1467

A SCHEME FOR DYNAMIC OBJECT MIGRATION IN STATICALLY- TYPED OBJECT-ORIENTED DATABASES

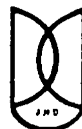
Dissertation submitted to the Jawaharlal Nehru University
in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY
in
COMPUTER SCIENCE

by

SIBA CHARAN BESHRA

67p.



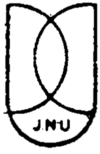
SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110 067
INDIA

JANUARY 1996

....Dedicated

to

Daai & Teng-ang

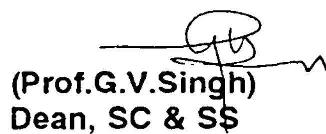


जवाहरलाल नेहरू विश्वविद्यालय
JAWAHARLAL NEHRU UNIVERSITY
School of Computer & Systems Sciences
New Delhi-110067

CERTIFICATE

This is to certify that the dissertation entitled **"A SCHEME FOR DYNAMIC OBJECT MIGRATION IN STATICALLY-TYPED OBJECT-ORIENTED DATABASES"** being submitted by **SIBA CHARAN BESHRA** to School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi, in partial fulfilment of the requirements for the award of the degree of Master of Technology in Computer Science, is a record of the original work done by him under the guidance and supervision of **Dr. R. C. Phoha**, Associate Professor, School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi, during monsoon semester, 1995.

This work has not been submitted in part or in full, to any other university or Institution for the award of any degree or diploma.



(Prof.G.V.Singh)
Dean, SC & SS

(Dr.R.C.Phoha)
Supervisor

Gram JAYENU Tel: 657676, 667557 (Extn. 3441 Telex: 031-73167 JNU IN ★ FAX: 91-11-6865886

Acknowledgements

I wish to take this opportunity to express my deep sense of gratitude to Dr. R.C. Phoha, Associate Professor, for his supervision, constant help, suggestion and encouragement during this project work. I am also thankful to him to expose me into a new field.

I am thankful to Prof.G.V. Singh, Dean, SC&SS for extending necessary computing facilities.

I am also thankful to Shri Manoj Kumar Sarangi for his timely suggestion which helped me a lot for the improvement of this project.

I wish to thank Pratap, Manoj, Chitrasen, Amulya, Sushanta, Dillip, Meghnath, Bajinath and Gopal for their help.

It would be a sacrilege to record any formal gratitude to my mother, sisters, brothers, and Rima for their perennial encouragement and moral support without which it world have been impossible to accomplish this study.

Finally, I gratefully acknowledge the JRF provided by the University Grant Commission.

Siba Charan Beshra

CONTENTS

Certificate

Acknowledgement

Chapter I **Introduction and Organisation of Chapters** 1-5

Chapter II **An Overview of Object-oriented Database Environment** 6-39

2.1 Introduction

2.2 Database acess and query relevance

2.3 A new look at the database utility

2.4 The architecture of an object base environment

2.5 Towards a common definition

Chapter III **The Problem** 40-49

3.1 Introduction

3.2 Statically typed object oriented languages

3.3 An object migration model

Chapter IV **The Scheme** 50-63

4.1 Migration Operators

4.2 Object Migration Control

4.3 Specification Design and Implemetation

Chapter V **Conclusion** 64

Bibliography 65

Chapter I

INTRODUCTION

Since half a decade or so, object-oriented database systems have become an extremely hot topic of database research and development. At many places all over the world, people work on individual aspects or complete system prototypes. Already, some systems have even reached the market place. However, the notion of "object orientation" in the context of database systems is unfortunately still widely unclear and means different things to different people. When the database community got attracted little latter and started to carry over object-orientation to databases/database management systems.

Object-oriented database management is a new technology that aims to provide data management support to applications (such as computer-aided design, computer-aided software engineering, geographic information processing, and office automation) that currently are not well served by conventional, record-oriented database management systems (DBMSs).

Conventional DBMSs provide only a small, fixed set of data types (e.g., integers, real, strings) and conceptual data structures (e.g., records, relations) that were designed for business data processing applications. These data types and structures have been found to be inadequate for representing the richly structured, complex objects (e.g., maps, documents, programs, part assemblies) encountered in the new applications. Furthermore, conventional DBMSs provide only a fixed repertoire of operations for manipulating records or relations, and a fixed set of access methods and processing strategies for efficiently implementing these operations. These require the application programmer to coerce the objects and operations of his application domain into those supported by the DBMS; such mappings are clumsy and inefficient at

best, and often impossible because conventional database query language are computationally incomplete.

Conventional "modern" programming languages are computationally complete and have rich type systems. However, they are inconvenient, inefficient, and inadequate for applications that require access to shared persistent data, i.e. data whose lifetime is longer than a single program execution. They provide only files for the storage and retrieval of persistent objects; before a persistent object can be manipulated, it must be retrieved from the file and translated into an "internal" format, afterwards, it must be translated back into the "external" format suitable for secondary storage, and then stored in the file. Sharing is at the level of mutual exclusion over files provided by the operating systems.

The conventional approach to supporting application development has been to embed a database query language in a programming language (e.g., SQL embedded in PL/1 or C). The complex structures and complex operations of the application are implemented over transient data in the host programming language; calls are made from the program to the DBMS to store persistent data records in the database, are to retrieve persistent data into the program's data structures. This approach does not eliminate the translation problem : since arbitrary objects cannot be made to persist, there still are two different type systems to deal with, each with its own set of operations. The application programmer has to learn two different languages. Also, the translations are likely to be inefficient, since each complex operation may require several database operations, which cannot be optimized in toto by the DBMS.

Object-oriented database systems lie at the confluence of research in DBMSs and programming languages. Their goal is to simplify application development by alleviating, and ultimately eliminating, the translation problem.

The first stage beyond relational DBMSs was the development of DBMSs based on semantic data models. These DBMSs retained the powerful concepts of transactions and set-at-a-time queries, but introduced enhanced data modelling features in support of "structural object orientation". Entities were introduced to model real world objects each entity was uniquely identified by a system-generated identifier, rather than by a user-supplied key; entities existed even if their attributes had not been assigned any values. Attributes were allowed to be non-scalar; this permitted the direct modelling of entity-to-entity relationships, and as a special case, the structure of complex objects. In these semantic models, entities were typed and the entity types participated in type hierarchies, down which attributes were inherited.

Progress towards object-oriented DBMSs has required extending semantic DBMSs with concepts to support "behavioural object orientation". These include user-defined type-specific operations or methods (which are implemented by procedures or rules), computed and derived attributes, single and multiple inheritance of attributes and operations down type hierarchies, and encapsulation of structure and behavior with the object types. Encapsulation means that, while the implement its operations, the user of the type sees only its interface, i.e., the name of the type and the signature of its operations.

In recent years, a great deal of research and development in database management has taken place to satisfy the need for database support for complex application areas such as computer- aided design, computes-aided software engineering, office information systems, pictorial and graphics databases, etc. The result of this research is the emergence of a new generation of database management systems. These new system fall roughly into three categories; extensible database systems, DBMS, object- oriented database systems and database system generators.

The term extensible database system to refer to a DBMS which allows an application programmer or database administrator to add new data types

and new operations on these new data types to an existing DBMS. In an extensible database systems, the underlying DBMS will always be the same in such models as concurrency control, recovery, basic storage, and query language. Extensible databases may also allow the addition of access methods and hooks to allow the query optimizer to make use of these new access methods. Note that the new operations added usually pertain to operations on what would be the domains in the relational model, and that the major operators of the query language would not be modifiable. Assumption is that, there is already a database system so that persistence of data and data structures from one instantiation of a program to another is already taken care of. Persistence only becomes an issue if such a system is implemented by augmenting a traditional programming language.

An object-oriented database systems is an extensible database systems which incorporates a semantic data model, which in turn is sufficiently powerful to allow reasonably straight forward modelling of complex objects. Complex objects are objects which have a highly nested structure, like a large piece of software or an engineering design. They may also be very large. The semantic data model should be able to model such things as arbitrary levels of aggregation, components of aggregates which may be sets of other objects, and IS-A or generalization hierarchies with inheritance of object components and of operations. Some object-oriented database systems also model revisions and knowledge. One should be able to define new operations on these objects, which could have the effect of changing the query interface. It is also reasonable to expect that the other pieces of the database management system work at the object level, for example that the object are passed to the storage manager as a unit, that locking and recovery are done on a per-object basis etc. Since this definition inherits all the properties of an extensible database system, object-oriented database systems also allow the definition of new data types, operations on them and access methods. They will have the same underlying concurrency control methods, recovery etc., from one instantiation to another, although, being object-oriented, these modules may differ quite a lot from those found in traditionally DBMS's. Some example of object-oriented

database system are GemStome, Probe and Iris.

The third category of new systems is the database system generators, customizers are compilers. These system allow a database system implementor or architect to design and implement a new type of database management system without having to write all the code for his new system from scratch. Using one of these database generators, one could generate different database management systems which differ in virtually all of their modules. The resulting could be traditional (non-extensible) DBMS, an extensible one or an object-oriented one. Examples of database generators are EXODUS, the Data Model Compiler and GENESIS.

Organization of Chapters

The rest of the report is organized as follows :

Chapter II is an overview of object-oriented database environment.

Chapter III is the introduction to the problem and the object migrations are discussed. The general goals for supporting object migration are also discussed and a model for describing kinds of object migration is specified.

Chapter IV presents a set of operations for facilitating object migration. The issues of controlling such object migration, by a "migration control specification" is addressed and discussed. Some theoretical issues of object migration, which include inference rules for migration control specification are also focused. The issue of conducting more meaningful and desirable migrations are also stated. Design and implementation details are given.

Finally conclusions are given.

***

Chapter II

AN OVERVIEW OF OBJECT-ORIENTED DATABASE ENVIRONMENT

2.1 Introduction

We need large databases, their main goals is to provide facility in storing, retrieving, and manipulating information elements. The advantage of using an object oriented approach is information elements and operations an database contents can be linked together. The object-oriented design is useful because of its dynamic aspects, given the complexity of working through more traditional approaches.

An object databases are not searched in sequential fashion but in parallel, with particular emphasis on query relevance and flexible interlinking. If a flat file structure prevails, for instance, there will be separate processor for each tuple whose task is to store the information in that row and to wait for a query. If a query requires some part or all of the information that is stored on a processor, the latter will respond; otherwise it will do nothing.

In an object-oriented databases organization on the other hand, a more elegant solution is possible. In terms of search time, it no longer matters where in a database an object is located. A system optimizer sees to it that databases, or sections thereof, that are accessed frequently are searched first. This reduces the response time and increases flexibility for the enduser.

