ISBN 0-315-59143-9

Temporal Object-Oriented DataBase - A Data Model

Rajwantbir Singh Kohli

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

June 1990

ABSTRACT

Temporal Object-Oriented Database - A Data Model

Rajwantbir Singh Kohli

In this thesis a data model (Temporal Object Oriented Database) is discussed. This model incorporates the concepts of Object-Oriented Design with the concepts of time to model the real world and support applications from the CAD/CAM, AI, OIS, etc. domains. The System supports schema evolution, object (instance) evolution, versions, multityping and history maintenance. The system provides a natural desig· environment with temporal capabilities. The model can support versions and schema management for designs as they evolve (i.e., schema evolution with time), and concurrency control for cooperative work in multiuser environment (the issue of concurrency is not discussed in this thesis).

Dedicated to

My Parents

and

D. S. Kohli

ACKNOWLEDGEMENTS

There were significant contributions from a number of people without which this work could not have been completed successfully. First and foremost, I would like to express my sincerest thanks to my two advisors Dr. F. Sadri and Dr. P. Goyal. I consider myself singularly fortunate to have had the opportunity of working with them and sharing their experience. I am thankful to Dr. Goyal for introducing me to the area of Object-Oriented Databases and their problems and motivating me to pursue graduate studies, for which I am very grateful. I am thankful to Dr. Sadri, for his invaluable guidance, immense patience, and encouragement through every step of the work leading to the preparation of this thesis. He had to work harder than me in order to help me to mould my fuzzy half baked ideas and present them precisely. He taught me the art of tackling problems in a top down manner and conceptually structuring them. Their patience, encouragement and moral support are deeply appreciated.

Differential fees and inflation make the financial aspect the most worrisome issue for international students. I am thankful to Dr. Goyal and Dr. Sadri for sponsering my graduate studies and providing the financial support, thus mitigating my financial woes.

I am also thankful to Dr. Peter Grogono for helpful discussions and giving me valuable insights into the issues of Object-Oriented Programming. I thank my fellow students of the Cave, for providing a stimulating atmosphere for discussions and criticism. Special thanks to Dr. T. S. Narayanan, Dr. Premchand

Nair, Derek Pao, Hasan Jamil and Basudeb Dash with whom I closely interacted during my tenure here. Dimitri Livas, Jason Cheng both helped me with the implementation problems and providing a better understanding of the UNIX system's features. Dash and Jamil both did the daunting task of proofreading this thesis and providing valuable suggesstions.

Last but not least, I would like to express my deepest gratitude to the members of my family for their unwavering support and encouragement during the entire course of my studies.

Contents

List of Figures

CHAPTER 1

INTRODUCTION

Many new data-intensive applications in computer-aided design and manufacturing (CAD/CAM), artificial intelligence (AI), office information systems (OIS), multimedia information systems, computer-integrated manufacturing (CIM), computer-aided software engineering (CASE), etc., are quite complex to model using conventional data base systems (network, hierarchical, or relational). The domains of these applications are not static. Their domain is constantly evolving, (domain refers to the conceptul structure of the application). It undergoes refinement over period of time. These environments are characterized by constant change, hence they are not static, thus rendering the conventional data base management systems inadequate to model and support the needs of these applications. The conventional database management systems also lack the capability to record and process time-varying aspects of the real world, support historical queries about past status, permit trend analysis - essential for decision support systems, and represent retroactive or post-dated changes.

Currently most of the research work in this area is concentrated on Object-Oriented Databases, for example [6,7,11,13,16,17,19] are related to ORION system, carried out at MCC. Their system (ORION) caters for schema evolution and version control but does not support temporal capabilities. GEMSTONE/OPAL of Serviologic, [22], is an Object-Oriented database system, based on Smalltalk-80

[14] is in commercial use, Vbase integrated object-oriented System of Ontologic, Inc. [3], IRIS [12], Graphael's G-Base, and Innovative Systems VISION are other systems in use. The work on temporal database involve extensions of relational model to support temporal concepts [27,28].

Temporal Object Oriented Database integrates temporal aspects into object model to meet the needs of these new applications, enabling one to model objects of any complexity, express relationships among objects and support dynamic evolution of the application environment.

This thesis is organized as follows. In this Chapter the Object Model, i.e., basic object-oriented concepts are reviewed. Chapter 2 describes the Temporal object model and Chapter 3 discusses the proposed implementation issues for the model. Chapter 4 concludes with some open issues for future research.

1.1. OBJECT-ORIENTED CONCEPTS

There are many notions of the term "object-oriented" and object-oriented databases in literature. In this section the basic object-oriented concepts and terminology relevant to our model are reviewed.

The database field is concerned with the management of large amounts of persistent, reliable and shared data. "Large" means too big to fit in a conventional main memory. "Persistent" means that data persists from one session to another, i.e. data (objects) are accessible past the end of the process that creats them. "Reliable" means recoverable in case of hardware or software failures, i.e. data is resilent in the face of process failure, system failure, and media failure. "Sharing" implies simultaneous use of the database by multiple users.

New applications of database technology, CAD, CASE, Office automation, AI etc., need large amounts of reliable, sharable and persistent data. To cater for the needs of these users, the approach adopted is to integrate database technology and the object-oriented approach in a single system - Object-Oriented Database System [5].

Object-oriented database management systems (OODBMSs) use object-oriented concepts (discussed in this chapter) and provide additional characteristics necessary to support large, shared, persistent object stores (facilities for objects that must persist beyond a single transactions execution's life-span). These characteristics include efficient processing over large secondary storage organizations, concurrency control, recovery facilities, and efficient processing of set-oriented requests or queries.

1.1.1. OBJECTS, ATTRIBUTES, METHODS AND MESSAGES

In an object-oriented system, the database is a collection of objects, where all conceptual entities of any complexity from the application domain are modeled as objects. An object is a self-contained entity which has a state, method specification, and method implementation [4]. The state of the object is a list of attributes, each attribute having an associated object type (class), i.e. we can represent objects of any complexity. The method is a property (behavior) exhibited by the object consisting of a name, arity and arguments. Implementation of a method is a internal code which implements the operation.

The object types are defined according to the abstract data type (ADT) concepts. They encapsulate a data structure and its associated operations, the effects of operations being public, the implementation of operation being private [29]. An object has an interface part and an implementation part. The interface part is the specifications of the set of operations which can be performed on the object. It is the only visible part of the object. The implementation part has a data part and an operation part. The data part is the memory of the object and the operation part describes the implementation of each operation in some programming language [5].

Every object is an instance of some type (object type / class) which describes the behavior of its instances. A type T is a specification of behavior. As such, it describes a set of operations *op* (T), a set of attributes *atr* (T), and a set of constraints *con* (T) that pertain to any instance *t* of T. Any operation *o* in *op* (T) can be applied to *t*, and every constraint *c* in *con* (T) must be satisfied by *t* [26].

The behavior of the objects is encapsulated in methods. Methods consist of code that manipulates or returns the state of the object. Objects are autonomous entities that respond to messages or operations and share a state. Objects can communicate with one another through messages. Messages constitute the public interface of an object, they are implicit invocation of the operations. For each message understood by an object, there is a corresponding method that executes the message. An object reacts to a message by executing the corresponding method, and returning an object [6]. A message is sent to an object, called the receiver, to invoke one of the object's operations. The message includes a symbolic name, the selector, which describes the desired operation. It may also contain arguments to be passed to the operation. The message, then, describes what the invoker wants to happen, not how it should happen. The message receiver, in turn, has methods which describe how the operations are performed. A method is like a procedure in that it is comprised of a sequence of executable statements. However, methods are inseparable from the objects they are defined for; a method can only be invoked when the object receives a message whose selector corresponds to that method.

