when objects were transformed lazily—the objects they accessed during their execution (1) could be affected by earlier type changes or (2) could have been transformed according to later type changes. This problem is exactly the intermediate state exposure problem in the Lazy Transformation Model described in Section 3.4.4.

To handle the first case, O_2 used a stack to determine if an object accessed by a complex transform function should be transformed according to defined type changes. Only transformations due to type changes earlier than that of the complex transform function would be applied to the object before it was accessed. The same technique is adopted in this thesis.

To handle the second case, O_2 used an approach called *screening*. Each object was divided into two parts: a regular part and a screened part. Both parts were visible to transform functions but only the regular part was visible to applications. When data were deleted from an object while it was being transformed, they were not physically discarded. Instead, they were simply moved from the object's regular part to its screened part. Therefore, when a complex transform function from an earlier type change accessed the object, the data would still be available for access.

Unfortunately, the screening approach could lead to incorrect object transformations. Because the screened part of an object was not visible to applications, when an application modified the regular part of an object and broke the consistency between its two parts, the application would not be able to modify the screened part to restore the consistency. If the inconsistent data in the object were subsequently accessed by a complex transform function, the complex transform function would produce incorrect result. Therefore, this thesis adopts another approach based on the complex transform function protection rule described in Section 3.4.4 to handle this case.

Chapter 6

Conclusion

This chapter concludes this thesis with a summary of its contributions and a discussion of future research topics.

6.1 Summary

This thesis contributes a general execution model—the Lazy Transformation Model, which allows programmers to correctly modify types that govern the behavior of persistent objects and transform the objects accordingly without affecting the availability of the underlying object-oriented database. The model makes it feasible to maintain applications that rely on large-scale databases and mission-critical databases, which are becoming more and more common.

Changing a type can affect other types. Therefore, type changes are defined and carried out in groups called upgrades. Each upgrade has to be complete, meaning that its type changes cannot affect types that are not also changed by the upgrade. To avoid run-time type errors, each upgrade is executed atomically with respect to application transactions. An ordering is imposed on upgrades so that new upgrades can be defined based on the results of earlier upgrades. When executing an upgrade, all objects affected by its type changes are transformed in arbitrary order. This means that while transforming objects, data stored in the database cannot be modified or else the results will be non-deterministic. Also, when transforming an object, if another object that is affected by a type change in the executing upgrade has to be accessed, only the behavior that is preserved by the type change can be utilized or else a run-time type error may be produced. Once the execution of an upgrade has terminated, no more objects of the types made obsolete by the upgrade can be created.

The Lazy Transformation Model, as its name suggests, is based on the technique of transforming objects lazily. When an application tries to access an object affected by a type change, the object will be transformed before it is accessed. Therefore, the workload of an upgrade is spread over time and distributed among the applications that share the underlying database. This efficient distribution of workload is the key to avoid disruption to database availability. To guarantee the atomicity of upgrades, application transactions are sometimes aborted. However, such aborts are rare as they can happen only when new upgrades are installed.

A model is meaningless if it cannot be implemented. Therefore, this thesis also presents an implementation design of the Lazy Transformation Model on Thor [LAC+96, LCSA99], a distributed object-oriented database management system. The design highlights several issues that have to be addressed by any implementations of the model. These issues include: obsolete object detection, object size variations due to transformations, the transformation of rarely accessed objects, and communication delays and failures. Efficient techniques are included in the design to deal with all these issues.

6.2 Future Work

This thesis is concerned with preserving type consistency between persistent objects across upgrades. However, it has not addressed type consistency between applications and persistent objects. If a type is changed incompatibly, applications that access objects of the type directly may cause run-time type errors. For example, if a method is removed from a type, run-time type errors will be produced when applications call the method on objects of the type. In Thor, such errors are trapped by the database run-time system and the concerned applications are halted. The applications will have to be modified and re-compiled before they can be used again. Such disruption of services provided by applications can cause inconvenience and frustration to users. Therefore, it will be desirable to have a mechanism that can update applications affected by an upgrade on the fly while the upgrade is being executed.