2.2 Database Access and Query Relevance:

The database model must consider object behavior requirements when

classes have already been defined and composed. Such models, will increasingly be tailored for reusability of object class specifications, the basic mechanism providing a description of the behavior of objects of a given class in a specific application. Thus

- Computational requirements can specify implementation of each object in a given object-oriented language.

- Operation and properties associated with an object can be portioned according to the different roles played by that object.

- For each role status graphs can be associated with incoming and outgoing messages, for the instantiated role and at each state.

An object database can be built with schema construction tools independent of an object programming language, and with query language that can deliver information elements to languages such as C++.

Database management systems (DBMS) that address the more classical extentional aspect of accessing and manipulating information elements. Contrary to the general acceptance of relational database management systems, there is no consensus on the proper model for an object-oriented DBMS. One of the reasons is that this fact of DBMS is only now emerging; another is that object-oriented approaches are seen as a set of concepts drawn from several areas in information science, which researchers adopt to their idiosyncrasies and that of their problem domain.

2.3 A New Look at the Database Utility

The mission of an object-oriented database utility should be that of combining the features of objects with traditional database capabilities. The goal is the structuring of referential and concurrent sharing of objects through

the support of meta concepts, data abstraction and inheritance.

Among the functionalities to be added to the database utility are integrity, versioning and control of a distributed environment of objects. The aim is to create true repository of information elements shared by multiple platforms, by removing the semantic gap between an application domain and its representation. To reach this goal we must overcome some important constraints.

1. The heterogeneity in the design of information elements to be encapsulated within objects.

2. The heterogeneity of platforms running within the same applications environment.

3. Internal cultural issue in systems usage which magnify and perpetuate these heterogeneity issues.

4. Requirements for increasingly rapid transaction and message turn around.

5. The specificity of hardware and software characteristics, which add further to heterogeneity.

Rules are the actual triggers. Instance can go in and out of a frame, a frame being like a query. Declarative facilities are added, as rules are procedural, not declarative. The methodology includes the notion of setting up a query by reformulation.

The links and relationships among entities in the complex real world are represented and manipulated directly.

This approach achieves its modeling capability through object- based concepts while trying to alleviate the mismatch between heterogeneous programming languages and incompatible data structures. Most important, the method just described could not be without object-oriented concepts.

2.3.1 Application domain for object databases

A conceptual framework is of fundamental importance. It is most necessary in connection with the design of object-oriented databases and their access through a query language or retrieval paradigm.

Object oriented approaches support the necessary expressive power for performing complex computations associated with an applications. The need for associating specially adapted programming products to the object database metaphor has been present for some time. The problem that have been encountered with traditional approaches have meant that today the main markets for object-oriented solutions are:

- Computer-Aided Design (CAD)
- Computer-Aided Manufacturing (CAM)
- Computer-Integrated Manufacturing (CIM)
- Cross-Functional Projects (Such as task management)
- Cross-Departmental Projects (for instance, cost control)
- Cartographical Implementations
- Computer-Assisted Software Engineering (CASE)
- Office Automation (OA)

Not only the characteristics of these applications different from traditional accounting type routines, but for some of them, such as office automation, there exist rather rich libraries of programming products that are well accepted by the user communities. What particularly characterizes these applications, however, is that they present stringent requirements in terms of

database manipulations.

There is another fundamental reason why CAD, CAM, CIM, risk management, cost control, CASE and office automation applications tend toward object-oriented, multimedia databases. They all require fast-growing amount of computer storage, and the connections between the information elements in storage is much more complex than in classical data processing. In many cases, database size and complexity make it nearly impossible to cluster related multimedia information elements (or even data) in a effective manner through pre-established paradigms, as has been done for nearly 40 years. A considerable amount of navigation is necessary to access and update such databases-object orientation presents a demonstrable advantage.

2.3.2 Object-Oriented Paradigms and Long Transactions

An object program is a construct of interacting objects that encapsulate information element (IE) and the algorithms that specify the behavior of the IE. Operation on object can take place only through a well-defined interface to the objects behavior, the actual implementation of such behavior being hidden from every user but the designer and the database administrator.

The routine that act on an object are the primary means through which the IE, encapsulated in that object, may be manipulated or modified.

An object invoke a given object's resource by sending a message, the target object interprets that messages, and the appropriate action is performed.

This the simplest form of an object-orientation paradigm. The evoked action (or method) uses known components: information elements and procedures, as well as encapsulation and interfacing.

The decomposition consist of a high-level analysis will be accomplished on an object-oriented approach not only in terms of objects, but also in terms of the services they provide. Each services will help define the properties and behavior of a set of common objects, or more precisely instances of classes created as a program running.

This identifies another limitation of functional decomposition, which becomes obvious when account for the fact that each object has the same number and types of fields, and differs from other objects only in the class of values stored in those fields. While class templates provide a basis for modularity, leverage is really gained by the ability to define one class of objects in terms of other classes-which requires new departures.

Methodologies currently being developed for object-oriented databases, and associated programming approaches are:

1. Identifying existing library classes,
2. Extending these by using inheritance, and
3. Developing new classes as requirements evolve.

Object descending from the same parent class are essentially compatible. Each understand the same message as the others, yet each performs the task in its own way.

This modularity allows the transparent creation and insertion of new class instances into the program, but to be implemented in an able manner, it also calls for an object-oriented methodology, starting at the design level. This path is feasible, but there are problems.

A rational methodology starts at the database level, focusing on object-based paradigms. It will elaborate on classes and their characteristics, identity objects and attributes, convert operations affecting objects, establish

visibility, and work out relational interfaces. These are necessary conditions for implementing a fundamental object solution, always keeping in mind the characteristic prerequisites: meta, inheritance, equilibration, data abstraction and so on.

Some of the preconditions in an object-oriented environment stand at a system level. Others are more strictly related to design and implementation details. But together such preconditions help provide;

- Direct support of object concepts
- Development of sharable objects
- Persistent constructs
- Dynamic schema evolution
- Queries on class lattices
- Version management capabilities.

2.3.3 Database-wide Object Identification

The identification and classification of objects within the database are necessary. It establish interactions between objects in terms of services required and services rendered. This is written in any-to-any sense, that is,

-cross-database
-Any object to any object
-Any location to any location
-Any time to any time

In a dynamic environment we never know in advance which object will be needed at any given time. Hence there is a need for steady accessibility, valid notation, and any-to-any connectivity.

Object notation pertains to design level as well as to the classes of runtime objects prior to and during implementation. Interrelationships between design objects and classes can and should be represented through different stages of the life cycle.

In a distributed database landscape, unique and unambiguous identification is a demanding task using traditional methods. The organizational prerequisite is object classification. Once the classification is done properly, the identification is fairly simple - provided we have clear concepts and the appropriate supports.

The handling of classes requires an iterative analysis of whether a new organization will be useful. This creates the needs for inheritance diagrams, but helps assure that future projects can reuse the structure without having to design class and object and object relationships.

Prototyping helps in providing constructive feedback, assuring that object identification requirments receive the appropriate documenattion and classifiaction. Such feedback is made possible by object-oriented techniques, which help provide a more reliable and robust database structure.

The identification can start with the persistent objects that can be handled early in the development cycle. These may be presented to programmers as object classes to be handled at a rather early stage, with a degree of confidence that even if system specifications change, such classes stand a good chance of remaining invariant.

The database is the root of the consistent object space. Every object reachable from this database root must itself be consistent. Instantiation variables should contain the names of all objects that serve as the root of such consistent object space. Only objects thus specified can be shared and accessed through different transactions (are queries) with the users assured access

rights in the database under authentication and authorization conditions.

These quantities are upheld through an object-type orientation, and have been found to be quite helpful in a query and/or transaction environment. Transactions are typically atomic, and their program is either executed entirely or not at all: If the user (software, terminal, enduser) performs updates to the persistent database related to a given transaction, then either an the updates must be visible to the outside world or none must be seen.

This consistency aspect and all related functionality must be supported by database languages and their primitives. But consistency *per se* is a meta concept.

Committing a transaction in an object environment is achieve by sending a commit message to the system, which however, must be interpreted in a homogeneous manner.

2.3.4 Making Sense of Database Heterogeneity

The object-orientation is not the way to solve all problems. This is particularly true when incompatibilities in database solution approach and programming tools perpetuate if not accentuate damaging discrepancies. It is wise to take the proverbial longer, hard look at object database solutions before making definite commitments, studying these in detail and establishing the research for heterogeneity, benchmarking them with transactions, messages, and ad-hoc queries, and making a factual and documented choice of tools and concepts, then sticking to it.

Under no conditions should there be one type of object-oriented solution for a transaction environment, a second for messages a third for queries. If this happens, most of the benefits of the implementation of object-based databases will be lost.

Account should also be taken of the need for concurrency control within a given implementation perspective. In the typical execution environment transactions run concurrently under a given DBMS, with multiple transactions being active at the same time. These transactions can access and update the same consistent objects, with a DBMS guaranteeing the handling of a given database and of the transaction results.

But in a distributed database environment if transactions are allowed to run concurrently without any conflict resolution, there can be anomalies in both the database consistency and the transactions' execution. A similar statement can be made of queries and messages. This leads to the need to:

- Specify object-oriented systems requirements
- Identify the objects and the services they must provide
- Establish their interactions in terms of interfaces
- Always observe inheritance relationships as well as the aggregation of classes.

Since transactions are automatic, the design to be adopted must guarantee that partial results or updates that fail are not propagated through the object database. There are also several types of failure from which a system must recover.

The important concept associated with object databases is the notion of nested transactions. In a nested model, each transaction consist of atomic subtransactions, which can well nested. For a transaction to commit, each of its subtransactions must also commit.

The updates of the transaction tree becomes visible only after the higher-level transaction commits. The updates of the committed transactions at the intermediate level (the leaves) become visible only within the scope of their immediate predecessors.

A system-wide "commit transaction" may ask the underlying system to commit its transaction (s), while the user might choose to continue the session after the commit. This will start a new transaction and create a new workspace. Alternatively, the user might decide to undo all the updates to the database and abort the transaction through a message such as "System abort transaction"

But the solution is not always so simple. A commit indicates success. For example, if a parent in a transaction tree has three children that are doing subtransactions, and if two of them commit and one child is running on a node that crashes, then parent might find out about it and abort its own transaction, but two of the children are committed. They cannot be rolled back, so there must be a cleanup to correct erroneous data. A simple approach is to use only phase 1 commits. But this does not solve for complex transactions.