1.1.2. CLASSES, CLASS HIERARCHY, AND INHERITANCE

Similar objects are grouped together into classes. Objects of the same class have common operations and therefore they exhibit uniform behavior. (In our model some objects may have exceptional behavior not common to all objects of a class). All objects belonging to the same class are described by the same set of attributes and methods. They all respond to the same message. Objects that belong to a class are called instances of that class. A class describes the form (attributes) of its instances, and the operations (methods) applicable to its instances.

A class is a template, it contains two aspects: an object factory and an object container. The object factory means that the class can be used to instantiate new objects by performing the operation *new* on the class. The object container means what is attached to the class as its extension, i.e. the set of objects of the system which belong to the class at this time. The user can manipulate the container by applying operations on all the elements [5,9,10].

The process of object class definition by specialization, a top down approach, results in the concept of a class hierarchy, which extends information hiding capability even further. A class hierarchy is a hierarchy of classes in which an edge between a pair of nodes represents the IS_A relationship, that is, the lower level class (node) is a specialization of the higher level class (node) (and conversely, the higher level class(node) is a generalization of the lower level class(node)). For each pair of classes in a class hierarchy with an edge between them, the higher level is called a superclass, and the lower level class a subclass. The attributes and methods (collectively called properties) specified for a class are inherited (shared) by all its subclasses. A class inherits properties from its superclass and forms a inheritance chain. The subclass can add new instance variables (attributes) and methods of its own. It can also define a method with the same selector as one of the superclass's methods; i.e., overriding a method.

When a message is sent to an instance of a certain object class to perform a certain operation on this object and no corresponding method, (which implements the response to the message) has been defined with the object class, the search is continued within the superclass.

When a class inherits from a single ancestor (superclass) it is known as *single inheritance*, and when it inherits from multiple (more than one superclass) ancestors it is known as *multiple inheritance.* model we focus only on single inheritance.

Inheritance allows new specialized classes to be built on top of the generic classes and they inherit all the instance variables (state) and behavior (methods) of the generic class to which they append specialized states and behavior. The specialized class can also define a method with the same selector as one of the superclass's methods; this overrides the existing generic behavior.

We introduce the notion of *Multityping*, i.e., an object may change its type dynamically based on the occurrence of some triggering event, but at any given instance of time the object exhibits the properties of only one object type (class) of which it is member at that moment. That is, an object may exhibit the behavior of several classes. It can be associated with different roles. However at a given instance it can portray only one role, so multityping differs from multiple inheritance. For example, an object may be a professor. a parent, and a musician, portraying different behavior at different instances of time.

Every class inherits from the class OBJECT. This is a class provided by the system which describes rudimentary behavior common to all objects in the system. The OBJECT class has methods for testing for class membership, etc.. These methods can be extended and modified in the object's class to provide more specific behavior. The subclass/superclass relation structures the classes of the system into an inheritance tree for single inheritance rooted at OBJECT. The inheritance relation in case of multiple-inheritance is in the form of a directed acyclic graph (a lattice).

An object is some private data and set of operations that can access that data. An object is requested to perform one of its operations by sending it a message telling the object what to do. The receiver responds to the message by first choosing the operation that implements the message name, executing this operation, and then returning control to the caller. The word message bears strong connotations of concurrency, it is not so. Messages are not a concurrency mechanism, but a modularity mechanism [9]. Messaging creates the encapsulation of the data and procedure that is called an object. One may do partial ordering of messages based on a prior information and introduce concurrency.

1.2. TEMPORAL ASPECTS

Conventional databases represent the state of an enterprise at a single moment of time. Although the contents of the database continue to change as new information is added, these changes are viewed as modifications to the state. with the old, out-of-date data being deleted from the database. The current contents of the database may be viewed as a snapshot of the enterprise. Temporal information, representing the progression of states of an enterprise over an interval of time, is not adequately supported by the conventional database systems.

The database is extended with the concept of time, which is an essential part of the information about the constantly evolving real world. Facts or data are to be interpreted in the context of time, which enable one to perform post-dated and retroactive changes, correct past errors, support historical queries and trend analysis.

To model changing state and changing structure(schema), two kinds of time: transaction (commit) time, and effective (valid) time are sufficient to support and handle temporal information. *Transaction time* (Commit time) is the time the information was stored in the database. *Valid time* (Effective time) is the time when the relationship in the enterprise being modeled was valid. The semantics of valid time are closely related to reality, and the semantics of transaction time are concerned with database activities. Temporal databases emphasize the need for both valid (effective) time and transaction (commit) time in handling temporal information, and both of these are orthogonal time axes.

To interpret the facts in the context of time and characterize the behavior, we need to specify temporal constructs. Temporal constructs like *before*, *equal*, *meets*, etc are supported by the system. These constructs are based on temporal intervals rather than time points as discussed in [1,2].

1.3. RELATED WORK IN SCHEMA EVOLUTION IN OODBMS

Design environments are characterized by continuous change. Traditional database tools do not deal well with certain kinds of change. In particular, changing the database schema in arbitrary ways is a very difficult process. The existing conventional database systems allow only a few types of schema changes: For example, SQL/DS only allows the dynamic creation and deletion of relations (classes) and the addition of new columns (instance variables) in a relation. This is because conventional record-oriented applications do not require more than a few types of schema changes. More over, the data models they support are not as rich as object-oriented data model.

To meet the requirements of the design applications, the System needs to provide flexibility in dynamically defining and modifying the database schema, that is, the class definitions and the inheritance structure of the class lattice. This is an essential feature of Object-Oriented Database System.

Skarra and Zdonik [25, 26] have presented one of the initial work on Schema Evolution. They discuss the problem of maintaining consistency between a set of persistent objects and a set of type definitions that change. They attempt to make a type's changes transparent with respect to programs that use the type. They introduce version control mechanism and a set of error handlers associated with the versions of a type. The handlers effectively expand the behavior defined by each version so that instances of different versions may be used interchangeably by programs. They discuss versioning of a class only, rather than the entire schema.

Class Modification in the GemStone Object-Oriented DBMS [24], discusses the impact of changing types on persistent instances and how to bring existing objects in line with a modified class. The Conversion approach, i.e., change all instances of the class to the new class definition, ensuring that the auxiliary definitions (such as class's methods) agree with the new definition, is adopted by GemStone. In this approach much time can be consumed at the time a class is modified and the underlying database is coerced to conform to the new class definition.

Banerjee et al. [6,7] have considered the problem of type evolution in the context of an object-oriented type lattice. They have classified basic categories of change. For each category, they then provide rules for keeping the lattice self-consistent. They have presented a formal framework for schema evolution, and defined the semantics of schema changes using the framework. The framework is based on a graph-theoretic model of the class lattice. The framework consists of invariants and rules. Invariants are those properties of a class lattice that must be preserved before and after any schema change. The rules guide the selection of one most meaningful option for preserving the invariants, in cases where more than one option is possible. The schema evolution model captures the essential characteristics of a class lattice with multiple inheritance. Their approach however does not take the time into account, an essential component in modeling real world applications.