The notion of base methods is proposed in Section 3.3.1 to avoid run-time type errors during the execution of an upgrade. A formal definition of the notion and an in-depth study of its properties will be an interesting area of research.

According to the implementation design of the Lazy Transformation Model described in Chapter 4, every FE receives the manifest of every new upgrade and activates the upgrade no matter whether the FE needs to access objects affected by the upgrade or not. To improve the efficiency of the design, a scheme to filter upgrades so that FEs only receive or activate upgrades that matter to themselves will be needed.

Also, it will be worthwhile to actually implement the design and build a compiler for upgrades. Assisting programmers in constructing complete upgrades, carrying out the base method restriction, and enforcing the complex transform function protection rule are all challenging tasks.

Finally, the issue of access control mentioned in Section 5.2 will be an interesting topic for future work as well. The owner of an object should be able to specify if the object should be affected by a type change defined by another user of the database or not. The challenges are:

1. How to express and enforce such a specification?
2. How to guarantee run-time type correctness when type changes no longer necessarily affect all objects of their pre-types?

Bibliography

[Ady94] Atul Adya. Transaction management for mobile objects using optimistic concurrency control. Technical Report 626, Massachusetts Institute of Technology, January 1994.

[AGLM95] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 23–34, San Jose, CA, May 1995.

[BKKK87] J. Banerjee, W. Kim, H. Kim, and H. Korth. Semantics and implementation of schema evolution in objected-oriented databases. In *Proc. of ACM SIGMOD Annual Conference on Management of Data*, pages 311–322, San Francisco, CA, May 1987.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone object database management system. *Communications of ACM*, 34(10):64–77, October 1991.

[CALM97] M. Castro, A. Adya, B. Liskov, and A. C. Myers. HAC: Hybrid adaptive caching for distributed storage systems. In *Proc. of 16th ACM Symposium on Operating Systems Principles*, pages 102–115, St. Malo, France, October 1997.

[FMZ94] F. Ferrandina, T. Meyer, and R. Zicari. Implementing lazy database updates for an object database system. In *Proc. of 20th VLDB Conference*, Santiago, Chile, 1994.

[FMZ$^+$95] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and database evolution in the O_2 object database system. In *Proc. of 21th VLDB Conference*, Zurich, Switzerland, 1995.

[LAC+96] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 318–329, Montreal, Canada, June 1996.

[LCD+94] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A. C. Myers. *Theta Reference Manual.* Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, February 1994. Also at http://www.pmg.lcs.mit.edu/papers/thetaref/.

[LCSA99] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *Proc. ECOOP '99*, Lisbon, Portugal, June 1999.

[LH90] B. Lerner and A. Habermann. Beyond schema evolution to database reorganization. In *Proc. OOPSLA/ECOOP '90*, pages 67–76, Ottawa, Canada, October 1990.

[LW94] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[ML94] U. Maheshwari and B. Liskov. Fault-tolerant distributed garbage collection in a client-server object-oriented database. In *Third International Conference on Parallel and Distributed Information Systems*, Austin, TX, September 1994.

[ML97a] U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by controlled migration. *Distributed Computing*, 10(2):79–86, 1997.

[ML97b] U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage using back tracing. In *Proc. of the ACM Symposium on Principles of Distributed Computing*, Santa Barbara, CA, August 1997.

[ML97c] U. Maheshwari and B. Liskov. Partitioned collection of a large object store. In *Proc. of SIGMOD International Conference on Management of Data*, pages 313–323, Tucson, AZ, May 1997. ACM Press.

[Mos90] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Trans. on Information Systems*, 8(2):103–139, April 1990.

[MSOP86] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented DBMS. In *Proc. OOPSLA '86*, pages 472–482, Portland, OR, September 1986.

[PS87] D. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. In *Proc. OOPSLA '87*, pages 111–117, Orlando, FL, October 1987.