The best way to look at this issue is to appreciate that solutions within an object-oriented transactional environment are a composite of the best parts of applications we did in the past and of new concepts substantiated through an object orientation. The latter can enhance the sophistication of database design, and help in assuring reasonable homogeneity provided we know how to use to advantage the object-oriented approaches that are available today.

2.4 The Architecture of an Object Base Environment

A system architecture present us with the need for fundamental choices. These determine what we will do and what not, but art of architecturing large artifacts is only now being developed. There are no priori criteria for choices; only basic system study can define fundamental requirements and how to characterize the products we want.

An architecture refers to a set of design principles that define relationships, interactions among elements, and ways and means for integration. In object-oriented implementation, such design principles characterize various part of network objects.

The difference between the object-base and object -oriented approaches lies in the fact that in a object base system, object encapsulate a collection of services accessible by a message passing interface. Typically object have an identity and a mutable state. By the contrast, the same definition suggest that, an object-oriented environment, an object can be instantiated from classes with subclasses defined by inheritance. As we have seen, classes are also objects, with new classes possibly created at run time.

The object-oriented representation supported by a message-passing model is the dominant scheme. Object have associated messages, which sometimes referred to as methods. By representing design components as object capable of sending and receiving messages, the responsibility for assimilating run time changes becomes distributed. Message can be received by a class of objects or a single object only when the domain defined by them is idle. Message sent to domain that is active or blocked are queued up and wait to be delivered, which typically happens a first-come, first-served basis.

This notion is important since, in object-oriented applications, message are the only form of interobject communication.

The concept behind message-passing approach akin to that of pipes in UNIX implementation. The pipe is an input/output bufffer that accepts one input and gives one output. In such an environment, programs are connected with pipe files that have the ability to pass data from writer to a reader.

Object-oriented computation in the broad sense is computation described as sequence of requests to the objects through a single access method such as

message passing. The class method mechanism handles well the requirements for the type definition and information hiding. The message protocol provides a useful way of controlling the updates that can be performed on a data object. In addition the inheritance mechanism makes database schemas easy to modify,and new variant can be constructed as easily as subclasses.

An object base is defined to be a system which contains a large set of active as well as as passive objects. For active objects, not only the data portion, but also the control and knowledge portions of an object are stored and managed.

In the past, researcher and developer have approached object- oriented database implementation along two direction:extending the relational model or applying the ideas of object-oriented programming to permanent storage. Most of the system in the first category have been designed to simulate semantics data models by including mechanisms such as abstract data types, procedural attributes, inheritance, union type attributes, and shared sub-objects. Most of system in the second category extend an object-oriented programming language with persistent objects and some degree of declarative object retrieval. Both approaches have drawbacks in processing a large number of active objects. The first approach suffer from the unstability problem resulting from separation of control and data. The second approach, on the other-hand losses the advantages provided by fact-oriented database operations.

2.4.1 Representing objects

In the work C++ is chosen to be extended as the object representation language. It is chosen based on the observation that C++ has acquired enough attention and acceptance as the object-oriented language in the computer community. These extension are described in the following subsections.

2.4.1.1 Complex objects

A complex object is an object consisting of a set of (possibly complex) objects in the sense that

(1) domain of an attribute can be any class;
(2) The value of an attribute can be set of objects.

A complex object is an abstraction of its component objects. Consequently a method associated with a complex object imlements a function of its own component object as a whole; so are the attributes of the complex object. Non-complex object are called simple objects. To realize the concept of complex object, it is necessary to explicitly incorporate the notion of 'set' in the object language:

set classes: Given a class α, the class of all possible ordered sets which can be derived from instances of α is declared as :

class set_of_α
{
...
methods
...
}

The following declaration defines a set **a** of class α:

set_of_α a;

Set Projection given a set or an object of a class α, the following notation designates the projection of an attributes $A_1,...A_n$:

a $|A_1,...A_n$

2.4.1.2 Active Object and Models

The construct provide in C++ are sufficient to describe a passive objects, i.e.,objects whose activities are triggered when a method associated with the objects is called. In real applications, a special class of objects need to be defined in order to describe objects which are continuously active according to some control mechanism. Such object are called *active* objects.

Control : An active object can be characterize by a set of state transition rules. In the extended language, a class of active objects is declared as a subclass of the class active, for which any attribute, one defined, can be a state as follows:

state attribute, ..., attribute;

The control portion of an active object is expressed as a set of production rules, which is designated as the attribute control (where domain is set_of_production) of the object. A production is asserted in the following form:

condition => statement;

Where condition is any logical expression over states and inputs, and can include any quantifier over sets:

Universal Quantifier

A variable in a logical expression can be universally quantified by the quantifier:

(forall < variable_id > in < set_id >)

Existential Quantifier

A variable in a logical expression can be existentially quantified by the quantifier:

(exist < variable_id > in < set_id>)

Membership

The following function returns 1 if <variable_id> is an element of <set_id> :

<set_id> : member (< variable_id >);

Similar to complex objects, we can define a complex active object to be a set of (possibly complex) active objects. With this definition, a complex active object can be regarded as a concurrent production system.

Communication

For a complex active object, we classify the styles of communications among its components objects into two categories: synchronous and asynchronous. Communication between two objects is defined to be synchronous if;

1. The calling object suspends its execution after a message is sent to the other objects and
2. THe calling object resumes execution immediately after reply is received from the called object.

Communication between two objects is said to be asynchronous if the calling object continues its execution after a call is made. Asynchronous

TH-6370.

communication is achieved in the extended language via messages. The class message is defined as follows:

```
Class message{
Public:
time time-stamp, reference;
object sender, recipient;
set-of-objects arguments;
void send ();
boolean receive();
};
```

An asynchronous call to method associated with object C with arguments is made by first creating message objects, assigning appropriate values to its attributes, followed by sending the message with the send operation:

δ.send();

Where δ is message just created, on the other hand, any message sent to an object is picked up by the boolean function receive, which is called in the form of;

m.receive();

The junction returns true if a message has been received; in this case m is instantiated to the message received. It returns false otherwise. A message m is regarded as the reply to a previously sent message δ. if:

m. reference = δ.time-stamp
m. sender = δ.recipient
m. recipient = δ.sender

According to the above, an ordinary C++ function call c.a. (x_1.., x_n) implements a synchronous communication sessions; it is equivalent to a send operation followed immediately by a receive operation.

Inputs, Outputs and **links** in the extended language, some attributes of an active objects can be chosen to be the inputs and outputs of the object as follows:

Classifier attribute,..., attributes;

Where classifier can either be the keyword input or the keyword output. The assignments of inputs and outputs allows different objects be connected directly in order to form an interconnected complex object. A linkage between two objects can be established by the operation *link*:

link (a.r,c.s)

The operation connects a.r presumably to be an output of object a, to c.s presumably to be an input object b, so that any assignment to a.r is made to be b.s as well instantaneously.

Class Template In summary, the general form of the class declaration for an active object class is;

class <class.id>{
<class-id> <variable-id>, [...<variable-id>];
...

<class-id> <method-id> (parameter.1: domain-1,...,
parameter-n; domain-n);
...

```
classifier attribute,...., attribute;
...

int clock;
set-of-production control={
<logical-expression> => actions;
...
}
}
```

In the above, classifier can be one of the following key words; state, input, and output.

Associative object Retrieval The availability of sets as described in the above also allows objects be retrieved in an associative fashion. It is assumed that the following functions/statements are used to access the elements in a set:

1. *<set-id> <variable-id>);*
2. *<set-id> ; delete (<variable-id>);*
3. *(foreach <variable-id> <set-id>) statement;*

2.4.1.3 Management of Active Objects

Given a set of active objects, the problem of object management is concerned with the impact created by any change made to the system. It is desirable that adjustments can be made automatically according to any change so that the system is always consistent.

State Space and Criteria

Given an active object P with an states s_1,..., s_n, we define the state space of the object to be Domain (s_1) x Domain (s_2) x...x Domain (s_n). Among

the states, we assume that one is chosen as the final states. Let's define the set of reachable states of P to be the set of all possible states which can be reached, either directly or indirectly, from the initial state. The following criteria are chosen as the constraints when an active object is updated:

Liveness: An active object should have no state which is a dead- end state, where a dead-end state is a state from which no further state transition can occur and it is not a final state.

Consistence: An active object should be consistence in the sense that, in any state, there exist no conflicting actions, where two actions conflict each other if their effects logically violate each other.

Other criteria, such as reachability and deadlock-freeness, can be considered in a similar way.

Adding and Removing a State

If an active object is live and consistent, these two operations are processed as follows:

Adding a state: Assuming s_{n+1} is added to object P and the object becomes P', the state space of P is enlarged to Domain (s_1) x Domain (s_2) x...x Domain (s_n) x Domain (s_{n+1}). Any active V in the original state space now corresponds to Domain (s_{n+1}) state (v,u_1) ... (v, u_r), where r=Domain (s_{n+1}) and the set of u_i's spans all possible values of s_{n+1}. Let us assume that v is not a final state. Since V is not a dead-end state in P, there must exist a rule in P for which v satisfies its left hand side. Clearly, in P' each of (v, u_1)...(v, u_r) still satisfied that LHS of the same rule. Consequently, P' remains to be live. On the other hand, since no new rules (and actions) are added to P' no conflicting actions may be taken in each of the new states. In summary, adding a state variable to a line and consistent object does not damage such properties.

Deleting A State: Deleting a state is more complicated than adding a state. Assuming s_i is removed from object P and the object becomes P', the state space of P is shrunk to Domain (s_1) x...x Domain (s_{i-1}) x Domain (s_{i+1}) x...x Domain (s_n). Domain (s_i) states (v,u_1) ... (v, u_r), where R= Domain (s_i) and the sets of u_i's spans all possible values of s_i, now converge to a single state v. Since non of the original states (v,u_1) ... (v,u_r), where R= Domain (s_i), is a dead-end state, it is clear that v is not a dead-end state. Consequently, P remains to be live after s_i is deleted.

Deleting a state is complicated due to the requirements of consistence. The complication arises from the fact that the left hand side of some rules may include conditions on s_i. Simply dropping such conditions from those rules can create the following problems;

(a) An update rule can violate the intention of the users;

(b) Actions which used to be taken in states (v, u_i) ... (v,u_r) are now collectively taken in the state v; and some of them may be conflicting. Instead of inspecting very state for possibly conflicting actions, the following procedure can be taken: for each pair of conflicting actions a_i and a_j,identify conditions (a_i) and conditions (a_j). In the above, conditions (a_i) designates the set of LHS's of those rules whose actions include a_i; conditions (a_j) can be defined in a similar way. Intersection of conditions (a_i) and conditions (a_j) identifies the states in which conflicting actions a_i and a_j can be taken at the same time. Due to these factors, the user is consulted when a state is deleted and some rules are affected by this change. The conflicting actions are reported in the mean time, assuming all conditions including the deleted state are dropped. Subsequent actions from the user are handled according to the procedures for adding rules and deleting rules.