In [18] Kim and Chou propose a model of versions of schema in a multi-user design environment where the schema of the design objects may undergo dynamic changes. Their model of versions of schema extends the model of versions of single objects (class). They discuss three approaches which allow the users to be able to view and manipulate different sets of objects under different versions of the schema. One is to view the entire schema as a versioned object; this is known as versions of schema approach. Another is to view each class as a versioned object; this is the version of class approach. Another is to provide dynamic views, rather than versions of the schema; this is the view of schema approach. The model includes notions as derivation hierarchy for versions of schema and incorporates the view that each version of the schema captures the state of the database at some point in time and that the state may be inherited into any derived schema versions for read and update.

CHAPTER 2

THE TEMPORAL OBJECT MODEL

The Temporal Object Model captures the inherent dynamics of the object and its behaviour, versions of classes and object instances are maintained. Version control is an essential feature of integrated data-intensive applications, to enable the users to experiment with multiple versions of an object before selecting the one that meets their requirements [6]. This chapter discusses the temporal object model and schema evolution.

In an object-oriented system, Class (object type) is the building block of the system, analogous to relations in the relational database system. In a temporal model each class is associated with *commit_time* and *effective_time*, whose semantics are discussed below.

The *database schema* is the class definitions (state and operations) and the inheritance structure of the class hierarchy. The root of a class hierarchy is the system defined class OBJECT, with all classes as its subclasses, as illustrated in Figure 2.1. The class definition comprises of the structural specification (attributes) and the behaviour (methods) of the objects in the class. Each class has only one superclass in this model, where each subclass is a specialization of its superclass. This is known as *single inheritance* , and is shown in Figure 2.2., where class C and class D are subclasses of class A; class A and class B are subclasses of root class OBJECT.

The semantics associated with *effective_time* of a class is that it is the time when the instances of the class are valid, whereas the *commit_time* of a class indicates when the class is available for the users to create instances of it and perform operations on it. The availability of the class is ensured by stabilizing the classes by the system discussed in the following section.

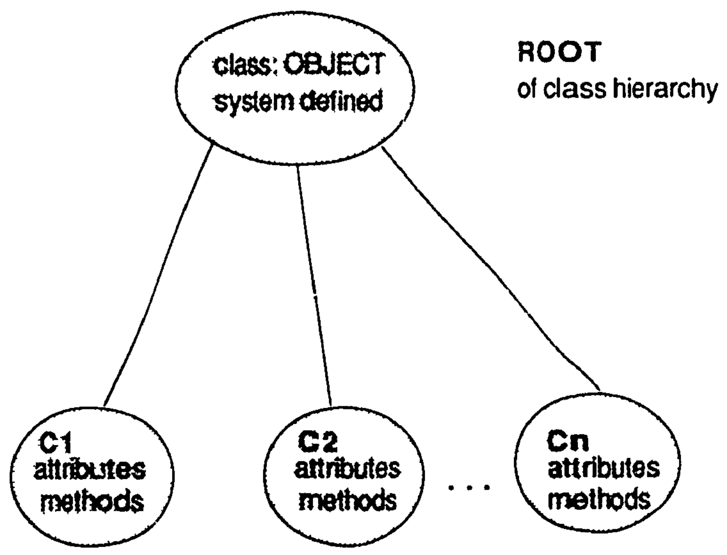


Figure 2.1:Schematic representation of the Database Schema

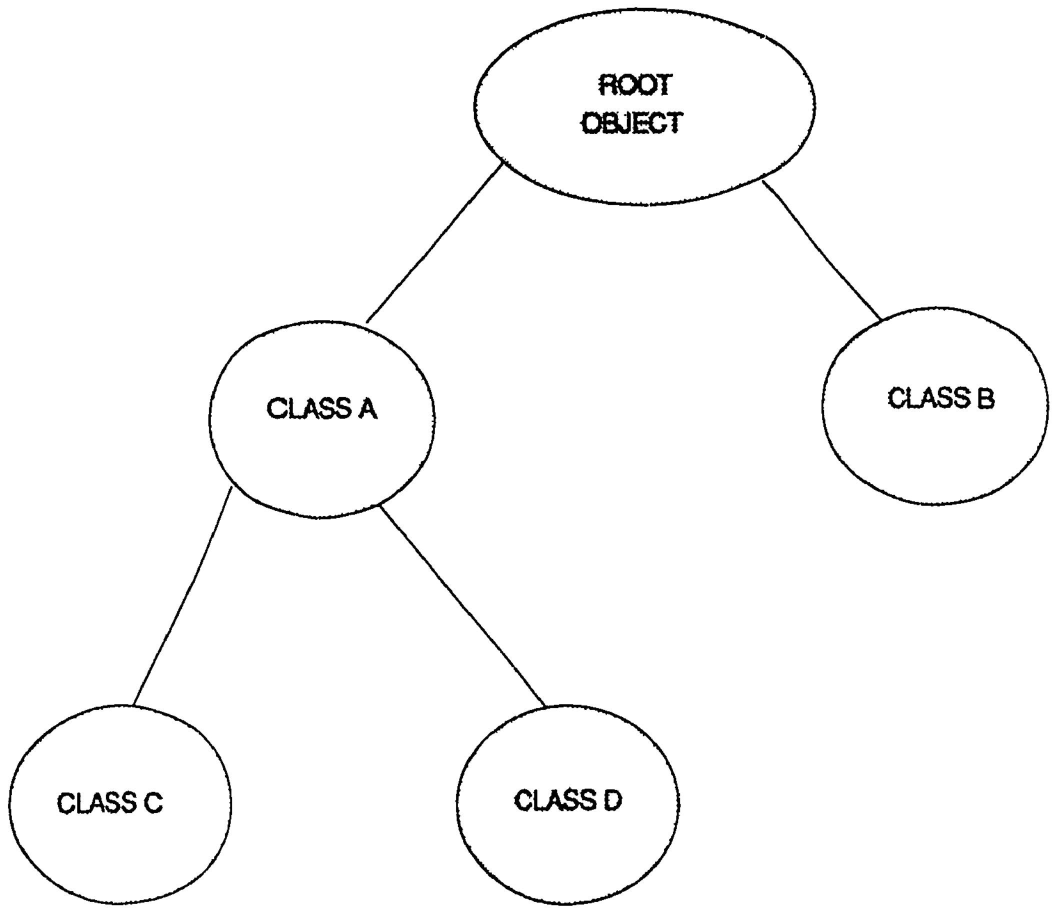


Figure 2.2 : Database Schema - (SINGLE INHERITANCE)

2.1. CLASS DEFINITION

In our model, any real-world entity is modeled as object. An object has associated state and a behaviour. The state of an object is defined at any time by the value of its attributes (called instance variables, elsewhere). Attributes can have as values both primitive objects, such as strings, integers or booleans, and non-primitive objects; a non-primitive object in turn consists of a set of attributes. Therefore objects can be recursively defined in terms of other objects. The behaviour of an object is specified by the methods that operate on the object state. Each object is uniquely identified by a system-defined identifier, which is independent of how an object is accessed or modeled with descriptive data.

Objects with the same properties and behaviour are grouped in classes. Class represents a type template, i.e., specifying the domain associated with each attribute of an instance of the class. A class C consists of a number of attributes, and the domain (value set) of an attribute A of an object instance of C is a primitive object class or some other class C'. The class C' in turn consists of a number of attributes, and their domains are other classes. The class C' is called the *component class* of class C. The fact that class C' is the domain of an attribute of class C establishes a relationship between the two classes. A relationship from class C to class C' specifies that C is defined in terms of class C'.

Classes are organized in a class hierarchy, also referred to as inheritance hierarchy. Inheritance allows the definition of a class, called subclass, as a specialization of another existing class, called superclass. A subclass inherits attributes and methods from its superclass, and in addition may have specific attributes and methods. The model permits versions of a class to exist. A new version of a class is derived from an existing class. The new version is called the *derived class* and the existing class from which it is derived is called the *deriver class.*

Cl: s definition of a class includes the superclass, deriver class if any, structural specification, method specificatior (i.e., interface), implementation and *effective_time* of the class.

The structural specification is a list of <attribute_name, class> pairs, where class is the Object Type of the attribute, thus allowing objects to be defined in terms of other object types (Class) and model objects of any complexity. The primitive object types are Integer, Float, Char and Boolean; which are used as the basis for defining classes. Method specification (i.e., interface) is a list of <method_name, argument_type_list>, while implementation is the internal code, which implements the behaviour (operation). Each method is also associated with its Effective_time.

The creator of the class also specifies the Effective_time associated with the class. The constraint that the effective_time of the method should be same as that of the class or later but not earlier than the class should be satisfied.

Class definitions may involve cycles, i.e., a class C may have a attribute whose domain is class C', and class C', may have attributes, which may directly or indirectly have domain class C. The classes involved in a cycle are called *cyclic dependent classes.*

2.2. CLASS STABILIZATION

The System provides users the flexibility to specify the classes in any order, but before the users can operate on the class, the system must ensure that all components of the class have been specified and satisfy the necessary temporal constraints, only then is the class available to the users, i.e.. *stabilized*, and is assigned a Commit_time by the system when it succeeds in stabilizing the class.

The *Temporal Constraints* to be satisfied by the class are that (i) the Effective times of the *Component Classes* are earlier or the same as that of the class, and (ii) the component classes are stabilized. Stabilization implies that the Superclass of each component class is also stabilized and they also satisfy the temporal constraints amongst themselves.

Consider Figure 2.3, which shows the structural specification of class O. class O contains three attribute of object types C_1, C_2, and C_3, which are user defined classes and are component classes of class O. For Stabilization of class O, the following constraints need to be satisfied.

1. The Effective_time of component classes C_1, C_2, C_3 should be earlier or the same as that of class O.

2. The Superclass of class O should be stable.

3. The Effective_time of Superclass of O should be $\leq$ Effective_time of class O.

4. The component classes should be stable, or should be stabilized at the same time the class is stabilized.

5. If the class is a derived version (discussed in Class Versions), then it's parent deriver class version should be stable.

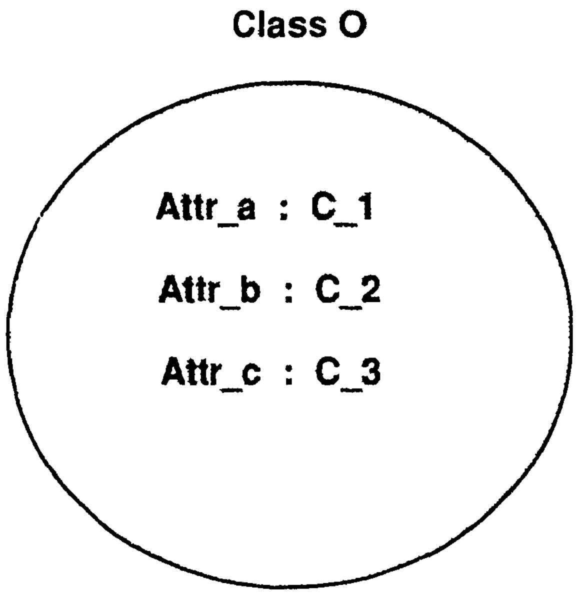


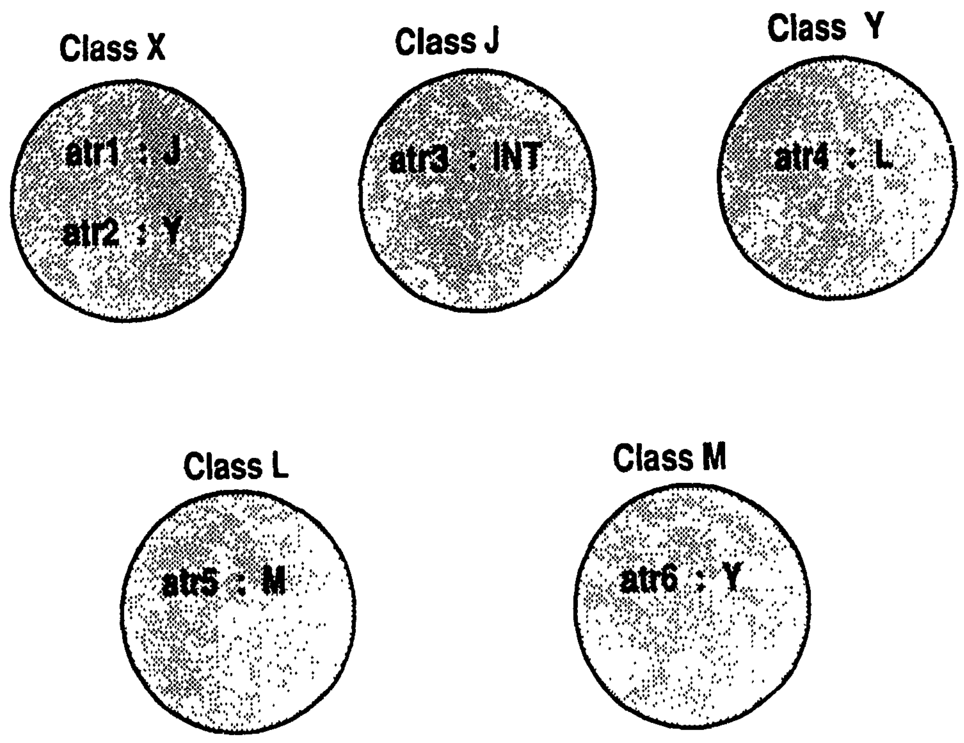
Figure 2.3 : Class Structure

The Object base allows us to model objects of any complexity, i.e., classes can be defined in terms of other classes (Object types), cyclic dependencies can occur amongst classes. The Stabilization process identifies the cyclic dependent classes and ensures that they satisfy the temporal constraints. Consider Figure 2.4, which illustrates cyclic dependencies amongst classes.

Class X, consists of component classes J and Y. Class J is defined using primitive class Integer, whereas class Y is defined in terms of class L, which is defined in terms of class M, and which in turn is defined in terms of class Y. There occurs a cyclic dependency amongst classes Y, L and M.

Stabilization of class X, will generate a directed graph, with a directed edge between the class and component class, for the classes under consideration. To identify the cyclic dependent classes, the strong component of the graph is determined and members of the strong component are cyclic dependent on each other. The *temporal constraint* to be satisfied by the cyclic dependent classes is that they have the same *Effective_time* and *Commit_time.*

So stabilization of class X requires that the class J be stable, i.e., Commit_time of class J be earlier than that of class X, and Effective_time of class J is also earlier or same as that of class X. Classes Y, L and M must have the same Effective_time and Commit_time and should be stable, and their timestamps should be earlier or the same as that of class X.



$$\begin{tikzcd}
 & X \arrow[dl] \arrow[dr] & & \\
J & & Y \arrow[dl] & \\
 & L \arrow[rr] & & M \arrow[ul]
\end{tikzcd}$$

Figure 2.4 : Cyclic Dependent Classes :
Classes Y, L, and M are Cyclically dependent

2.3. OBJECT BASE - SCHEMA EVOLUTION

The data-intensive design applications require the flexibility of dynamically defining and modifying the *object base schema*, that is, the class definitions and the inheritance structure. The changes to schema over time is known as *Schema Evolution*. The changes to the Schema may involve changes to a Class definition in the Class Hierarchy - *Class Evolution*, changes to the links between classes in the Class Hierarchy, and creating new nodes in the Class Hierarchy [6,7]. The approach adopted is that of an Historical database, so nothing is ever deleted, but at a given time only part of the schema may be valid and be of interest to the user.