Adding and Removing a Rule

If an active object is alive and consistent, these two operations are processed as follows:

Adding a Rule: Assume a rule R of the form C_R => A_R is added to object P. That state space clearly remains to be the same. Let state (R) be the set of states in which C_R can be satisfied. It is possible that executing R from a state in states (R) can result in a state from which no rule is applicable: a dead-end state. Such states can be detected by identifying all the states which may be directly reached from the states in states (R) and followed by inspecting each of such states and looking for applicable rules. If a dead-end state can be discovered the addition of R is not safe.

The addition of R may as well create conflicting actions, since the actions associated with R may be in conflict with actions of some rules which are applicable in a state of states(R). The procedure described in the section "Deleting a State" can be applied to delete such states.

Deleting a Rule: Assume a rule R of the form C_R => A_R is removed from object P. The state space clearly remains to be the same. Let states (R) be the set of states in which C_R can be satisfied. It is possible that removing R can result in a state which to be directly reachable from a state in states (R) no longer satisfies the LHS of any remaining rules : a dead-end state. Such states can be detected by identifying all the states which may be directly reached from the states in states (R) and followed by inspecting each of such states and looking for applicable rules. If a dead-end state can be discovered, the removal of R is not safe.

On the other hand, the removal of R from P causes no problem as far as consistence is concerned. This is because each state v of the original state space is consistent, and the removal of R does not create any new action in v.

Adding and Removing an Attribute

If an active object is live and consistent, these two operations are processed as follows:

Adding an Attribute: Adding an attribute of a class in an object causes no problem since the state and the rules of the object remain intact.

Deleting an Attribute: Deleting an attribute from a class of an object has not impact on the liveness of the object. Since some actions of the rules may include the attribute to be deleted, the removal of the attribute may make the action part of such rules incomplete. Simply dropping such updates from such rules may create problem as the updated rules may violate the intended semantics of the object, although the object remains consistent (as no new actions are taken). User involvement is required in this case.

Adding and Removing a Method or a Class

Adding a method is like adding an attrubute ; and deleting a method is like deleting an attribute. Adding a class is equivalent to adding a set of states, attributes, methods, and rules. Deleting a class is equivalent to deleting a set of states, attributes, method and rules.

2.5 Towards a Common Definition

A List of features and characteristics that an ideal OODBs should have and that together constitute a definition of the notion "Object-oriented database system".

Like a relational database systems is one which is based on the relational data model, an object-oriented database system to be database management system with an object-oriented data model.

To fall into the category of database management systems means to have the following features.

1. Persistence
2. Disk management
3. Concurrency control
4. Recovery
5. Ad hoc query facility

Access control and distribution might be added. Above listed are the characteristics that are required and not the mechanisms by which they are provided (e.g. locking, transactions etc. for concurrency control, recovery,...). Disk management include such things like access paths, clustering, or buffering they are not directly visible to the user, but no realistic DBMS can fulfill its task without them. Under a ad hoc query facility, we understand any means that allows access to database data without the necessity to go through the usual cycle of programming, compilation, execution and debugging. For example, a descriptive query language, a graphical browser are some "fill in the form" facility would do that job.

The list of characteristics we require for an object-oriented data model is much more controversial. First of all, the concept of data model itself has to be understood in a broad sense as a framework in which to express real would semantics in a database system. Though this is not at all a novel view, traditional data models offer rather limited means and do not address at all some features for advanced semantic capture. They should thus not be taken as a yardstick for understanding what a data model is. The requirements include the following.

1. Composite objects
2. User-definable types
3. Object identity

4. Encapsulation
5. Types/classes
6. Type/class hierarchies
7. Overloading/overriding/late binding
8. computational completeness

The units an OODBS deals with are called **objects**. They have a representation i.e. a (possibly structured) value or state (sometimes also called a set of instance variable), and a set of operations that are applicable for that object.

Composite objects (or , synonymously, complex objects, structured objects, molecular objects) are built from (among others) components that are objects in their own right and may be composite themselves. The presence of this characteristic especially means that objects of any level of composition have to be supported in a uniform way, that object constructor (e.g., for tuples, sets, lists) have to exists, and that specific operations to dynamically modify and exploit object structures (both in their entirely and specific parts of them) are needed.

Every data model come with a number of predefined data types (which are usually simple ones like integers, characters, strings). Above that, traditional record-oriented data models provide some sort of "parameterized" types with fixed sets of generic operations. In the relational model, for example, there is the parameterized type "(homogeneous) set of (flat) tuples, called a relation. The only parameter to be chosen for a relation by the users are the number of attributes, their name, and there (predefined, simple) types (plus the name of the relation itself). In contrast, the *user-definable types* requirements means that really new types can be introduced to the system by its users, and afterwards dealt with in the same way as with predefined ones. This includes particularly that mechanism are needed to program new operators and register them with the system.

Object Identity: means that the existence of an object is independent of its value. It is thus possible within the database to distinguish between the equality of two objects (i.e. happen to have - at a given point in time - the same value) and their identity (they really are-always-the same object). An object thus has to be considered as a pair (<Oid>, <value>) where <Oid> is an object identifier and <value>, according to the above, may be simple or composite. Obviously, an OID has to be system-wide unique, must not change during the objects lifetime, and even after its deletion, it should be guaranteed forever that any object may not have the same identity. Object identity (even when not explicitly made visible in a system) is an underlying concepts for shared objects (i.e. objects that are components of two or more objects); it is also necessary for easily and correctly reflecting updates of real world entities in the database, otherwise, if e.g. a person gets married and at this occasion changes his/her name which has been used as a key for the respective database objects, how could one tell whether the update object would still represent the same real world entity as before? Obviously, the current practice is to have the user introduced artificial attributes like employee numbers etc. and thus make it his task to deal with identity. It is now recognized that the database system can cope much better and more reliably with this problem.

Whether a user defines a new type, he has to choose, a representation for the values of its instances, he specifies the operators instances, and he programs their bodies (in terms of the representation). In this case, it usually does not make sense that user of the type may look at the representation details or at the operators codes; all they need to know to use objects of the type is its interface, i.e. the operator specifications.

Encapsulation: provides for information hiding in this abstract data type flavor. Note, however, that this should not be made an all or nothing principle; in some cases, one may well want to define a new type just on a representational basis and adopt the (generic) operators of the representation (e.g. direct access to the attributes of a tuple) for the type interface, maybe

augmented by one or two additional operations that are user- defined.

The requirement for types/classes as such is not that new for databases systems, but it has an extended meaning for OODBS and the notion of a class has been carried over from the programming language area. Whatever the favorite definitions type/classes includes the following:

- the specification of the communalities of a set of objects with the same properties (i.e. their operators and representational structures),

- a mechanism to create instances (objects) of the type/class ("object-factory").

- mechanisms to query and manipulate the set of instances currently in existence for a type/class (the extension, "object warehouse").

The real novelty with respect to types/classes in OODBS is that they can be organized in hierarchies and thus allow to express that one type is considered a subtype of another one i.e. that it is specified in more details (with respect to representational structure and/or operations) than the supertype. The standard example is a type person with attributes like name, address, age etc. and subtypes employee with additional attributes for salary, department,... and student (semester, courses, grade points,...). Along with type hierarchies goes the concept of inheritance in that objects of subtype inherit the properties (again, structure and/or operators) from the supertype, in addition to those properties that have been specified with the subtype itself, of course, inheritance propagates all the way to the top of the hierarchy of these are none than just two levels. In summary with type hierarchies and inheritance, more semantics can be expressed than without it introduces an additional modeling discipline (refinement), and it may even save coding effort

because operators need not always be recorded.

Closely connected are the requirements for overriding, overloading, and late binding. Overloading means that the same operator name (and interface) may be used for different operations in different types. This allows to reimplement ("override") an inherited operator for a subtype, taking into account the additional semantics that may be known there. The advantage one gain is that users that deal with objects of various subtype of a given supertype don't have to program tedious case selections to find out the exact type of any object and apply the appropriate operators; they may uniformly use the operator specified for the supertype, and the system will automatically determine which implementation to execute. Obviously this mandate late binding; the system cannot bind names to programs previous its runtime. However, this is again not that much different from traditional approaches (consider e.g. the "Find" operator in a network database system).

Finally, Computational Completeness relates to the language facilities provided for programming the operators of user-defined types. We require that arbitrary algorithms may be coded, and thus more query language facilities will typically not be enough for this purpose.

The first two criteria, composite objects and user-definable types, are the decisive ones, i.e. if a model does not considered enough progress to existing approaches to justify a new name. On the other hand, once composite objects and/or user-definable types are available, the data model is already a big step forward compared to record oriented ones, even if some of the other goodies are missing.

In this respect [10] introduced the following classification: A data model is called

1. **Structurally object-oriented** if it supports composite objects.

2. **Behaviorally object-oriented** if it supports user-defined types.

3. **Fully object-oriented** if it supports both and has all the other features as explained above.

Full object-orientation matches the comprehensive definition as introduced. Of course, the provision of the other six data model features (where applicable; some of them do not make sense for structural object-orientation) does not hurt for the first two classes either. In fact, at least most behaviorally object- oriented database system etc., then, is again a database system with an appropriate data model.

The criteria to be meet as given above - even of they are all adhered to do not define the one and only object-oriented data model (in contrast to e.g. the relational model which defines exactly one such thing). In consequently various way of providing the individual concepts and mechanisms can be (and have been) investigated. The same holds true for the other database system issues, and many of them do have to be reconsidered in the light of the data model that is now different from classical ones. And finally, several additional aspects resulting mainly from the applications areas object-oriented systems are primarily aiming at should be (and already are investigated in this context).

* Within the framework as introduced, concrete ways have to be found to present all the necessary features to fall object-orientation in a coherent and orthological fashion. Quality criteria will composite, among others, expressive power and ease of comprehension and use. To best integrate composite objects with user-definable types (including encapsulation). Furthermore, most present approaches offer just one uniform mechanism to express object composition as well as object referencing (as a means to model all kinds of other-typically rather "lose"- associations

between objects). Unfortunately, this conceals apparent differences to real world facts and is thus counterproductive to the goal of offering highly expressive data modelling facilities.

* Beyond composite objects, applications are often concerned with object versions (i.e. multiple representations of the same semantic entity, to account for different states, different times of validity are creation, alternative or hypothetical information etc.). Appropriate modelling mechanisms and operators should be integrated into an object-oriented database systems.