2.3.1. CLASS VERSIONS

Changes to the contents of a node (Class), i.e., stabilized class, in the class hierarchy results in the creation of a new version of a class. Changes to the contents of a class involve adding/dropping of attributes and methods. Figure 2.5 schematically shows Class Evolution. Class A has two versions, version 2 is derived from version 1. Class evolution results in *version hierarchy* among classes, where each version is an independent class by itself. A version may share specifications of its parent deriver version. Henceforth, class refers to a versioned class. By default, the versions have the same superclass and each derived version should satisfy the constraint that its Effective_time is later or the same as that of its parent deriver version. Each version represents a different view of the object, so one can experiment with multiple versions in design applications.

Figure 2.5 also shows three versions of class B, and two each of class C and class D. Figures 2.6, 2.7a, 2.7b and 2.8 schematically illustrate Class Evolution and Schema Evolution.

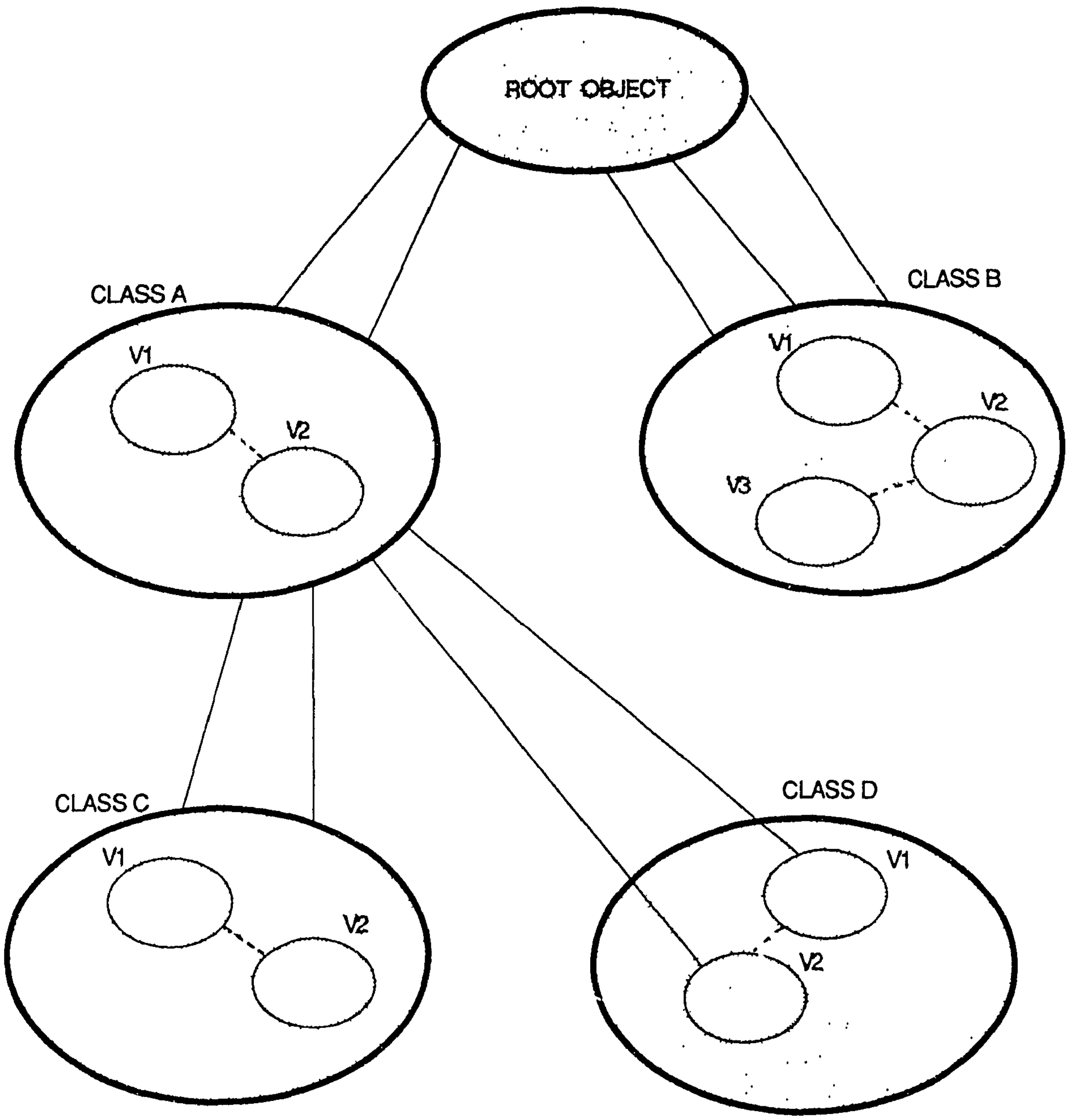


Figure 2.5: Database Schema - (CLASS VERSIONS)

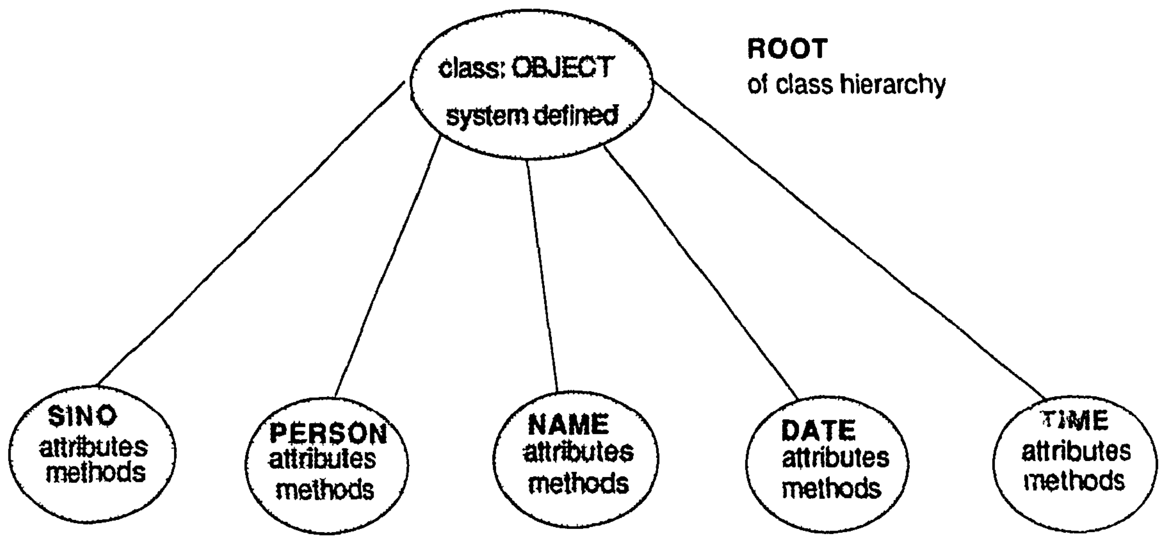


Figure 2.6: Schematic representation of the Database Schema

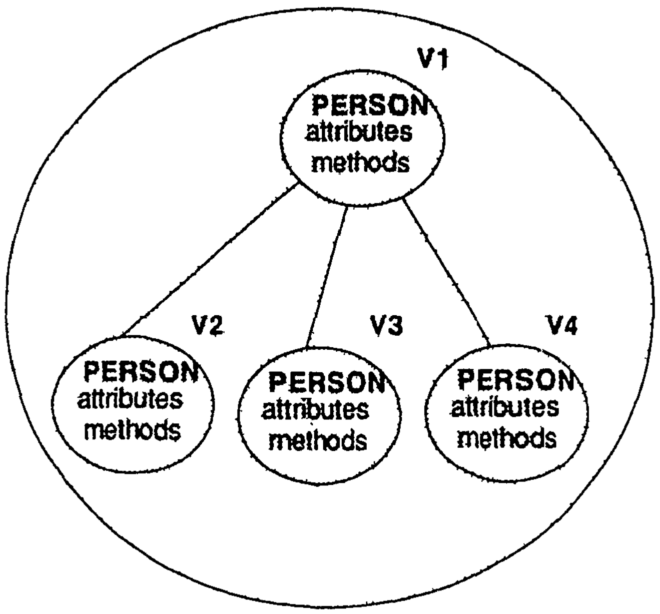


Figure 2.7a: CLASS EVOLUTION

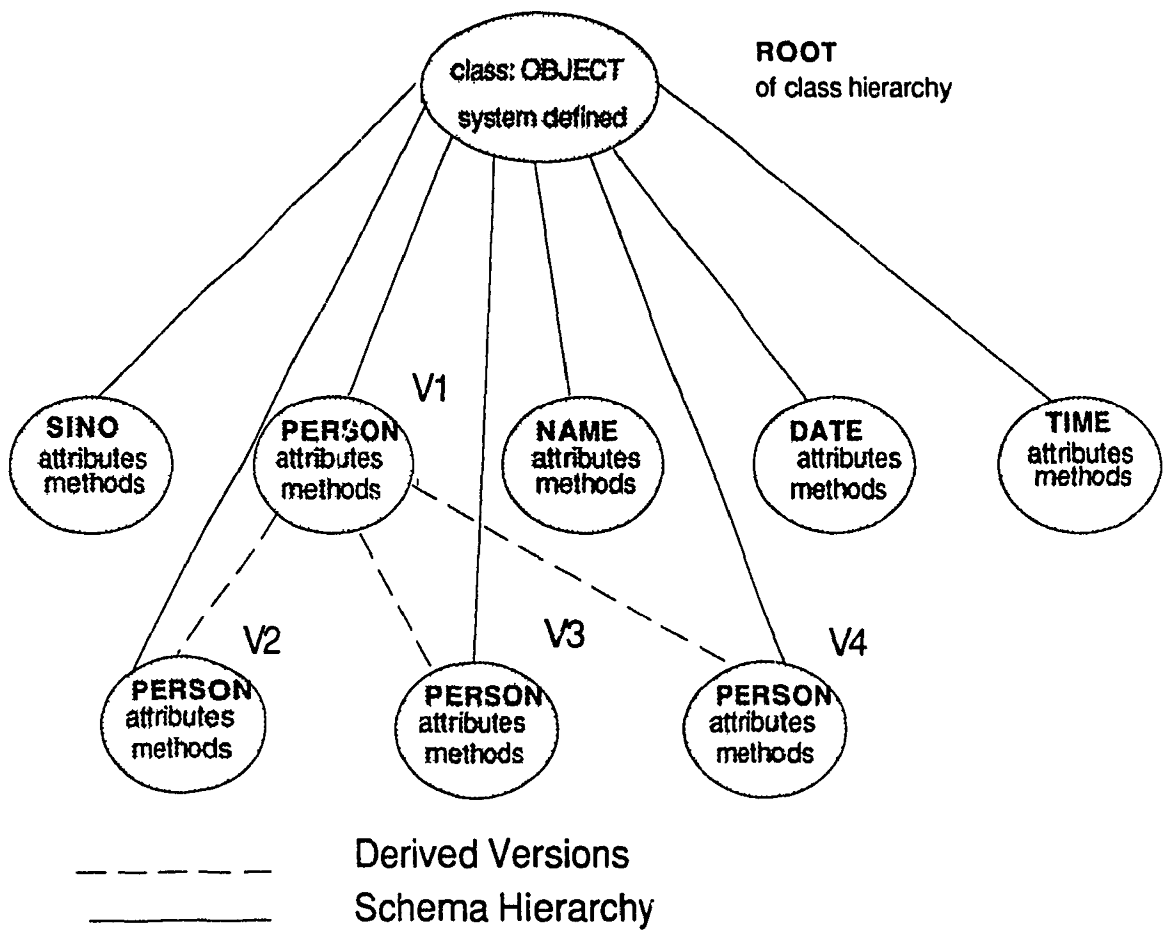


Figure 2.7 b: Schematic representation of the Database Schema

(CLASS EVOLUTION)