* How is query processing integrated into object-oriented database systems (both, at the conceptual- how do ad hoc queries interact with e.g. encapsulation?- and at implementation level)?

* How do the notion of database views and consistency transfer to object-oriented data models?

* Protection issues have to be based on the notion of object which is the natural unit of access control in this framework.

* For databases containing large number of data, archiving may become an important issue. Again, objects (and their, versions, if any) form the natural unit for this activity.

* Powerful implementation techniques are needed to provide object-oriented database system that perform efficiently. These may partly be carried over the wealth of approaches developed for relational and other traditional systems, but undoubtedly further concepts will need to be considered, e.g. specialized access paths for composite objects, storage for versions).

- Object-oriented main memory buffering.

Also, how do the ideas of extensible database system architectures relate to the special case of object-oriented database systems?

* How do the concepts of active [9] and deductive database systems tie in with object-oriented database systems?

* High quality database design is rather tedious job even for record-oriented database systems. It appears that it is even more difficult within the framework of object-oriented database systems as the target application areas show a much higher degree of complexity which may easily be exceed the gain in modelling power. Appropriate design methodologies and tools that support them have to be developed.

* In contrast to the development of the relational data model, formal foundations for object-oriented data became interested in this area, there is justified hope that these deficiencies are going to be remedied within the near future. However, care has to be taken that theory addresses full-fledged object-oriented data models and not just the easier issues.

* Finally, what migration paths can be meaningfully, offered for the users of today's relational database systems in order to enable them to benefit from the advantages of object-oriented ones? How can heterogeneous distributed DBMS s help?

Database concepts provide independence of storage and user data formats. The schema describes the form of the databases contents, so that a variety of users can satisfy their data requirements by querying the shared

database using nonprocedural declaration of the form:

SELECT what-I-want WHERE some-set-of-conditions-is-true.

A database administrators integrates the requirements of diverse users. The shared database can be changed as long as the schema is changed to reflect such changes. Concept of database normalization help avoid redundancy of storage and anomalies that are associated with updates of redundantly stored information.

The principle formal database mechanism to obtain selected data or an application is a view specification. A view on a database consist of an query that defines a suitably limited amount of data. A database administrator is expected to use predefined views as a management tool. Having predefined views simplifies the users access and can also restrict the user from information that is to be protected.

View have seen formal treatment in relational databases, although subset definitions without structural transformations are common in other commercial database systems.

A view is defined in a relational model as a query over the base relations, and perhaps also over other view. Current implementation do not materialize views, but transform user operations on views into operations over the base data. The final results is obtained by interpreting the combinations of view definition and user query, using the description of the database stored in the database schema.

The view, like any relational query results, is a relation. However, even when the base database is fully normalized, say to Boyce-Codd normal form, the view relation, is in general only in first normal form. Views are in that sense already closer to objects: related data has been brought together.

Object-oriented programming languages help to manage related data having a complex structure by combining them into objects. An object instance is a collection of data elements and operations that is considered an entity. Object are typed, and the format and operations of an object instance are inherited from the object prototype.

The prototype description for the object type is predefined and the object instance are instantiated then provide a meta description, similar to a schema provided for a database. That description is, however, fully accessible to the programmer.

Internally an object can have an arbitrary structure, and no user-visible join operations are required to bring data element of object instance together.

The object concept covers a range of approaches. One measure is the extent to which messages to external operation interfaces are used to provide access and manipulations functions. Object may be active, as in the Actors paradigm, are passive, as in CLU, or somewhere in between, as in similar or in Smalltalk in terms of initiating actions.

The use of objects permits the programmer to manipulate data at a higher level of abstraction convincing arguments have been made for the use of object-oriented languages and some impressive demonstrations exist. Especially attractive is the display capability of objects. Object concepts can of course be implemented using nonspecialized languages, for instance in Lisp or Prolog.

OBJECT AND DATABASE

Let us assume a database used for persistent storage of shared objects. A database query can obtain the information for an object instance, and an object-oriented programming language can instantiated that object. An

interface is needed, since neither can perform the task independently. A view can define the information required for a collection of objects. Linkage to the object prototype and its operations is performed in the programming system. The program queries the database nonprocedurally to remain unaffected by database changes made to satisfy others users.

The query needed to instantiate an object may seem quite complex to a programmer. A relation is a set of tuples, from an idealized point of view, each tuple provides the data for same object. However, normalization often requires that information concerning one object be distributed over multiple relations, and brought together as needed by join operations. The base relation must contain all the data required to construct the view tuples; the construct views. An ideal composition of an object should allow its data to be managed as a unit, but unless non-first-normal form relations-supporting repeating groups-are supported for views, multiple tuples are still required to represent one object in a relational view.

Hence storage of object is not easy in database, as indicated by the extensions proposed to Ingres for such tasks. A further complexity is that objects themselves may be composed from none primitive objects. In hierarchical databases records may be assembled from lower level tuples, but in internal database the programmer has to provide join specifications externally to assemble more comprehensive relations from basic relations.

***

Chapter III

THE PROBLEM

3.1 Introduction

Current object-oriented approaches to database modeling and design emphasize data abstraction and operation encapsulation. In a conventional object-oriented database (OODB), the conceptual structure (schema) is embodied by a collection of abstract data type (called 'classes') which, when defined, are organized into an inheritance (ISA) hierarchy. Objects are then created within classes, instantiating and justifying these predefined data types. In this way, a class prescribe both structural and behavioral properties (attributes and methods) of its objects. Therefore the object in a class can be stored efficiently through a shared representation, and the set-oriented access of class member is made more efficient; this efficiency becomes increasingly important as classes content larger numbers of objects [2].

Beneath such a classification-based approach is a salient assumption that each object in database has exactly one (specific) class (if we do not count the superclasses of the specific one), namely the one in which the object was created. This has been the case with those OODB models and systems which are based on 'statically-typed' (or strongly-typed) object- oriented programming languages (OOPLs) such as C++ [3], Eiffel [4], Trellis/Owl [5], etc. Such an assumption, however, imposes some serious restriction in modeling real world which may be quite dynamic in nature. As a matter of fact, in the real world which the database are set to model, object may 'migrate' from classes to classes dynamically in many practical situations. Consider a person Rama, for example, who was single but later on get married (hence Rama migrate from single to married), and who was recently promoted from a programmer to a

project manager (hence he migrates from programmer to manager). Object can also be 'transmitted' to more than one classes at the same time. For example, a student Shyam becomes an engineer and at the same time a lecturer (e.g. an adjunct position) after he graduates. Yet as another possibility, object may be 'propagated' to several other classes while still maintaining their membership in current classes. For example, a scholar Sita becomes an American immigrant, while she still maintains her Indian citizenship (hence Sita is propagated to be member of American, in addition to the current Indian membership).

3.2 Statically-typed Object-oriented Languages

Programming languages in which the type of every expression can be determined at compile time are said to be *statically-typed* languages. The requirements that all variable and expressions be bound to a type at compile time is sometimes too restrictive and may be replaced by the weaker requirement that expressions are guaranteed to be type-consistent, if necessary by some run-time type checking. Languages in which all expressions are type-consistent are called statically-typed language [1].

The class of statically-typed object-oriented languages is the subclass of object-oriented languages in which all object have abstract interface and languages and the language is statically typed (see **Fig. 1**). Statically-typed object-oriented languages are extreme in their choice of structure over flexibility. Languages like C++ are statically-typed.

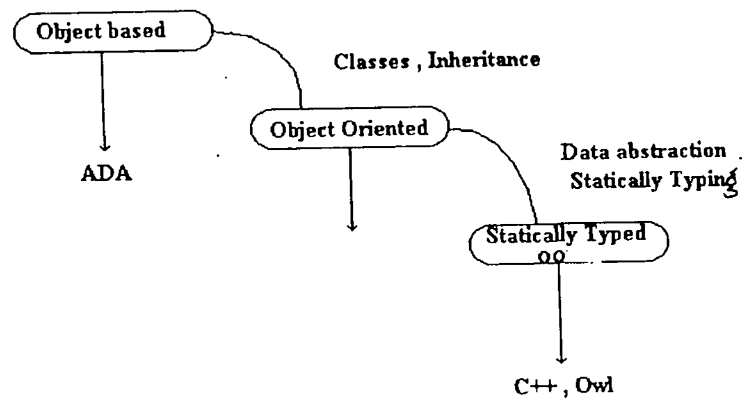


Fig. 1 Statically-typed object-oriented language

3.3 An Object Migration Model

In this section the general objectives of the migration modeling is discussed. The notion of base and residential classes which are fundamental to object migration modeling is then defined.

3.3.1 Objectives of Object Migration Modeling

Consistent with most current OODB models and systems which are based on statically-typed OOPLs, the basic concepts of class, inheritance, and object identifier (Oid) has been assumed. A class defines a set of instance variables (or attributes) and a set of procedural methods for each of its objects (members). The methods associated with a class, either explicitly defined or

implicitily inherited, are refered to as the behaviour of the object in the class. An OODB conceptual schema is a singly-rooted tree, embodying the ISA hierarchy structure among the classes, with root being OBJECT.

Updates in an OODB can be divided into two types: one is concerned with change of attribute values (value part), and other is concerned change of of class memberships (migration part). The theme of this project is on the latter, that is, object migration part. From a database management perspective, the following are three specific goals for object migration modelling:

3.3.1.1 Identity Preservation Goal

This is to preserve the identity of migrating object during migration. As the problem addressed here are object migration and not object deletions followed by creations, it therefore is absolutely necessary to preserve the same Oid of an object throughout its migration period (and its life time).

3.3.1.2 Behavior Transitional Goal

This is to give a migrating object the ability to abandon old behavior of the source class and to acquire new behavior from the target class. In an object oriented database, an object is accessible and operable from its 'external interface' which consist of a set of methods (behaviour) defined by its class. The migration of an object from one class to another should therefore reflect the change of its behavior by switching to the new behavior defined in the target class.

3.3.1.3 Information Conservation Goal

This is to preserve as much as possible applicable information associated with a migrating object. More specifically, if an object O is to migrate from a source class to target class, then all the properties (attributes and their values)

of O as a member of the source class that are still applicable as a member of the target class should be retained. Note that in our model, we assume 'applicable' properties to be those attributes which are common to O's source class and target class, and O's value of these attribute in the context of sources class are still up-to-date in the context of the target class. Thus this information conservation goal does not apply to so-called overwritten and/or homonymous attributes [18]; in this model, such 'variants' are simply treated as different attributes for simplicity.

Of course, each migration instance may achieve a combination of the above three goals.

3.3.2. Base and Residential Classes

There are many situations where object migration can occur. Fundamental to object migration modelling is the ability of an object to dynamically change its memberships and to be member of two ore more classes simultaneously. Multiple memberships of objects correspond intuitively to the notion of roles [6,7] in particular, an object O has a role C if the search for properties are methods associated with O can start with C, not with the designed class in the traditional approach. And an object O is said to reside at a class C if it is explicitly given role C.

A **base class** C of an object O is a distinguished class where O resides; initially C is the class where O is constructed (created); at any given time, each exiting object has a unique base class.

A **residential class** of an object is a class in which the object currently resides. There can be more than one residential class for an object. Residential classes other than the base class of each objectcan only result from object migrations.

Clearly a base class is a special residential class conceptually, each object is existence dependent on its membership to its base class. That is, if an object O ceases to have a base class then O ceases to exist in the database, and thus loses membership to all its residential classes. The existence of the unique base class for each object from the perspective of overall object management, also facilitates efficient object access while providing the supports for multiple perspectives (roles) of an object.

The notion of base class defined above corresponds to the term base role and residential class corresponds to the term aspect roughly. Besides these two kinds of class, an object O can also be accessed from any superclass of its base class or any superclass of the residential classes. For clarity, such superclasses are called accessible classes of O. That is, an accessible class of O is a superclass of O's base class or residential class. From role modelling point of view [7], the base class and residential classes of O are the explicit roles currently being played by O. Due to the inheritance hierarchy, each accessible class can also be assumed by O as an 'implied role'. Thus the total set of role play-able by O is the union of all the explicit and implied roles. But O cannot change its implied roles explicitly.

In most current object database models and systems (e.g. based on C++) each object has exactly one base class, and possibly a number of accessible classes (which are the superclass of the base class). It is not allowed for an object to have a residential class other than the base class (i.e. the only residential class is the base class itself), nor is it possible for an object to dynamically change its base class or acquire new residential classes. These are exactly the problems that this object migration model (OMM) is targeting at, as they embody a significant type of restrictions that current OODB models and systems are imposing upon active/dynamic applications.

3.3.3 Migration Varieties

From an individual object (O) point of view, there are two aspects associated with O's migration. One is the manner of migration i.e. whether the migration is in the form of one-to-one or one-to-many. In a one-to-one (1:1) migration, the migrating object O is migrating from a current residential class to a new residential class, whereas in a one-to-many (1:M) migration, O is migrating from a current residential class to several new residential classes simultaneously.

Another associated aspect is the nature of migration i.e. whether migration is a destructive one or a non-destructive (called constructive) one. In destructive migrations, a migrating object ceases to be a member of a current residential class, after it becomes a member of new residential class(es). In constructive migrations, however, a migrating object acquires new residential classes while still keeping its current residential classes hence all accessible classes. Constructive migration is more general than multiple inheritance in the sense that shared subclasses can be modeled by constructive migrations, but not vice versa.

As the two aspects (viz. the manner and the nature) associated with object migration are orthogonal, therefore the following four possible object migration cases are given below:

1. One-to-one constructive
2. One-to-one destructive
3. One-to-many constructive
4. One-to-many destructive

3.3.3.1 One-to-one Constructive

This type of migration refers to the situation where an object acquires an additional residential class while it still keeps its current residential class. As an example, a scholar Sita becomes a immigrant to American while keeping her Indian citizenship. Pictorially, a dashed line with an arrow to denote such a 1:1 constructive migration (see **Fig. 2**)

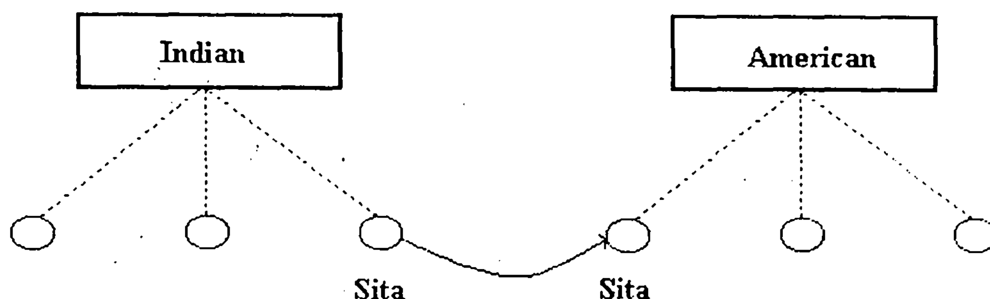


Fig. 2 A 1:1 Constructive Migration Case.

3.3.3.2 One-to-one Destructive

This correspond to the situation where an object stops being a member of a residential class (residential class can be either a non-base class or the base class), and takes up membership of another residential class for example, person Rama stops being Single, and becomes Married. Pictorially, a solid line

with an arrow to denote such 1:1 destructive migration occurrence (see **Fig. 3**)

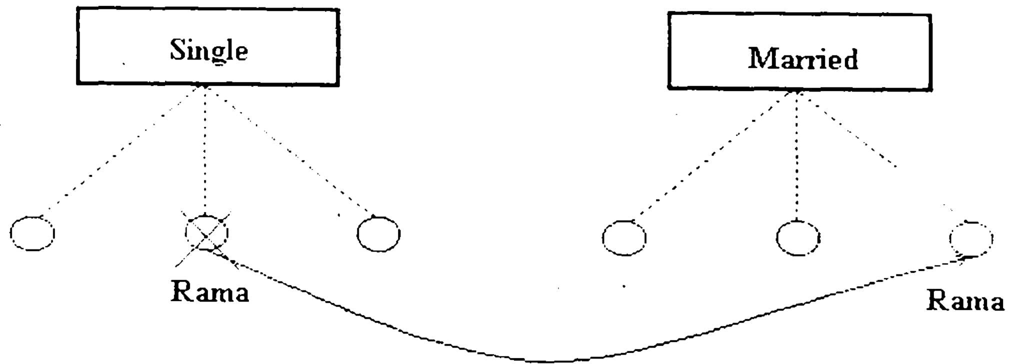


Fig 3 A 1:1 Destructive Migration Case

3.3.3.3 One-to-many Constructive

In this kind of migration, an object becomes a member of a number of new classes while still keeping membership in its current classes. As an example, a secretary Rita becomes a parent and wet-nurse while still keeping her job. A 1:M constructive migration is a general case of 1:1 constructive migration, as it is in effect equivalent to executing several 1:1 constructive migrations. Pictorially, a dashed line with forked arrows to denote such 1:M constructive migration (see **Fig.4**)

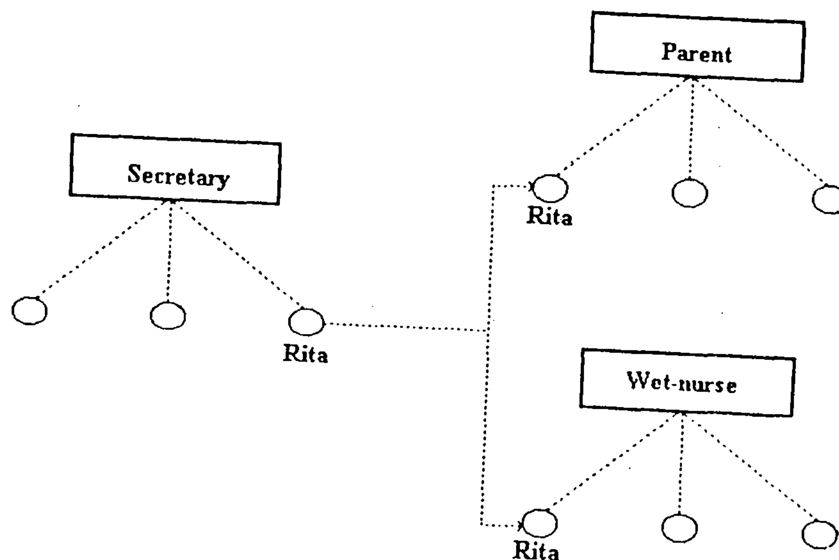


Fig. 4 A 1:M Constructive Migration Case.

3.3.3.4 One-to-many Destructive

This kind of migration corresponds to the case in which an object stops being a member of a class, and becomes simultaneously members of two or more other classes. For example, a M.Tech student finishes his/her study, and becomes a lecturer and an engineer at the same time. Due to the destructive nature, a 1:M destructive migration is not equivalent to simple composition of several 1:1 destructive migrations. Pictorially, a solid line with forked arrows to denote such a 1:M destructive migration (see **Fig.5**)

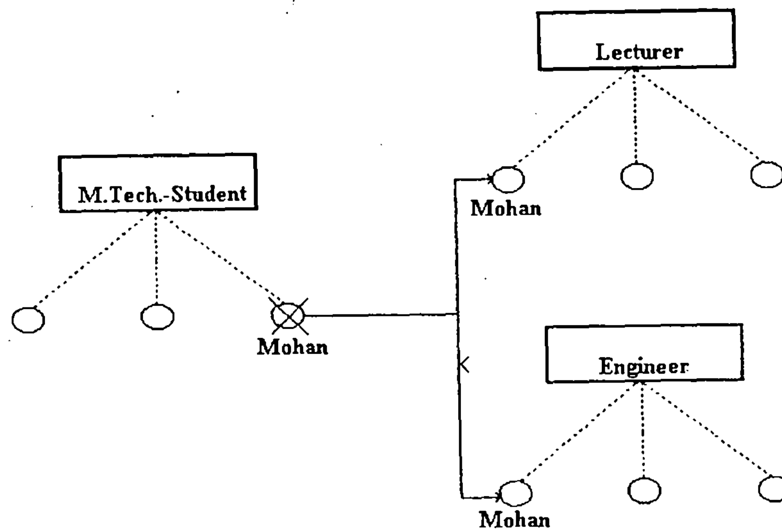


Fig. 5 A 1:M destructive migration instance

***

Chapter IV

THE SCHEME

The scheme involves to accommodate the kinds of migration cases described in **Chapter III**, several migration operators have been defined within migration model (OMM). All the migration types has been covered.

4.1 Migration Operators

The following describes all the operators that have been incorporated in prototype implementation.

4.1.1 RELOCATE (O,C1,C2): Boolean

The RELOCATE operation relocates an object O from its current- class C_1 (which is either a residential class or its base class) to the specified target class C_2. The obsolete properties (attributes and methods) of O (those associated with O's current class C_1 but not with C_2) are deleted, and the new properties (those associated with C_2 but not C_1) are added, with default values being 'unknown'. For example, RELOCATE(Rama, Single, Married) will relocate Rama from current base class (viz. single) to Married. If Rama was not in the class single, this operation will just return false.