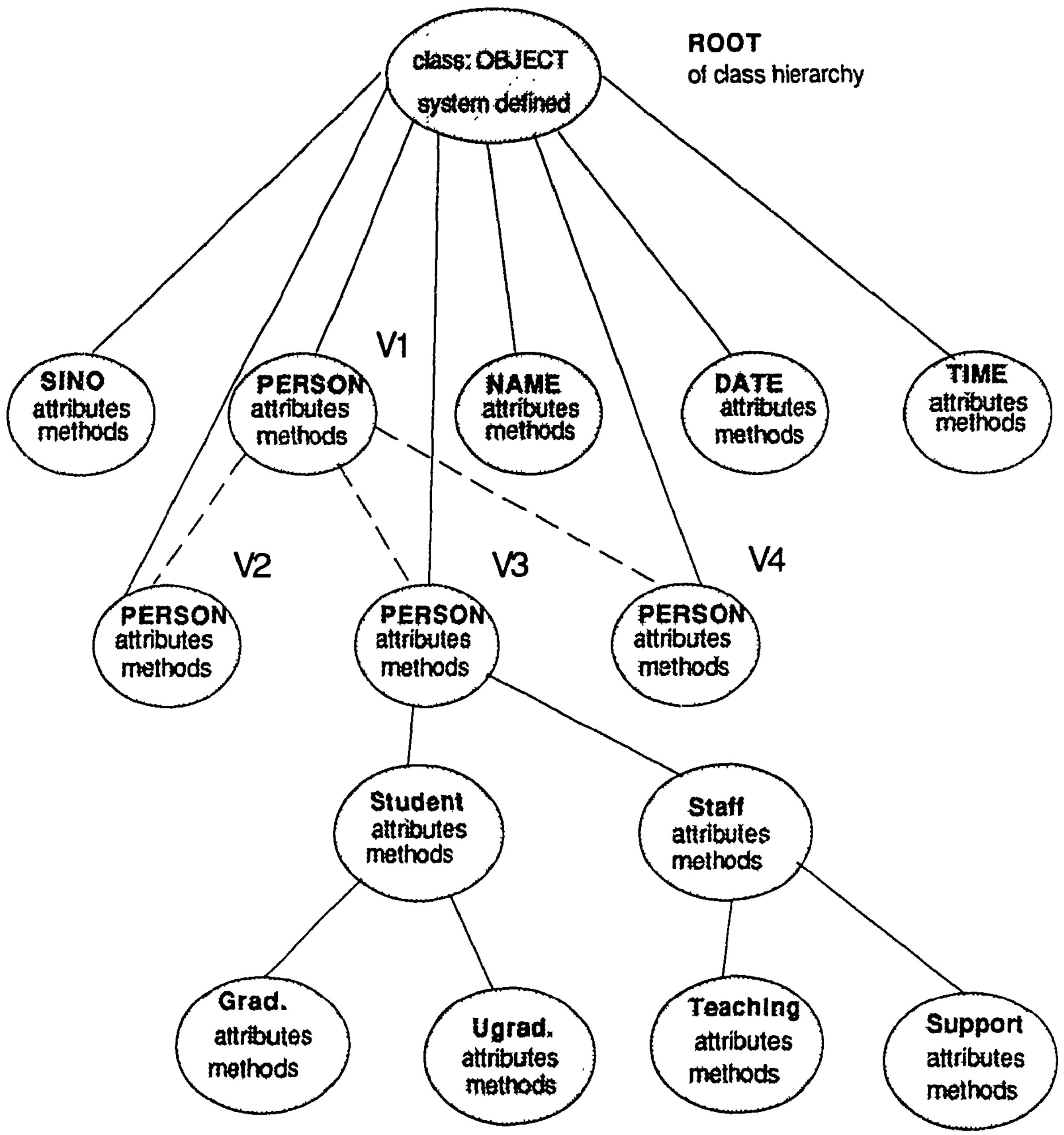


Figure 2.8: Schematic representation of the Database Schema
(SINGLE INHERITANCE via SPECIALIZATION)

A change to the definition of a class is captured as a new version of the class, without effecting the existing classes and their instances. The new version may retain the attributes of its parent deriver version or define them itself. For instance Version 2 of class A retains attribute a1 form its deriver version and defines a2, a4 and a5 itself, as shown in Figure 2.9. Each class version is an independent Object Type, the version hierarchy of classes is shown in Figure 2.10.

Schema Evolution by specialization enforces the (i) *Full inheritance constraint*, i.e., a class inherits all the attributes and methods from its superclass; (ii) the *Unique name constraint*, i.e., all attributes and methods of a class, whether defined or inherited, must have distinct names; and (iii) the *Domain Compatibility constraint*, i.e., the inherited attribute of the class must have the same domain as in the superclass or be a subclass of the domain in the superclass, as in ORION [6,7].

All class versions have the same superclass as that of their parent deriver version. All the versions retain the inherited attributes from the superclass, but may or may not retain the attributes defined in the parent deriver version. For example, in Figure 2.9, class A_v3, is the superclass of classes C_v1, C_v2 and C_v3. All the subclasses have the attributes inherited from the superclass in addition to the defined attributes. Inherited attribute v4 has its domain as D', which must be a subclass of D, the domain in the superclass. This is ensured by the system during the specification of the class. The class C_v3 may redefine the domain of attribute v4 to be D'', where D'' is a subclass of D'.