As a result of a successful RELOCATE operation for an object O, O's Oid is required to be 'moved' from the current class (the source class) C_1 to the target class C_2 (thus this operation corresponds to a 1:1 destructive migration). If the current class C_1 is a residential class, then after this operation C_1 ceases to be O's residential class while C_2 becomes O's residential class. Otherwise

(i.e., C_1 is the base class of O), O is removed from C_1 and added to the class C_2 which becomes the new base class of O. This operator corresponds to the 1:1 destructive migration case specified in the **Chapter III**.

4.1.2 PROPAGATE (O,C,C1[,C2...,Cn) Boolean

The PROPAGATE operation propagates an object O from the current class C (which is either a residential class or the base class) to the target classes C_1 through C_n. For each target class C_j ($1 \le j \le n$), the obsolete attributes of O (those associated with O's current class C but not with C_j) are deleted, and the new ones (those associated with C_j but not with C) are added, with default values being 'unknown'. For example, PROPAGATE (Sita, Indian, American, Canadian) will propagate Sita to the target class American and Canadian, while Sita still maintains the current Indian membership. If O was not a member of C, this operation will have no effect and just return false.

As a result of such a PROPAGATE operation for object O, O's oid is perceived to be 'copied' to all the target classes, which causes new residential classes being 'introduced' to O, while O's current class (viz. the residential classes or base class) are not effected. This operator covers both 1:1 and 1:M constructive cases.

4.1.3 TRANSMIT (O,C,C1[,...,Cn) Boolean

The TRANSMIT operation migrate an object from its current class C (which is residential class or the base class) to the target class C_1 through C_n simultaneously. For each target class C_j ($1 <= j <= n$), the obsolete attributes of O (those associated with O's current class C but not with C_j) are deleted, and the new attributes (those associated with C_j but not C) are added, with default values being 'unknown'. For example, TRANSMIT (Shyam, PhD_Student, Engineer, Lecturer) will transmit Shyam from the current class PhD_Student to Engineer and Lecturer. If Shyam was not in the class

PhD_Student, this operation will have no effect and simply return false.

The TRANSMIT operation resembles both the PROPAGATE and RELOCATE operation. When the number of target classes is equal to 1 (i.e. n=1), this operator is obviously same as the RELOCATE. When n>=1, however, the effect of this operator is different from that of executing several RELOCATE operations simultaneously. In particular, when the current class C is the base class, the operation exhibits a 'hybrid effect of executing PROPAGATE and RELOCATE operations : it is equivalent to executing PROPAGATE($O,C_1,C_2[,...,C_n]$ followed by RELOCATE($O,C,\hat{C}$), where $\hat{C}$ is the most direct common ancestor (superclass) of the target class C_1,C_2 through C_n. Thus the new base class of O is changed to $\hat{C}$ in this case. So for the same example above, if the class PhD_Student was the base class of Shyam, the operation TRANSMIT(Shyam, PhD_Student, Engineer, Lecturer) will transmit Shyam to the target classes Engineer and Lecturer, and cause the base class of Shyam to be changed from PhD_Student to the most direct superclass (e.g. Professionals) of the two target classes.

4.1.4 RETRACT ($O,C_1[,C_2,..C_n]$: Boolean

The RETRACT operation is essentially the 'inverse' of the above migration functions. It allows an object to abandon dynamically some (or all) of its current residential classes. For example, the operation RETRACT(Gopal, Canadian, Australian) will cause Gopal to relinquish Canadian and Australian as its residential classes. In this operation, we do not have to specify O's source class but only those (target) residential classes to be relinquished. This operation will return false (and operation causes no change) if

(i) the class is the base class, or

(ii) any of the specified target classes is not residential class of O.

4.2 Object Migration Control

A simple object migration mechanism has been provided in the above section to support specific migration cases. Care must be taken in the conducting various migration operations so that only meaningful and correct migrations are conducted. There are situation where migration is meaningless or incorrect. This motivates the need for devising a migration control mechanism to be attached to the migrat ion mechanism in order to ensure that only meaningful migrations are conducted.

4.2.1 Examples of meaningless migration

Just as there are any situations where object migration is a natural thing to occur, there also many situations where object migration is meaningless, in particular, it does not always make sense to have migration

(i) from a subclass to superclass,
(ii) from a superclass to a subclass, or
(iii) from a subclass to another subclass (even when the application justifies migration from the second subclass to the first subclass).

Example:

Consider the example where person is the (virtual) generalization of a Male and Female. For (i) and (ii), there should be no migration from person to Male [illegible] or vice versa, since person should have no explicit object which are not member of Male or Female. Further more one case of (iii) there should be no migration form Male to female or vice versa, since the member of Male and Female are mutually exclusive.

To illustrate the other case of (iii), consider the situation where Teenager and Adult are specializations of Person. Then, consistent with real world semantics, object may migrate from Teenager to Adult, but not vice versa (i.e. a person cannot change from an Adult back to Teenager).

Since object oriented database system are are set to model the real world, object migration in the systems should only reflect real world object migrations. That is, object migration in the system should be disallowed if the do not correspond to meaningful migrations in reality. In order to prevent meaningless object migrations from occuring, there should be a facility for controlling object migrations in the system.

4.2.2 Migration Control Specification

The ingredient of migration control specification are migration which, in a sense, are similar to semantics constraints in database by requiring that object migration can and only can occur by following the specified migration permissions.

The TRANSMIT operation can be defined by PROPAGATE and RELOCATE. Each PROPAGATE operation require a single target class, since each such operation with multiple target classes is equivalent to a sequence of such operations with single target classes.

There are three kinds of migration permissions. For the first two kinds, assume that the source class and target class are distinct.

(1) A relocation permission is a triple of the form (source_class, target_class, relocation). It captures the semantics that each object with source_class as its base/role class may be relocated to target_class.

As an example, the triple (Single, Married, relocation) is a relocation permission. If Rama was previously single,and is getting married now, then Rama can be relocated from Single to Married. Thus a relocation permission prescribes one-to-one (1:1) destructive migration from source_class to target_class.

Relocation permission may appear as dual as exemplified by (Single, Married, Relocation) and (Married, Single, relocation). However, this is not always the case. Take the Teenager and Adult example: (Teenager, Adult, relocation) is a meaningful migration permission, whereas (Adult, Teenager, relocation) is not.

(2) A propagation permission is a triple of the form (source_class, target_class, propagation). It captures the semantics that each object with source class as its base/role class may be propagated to target_class.

As an example, the triple (Professor, Consultant, Propagation) is a propagation permission. Thus if Bimal is a Professor, it is possible for him to acquire additional role such as a consultant to company.

(3) A retract permission is pair of the form (class, retract). It captures the semantics that the RETRACT operation can be applied to the object having class as the role class.

Control mechanism can be defined as follows :

A migration control specification Σ over an object schema is a finite set of relocation, retract and propagation permissions, where the participating class are from the schema.

RETRACT should always be allowed. Thus in every migration control specification, there is (implicitly) a retract permission for every class. Note that some instance of RETRACT may cause no change to object base, as specified in the semantics of RETRACT.

A migration control specification Σ is created when the database is designed, and is updatable when the database schema is changed (e.g., when new classes are added and/or existing ones deleted/updated).

4.2.3 Inference of Migration Permissions

A migration transaction is sequence of zero or more migration operations on one object, whose effect is achieved by executing the component operation from left to right. A migration transaction is legal with respect to a migration control specification if each component operation is permitted by the control specification.

A migration control specification Σ implies a migration P if

(i) P is in Σ or

(ii) P has the form (C_1, C_2, mode), there is an object O having C_1 as role class, and there is legal migration transaction T such that after executing T,

- O gains C_2 but loses C_1 as role classes if mode is relocation;
- O gains C_1 and keeps C_2 as role classes if mode is propagation;

The closure set Σ^+ of Σ is the set of migration permissions from a given migration control specification.

Inference Rules

(1) **MMM** : If (C_1, C_2, M) and (C_2, C_3, M) are in Σ, where $C_1 \neq C_3$ and M is either relocation or propagation, then Σ implies (C_1, C_3, M).

(2)**PRP** : If (C_1, C_2, propagation) and (C_2, C_3, relocation) are in Σ, where $C_1 \neq C_3$, then Σ implies (C_1, C_3, propagation).

(3) P2R : If (C_1, C_2, propagation) is in Σ, then Σ implies (C_1, C_2, relocation).

An inference rule is redundant in a set of *I* of inference rules if, for all migration control specification Σ and migration permission P, one can infer P from Σ using *I*-{i} whenever one can infer P from Σ using *I*; and *I* is minimal if no rule in *I* is redundant. *I* is sound if Σ implies P whenever one can infer P from Σ using *I* . *I* is complete if one can infer P from Σ using *I* whenever Σ implies P.

P2R has been used as a shorthand for 'propagation implies relocation', and PRP used as a shorthand for 'propagation followed by relocation'. The MMM rule actually stands for two rules : PPP and RRR. Of the eight potential inference rules without 2 : PPP, PPR, PRP, PRR, RPP, RPR, RRP and RRR, it is easily seen that RPP and RRP are not used. Further more, RPR,PPR and PRR are all redundant: form P2R and RRR we get RPR, from PPP and P2R we get PPR, and from P2R and PRP we get PRR.

4.2.4 Path Sensitivity of Migration

To illustrate that migration an object along different migration paths may lead to different final states, although the final role classes are the same. This Phenomenon is called path sensitivity.

A migrating path is a sequence of zero or more migration permissions. A migration transaction follows a migration path if ith operation in the transaction corresponds to the ith migration permission in the path for each i.

Example:

Suppose the attributes of C_1 and C_3 are A and B, an attributes of C_2 are A and D. Path sensitivity exhibits by considering the following two migration paths:

(C_1; C_3, relocation) and

(C_1, C_2, relocation) (C_2,C_3, relocation).

Suppose O is a member of C_1 and O.A = a and O.B = b. Then, RELOCATE (O, C_1, C_2), RELOCATE (O, C_2, C_3) will result in O as a member of of C_3 with O.A = a and O.B = 'unknown', whereas RELOCATE (O, C_1, C_3) alone will result in O as a member of C_3 with O.A = a and O.B = b. Hence object migration from C_1 to C_3 in both cases can be achieve with respect to identity, but the resulting databases states differ since information is preserved for the two cases.

Migrating objects follow different migrating paths lead to different degrees of information preservation. When limited to a pair of initial and final role classes, migration following direct migration permissions always preserve maximal information, where as indirect steps may sometimes cause 'information loss'.

4.2.5 Information-preserving completeness

A role set is a non empty set of classes. A role states of an object is the

set of all of its role classes. A set of migrating operators is complete if, for each pair of role sets S_1 and S_2 and for each object O with S_1 as its initial role state, there is a migration transaction T which maps O to a state with role state S_2.

It can be shown that {RELOCATE, PROPAGATE} and {PROPAGATE, RETRACT} are equivalent in expressive power, in the sense that each role_state transition implementable by using the first set operators is also implementable by using the second set of operators and vice versa. This is not the case, however, if information preservation is taken into account.

The notion of completeness as given above is good in reflecting reachability. However, its fails to capture reachability with information preservation. The migration transaction RELOCATE (O, C_1, C_2), RELOCATE (O, C_2, C_3) is not desirable as RELOCATE (O, C_1, C_3) from information preservation point of view : though to achieve object O's migration from C_1 to C_3 in both cases, the latter transaction allows us to retain more information than the former (longer) transaction.

For a class hierarchy, a migration transaction T on object O is information-preserving if, for each initial role class C_1, final role class C_2, each attribute A common to C_1 and C_2, O.A is defined in C_2 (resp. retaining its values defined in C_1).

A set of migration operator is information-preserving complete if, for each pair of role sets S_1 and S_2 and each object O having S_1 as its initial role state, there is an information-preserving migration transaction T using the given operators such that the T maps O to a final state with role state S_2.

4.3 Specification Design and Implementation

Based on migration specification facility introduced in previous section, and the analysis of its properties thereof, are taken care for migration

specification design. As part of schema design task which is more of art than science, migration specification design is full of uncertainty and multiple choices. For example, these may be many different paths to follow in accomplishing seemingly identical migrations which are in reality different. In particular the following specific questions are examined:

- Given an object database, what is the most critical aspect to consider in designing the migration specification Σ (knowing that there are several aspect and ways to define it)
- Given a specification Σ, what are the most desirable migration transaction to define (i.e., the 'optimal' transactions and most desirable order of executing the transaction)

To conduct object migration more effectively, a set of heuristic quidelines based on the studied properties of migration paths have been given. An elegant declaration facility for the user (designer) to directly use, which is an extension to statically-typed object database system upon which experimental prototype is implemented.

4.3.1 Design Constraints

Suppose that there exits a triple (source_class, target_class, mode) is a migration control specification. Let class_z be a minimal common super_class and target_class.

- to obtain nontrivial information preservation, class_z should not be OBJECT.
- the lower the level class_z is in the class hierarchy, the more information preservation this triple entails, thus the more desirable the triple is.

- The smaller the vertical distance from source_class and target_class to class_z, the more information preservation this triple entails, thus the more desirable the triple is
- using fewer operations usually preserves more information,
- it must be ensured that only meaningful and desirable; this should be the main factor to considered.

4.3.2 Implementation

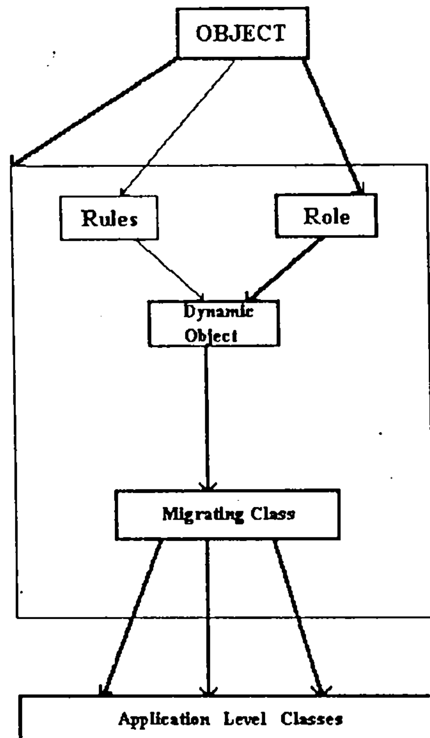
An experimental prototype embodying the above migration facilities have been implemented. The prototype system is based on a conventional object-oriented model (C++) which support such conventional object mechanism as classes, attributes (instant variables), methods subclass (ISA) relationship and inheritance, etc. Major aspect of the prototype experiment include the implementation of the aforementioned migration functions, and the declaration facility for describing permissible migration paths (PMPs) of classes.

There is meta class called DynamicObject that serves as the superclass of MigratingClass, which provides several primitive (base level) functions for implementing the migration operators; it in turn is defined based on another meta class called Role whose superclass is OBJECT (which provide for persistency). The meta class structure in in **Fig.6**.

The declaration of PMPs for a migrating class has the following syntax:

```
PMP set_of_objects      [relocatable_to list_of_classes]
                        [propagatable_to list_of_class-sets]
                        [transmittable_to list_of_class_sets]
                        [unmigratable_to list_of_classes]
```

Square brackets indicate optional arguments. The 'set_of_object' arguments can be one of the following forms : a set of enumerated object identifiers (Oids), a set of attributes defined Oids (as result of selection



Legends:

built in meta class

super /subclass relationship

class planned to be added

Fig 6. Structure

process based on the attributes from the class), the complete set of objects of the class (the default one). Similarly,the 'list_of_classes' arguments also can have several forms: a list of enumerated class names (or identifiers), the result of selection process stored in given meta-class, the complete list of subclasses of a given class, the complete list of superclass of a given class. The 'list-of-class-sets' arguments has similar possibilities, except that it is list of class sets.

An unspecified migration path is not necessarily illegal, but unknown. When a role/migrating class is defined, associated PMPs can be specified in an incremental manner. For convenience in practice, unmigratable paths can be declared, which are known to be illegal, and even change a path from migratable to unmigratable, and vice versa.

These features help in the situations where the nature of some migrations are not clear ar uncertain, and allow to be able to highlight (un)migratable paths to emphasize. For example, the following specification embedded in the definition of the class Student prescribes the permissible migration paths for those students who has SGPA $<$ 5.0.

PMP Student (SGPA $\leq$ 5.0)
relocatable_to Expelled
propagatable_to [Probational]
transmitable_to [Expelled, Switched]
unmigratable_to Honor_Student

The implication of this specification is that a student with SGPA $<$ 5.0 is regarded as in poor academic standing, and thus is possible to be asked to quit the program (i.e. to be expelled), or quit the current program and switch to less intensive one (reflected by transmittable path), and under no circumstances he could become a member of honoured students. Clearly, such PMP specifications can be certain uniformity along the ISA hierarchy, meaning

that a subclass may inherit from its superclass appropriate PMPs (just as if it can inherit attributes and methods from its superclass). It is implemented in selective inheritance way in prototype. Hence a subclass can selectively veto the inheritance of certain (or all) migration paths from the superclass, and for the inherited one, the subclass can also modify, refine the migration list.

***

CONCLUSION

In this work dynamic object migrations have been accommodated in statically-typed object databases. Although the work described here is for statically-typed (strongly-typed) object-oriented databases (OODBS), many of the principles and techniques are also applicable to other non-statically typed OODBS. In particular, the criterion of information preservation being the main concern for migration is clearly of equal importance to any type of OODB systems. The idea of declaring migration paths in a migration specification can be used as a means to control and to avoid abuse of migration functions in any OODB system.

However, as shown by example in section 4.2.4, the information preservation goal is not always satisfiable due to path-sensitivity, as well as the inherent properties of the class hierarchy.

***

BIBLIOGRAPHY

1. Cardelli, L. and Wegner, P., "On understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys*, December 1985.

2. Hughes, J.G., "*Object-oriented database*", (Prentice Hall,1991).

3. Stroustrup, B., "*The C++ Programming Languages*", (2nd ed.) (Addison-Welsey).

4. Meyer, B., "*Object-oriented Software Construction.*", (Prentice-Hall, 1991.)

5. Shafferi, C., Copper, T., Bullis, B., Killian, M., and Willpolt, C., "An Introduction to Trellis/Owl", in: Proc. conf. *on object-oriented programming Systems, Languages, and applications*. (1986)

6. Sciore, E., "Object specialization", *ACM trans. Informat. Syst.* 7 (2)(April 1989) 103-122.

7. Li, Q. and Mcleod, D., "Coceptual dadtabase evaluation through learnig in object databases", *IEEE Trans. Data and knowledge Eng.* 6 (2) (April 1994).

8. Elmasri, R., Weldreyer and Hevner, A., "The category concept: an extension to the entity-relationship model", *Data and Knowledge Eng.* 1 (1) (1985).

9. McCarthy,D.R., Dayal,U., "The architecture of an active Database Management system", *Proc. ACM-SIGMOD*,1989.

10. Dittrich,K.R., Preface, in: Dittrich,K.R.(ed.): "Advance in Object-Oriented Database Systems", *Lecture Notes in Computer Science*, Vol. 334, Springer, 1988.

11. Connors,T., Lyngback,P. "Providing Uniform Access to Hetrogenous Information Bases", in: Advance in Object-Oriented Database Systems, Dittrich,K.R.,(ed.), *Lecture Notes in Computer Science*, Vol. 334, Springer Verlag,1988.

12. Cardelli,L., "A Semantics of Multiple Inheritance, In: Semantics of Data Types", *Lecture Notes in Computer Science*, Vol. 173, Springal Verlag, pp. 51-67,1984.

13. Goldberg, A. "Introducing the Smalltalk-80 System" *BYTE Magazine*, August 1981.

14. Rowe L., "A Shared Object Hierarchy", In: Topic in Information Systems, Dittrich, K.R., Dayal,U., and Buchmann (Ed.)" *Object-Oriented Database Systems*, Spring-Verlag, pp.171- 188.

15. Wiedershold, G., "Views,Objects, and Databases", In: Topic in Information Systems, Dittrich, K.R., Dayal,U., and Buchmann (Ed.)" *Object-Oriented Database Systems*, Spring-Verlag, pp.29-44.

16. Pernici,B., "Objects with Roles", In: *Conferences on Office Information Systems* (ACM,1990) pp.205-215.

17. Su,J., "Object Migration Patterns" in: *Proc. Int. Conf. on Very Large Databases*, VLDB Endowment (1991).

18. Chorafas, Dimitris,N. and Steinmann, H., "*Object-Oriented Databases*", PTR Prentice Hall, Englewood Cliffs, New Jersy 07632.

19. Couplin, James,O., "*Advance C++ Programming styles and idioms*", Addison-wesley, 1992.

20. Stevens,AL.,"*C++ Database Development*" (second ed.), BPB publications, New Delhi, 1995.

21. Partridge,C., "Modeling the real world: Are classes abstraction are objects" ? *Journal of Object-Oriented Programming*, Nov.Dec.1994, pp. 39-45.

***