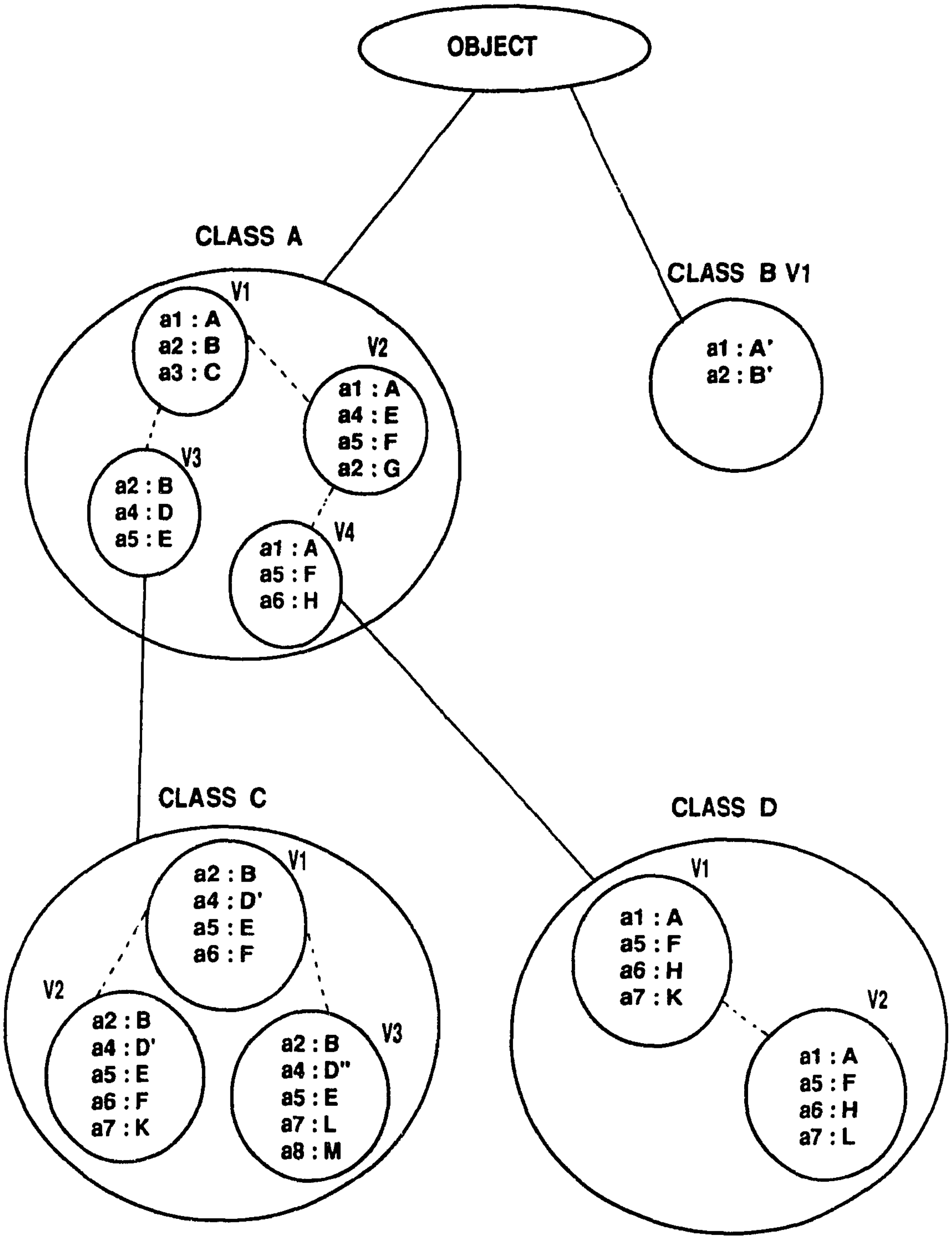


Figure 2.9 : Class Versions

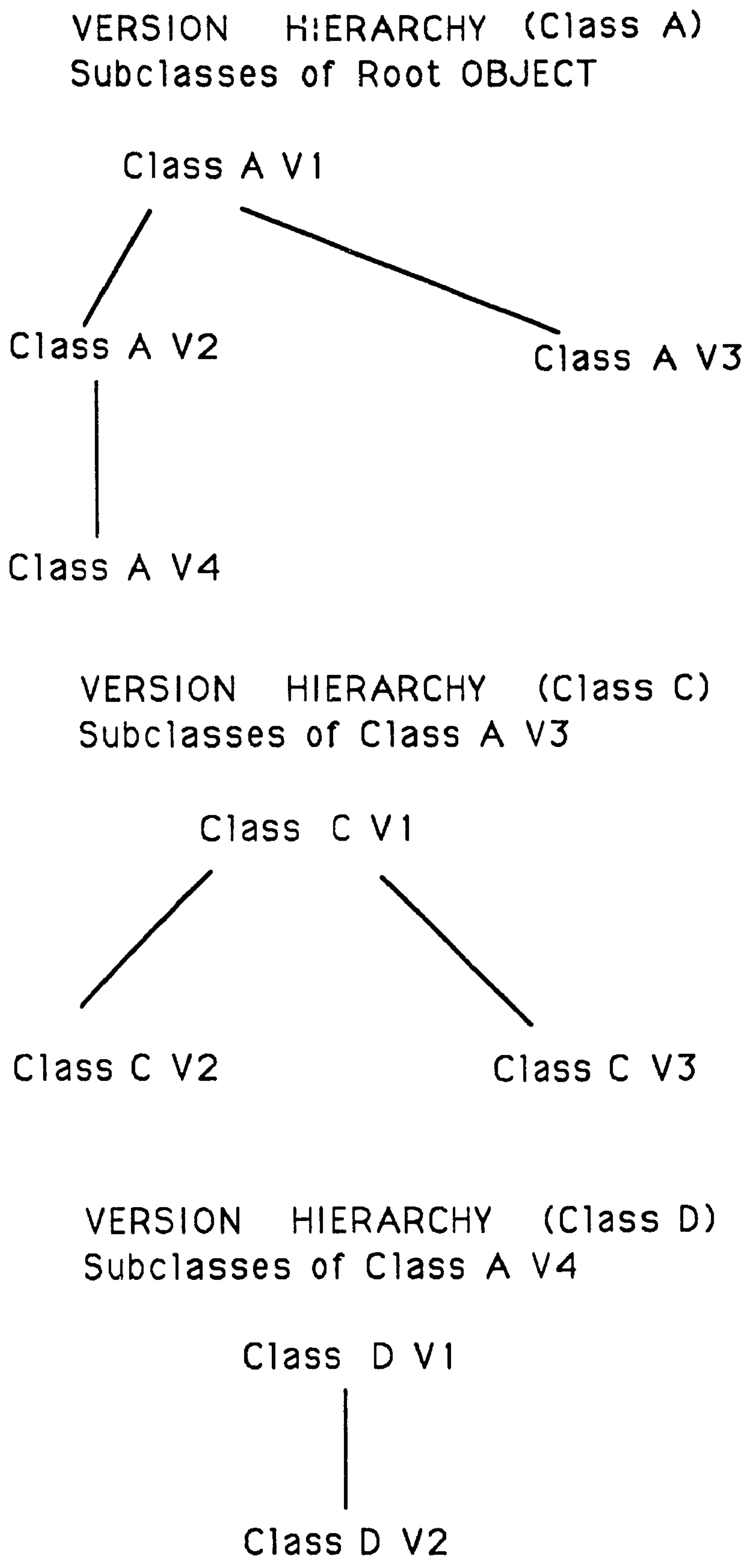


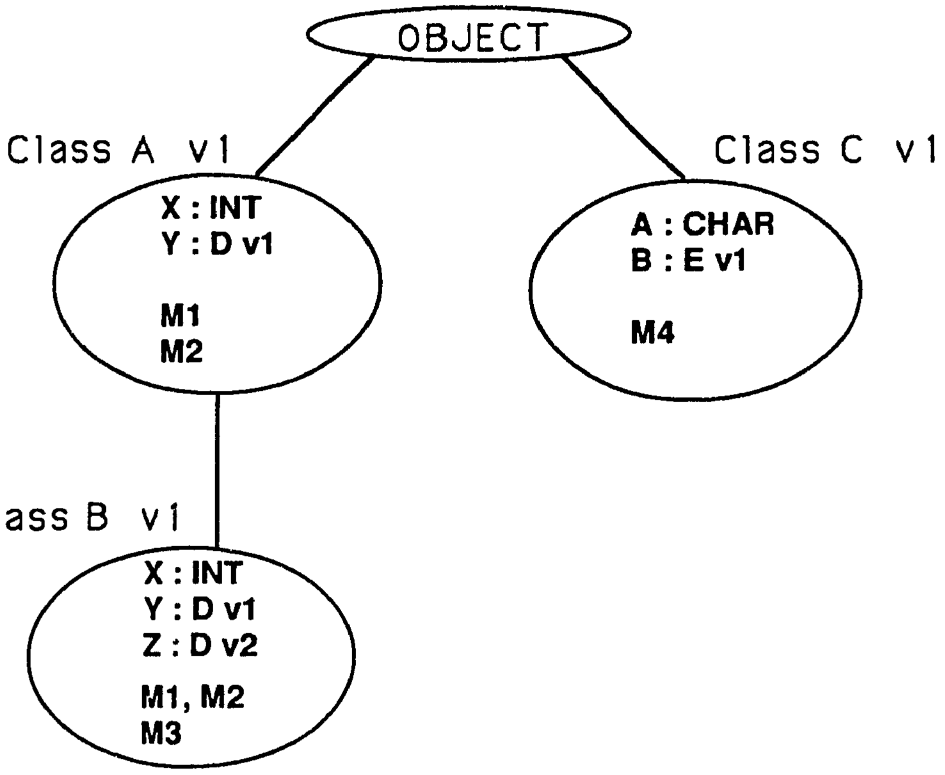
Figure 2.10 : Version Hierarchies

2.3.2. LINK CHANGES

Changes to an edge in the Class Hierarchy results in changing the Superclass of a class. In our model any change to a class will result in a new version. While making class S a superclass of class C, the following constraints have to be satisfied. A new version of C will be created called C'. C' will retain the attributes defined in C and will inherit attributes and methods from S. The methods defined in C will be retained in C' only if they do not have embedded references to the attributes and methods inherited by C from its superclass. In addition the effective_time of C' should be later than or the same as the effective_time of S and the components should also satisfy the temporal constraints. The changing of a superclass is envisioned by the user when he/she no longer wants to use class C, i.e., stop class C and change the design by changing the hierarchy.

Consider Figure 2.11 which shows the changes to an edge in a Class Hierarchy. Class B_v1 is a subclass of class A_v1. It inherits attribute X and Y and methods M1 and M2 from the superclass and defines attribute Z and method M3. Changing the superclass of B_v1 will define a new version B_v2 (it is not a derived version), which is a subclass of class C_v1. Class B_v2 inherits attributes A and B, and method M4 from its superclass C_v1, and retains the defined attribute Z. The retention of method M3 is determined by the system. It is retained if the method M3 has no embedded references to inherited attributes X, Y and inherited methods M1, M2. There is no semantic relationship between class B_v1 and class B_v2, as they are independent and not derived versions. The classes should satisfy the following temporal constraints, where the effective_time of class is denoted by ET(Class), and that of a method by ET(Method). The ET(class B_v2) $\geq$ ET(class C_v1), the ET(B_v1) $\leq$ ET(B_v2) and ET(D_v2)

$\leq$ ET(B_v2). The effective_time of inherited method is coerced to satisfy the condition ET(M4) $\geq$ ET(B_v2).



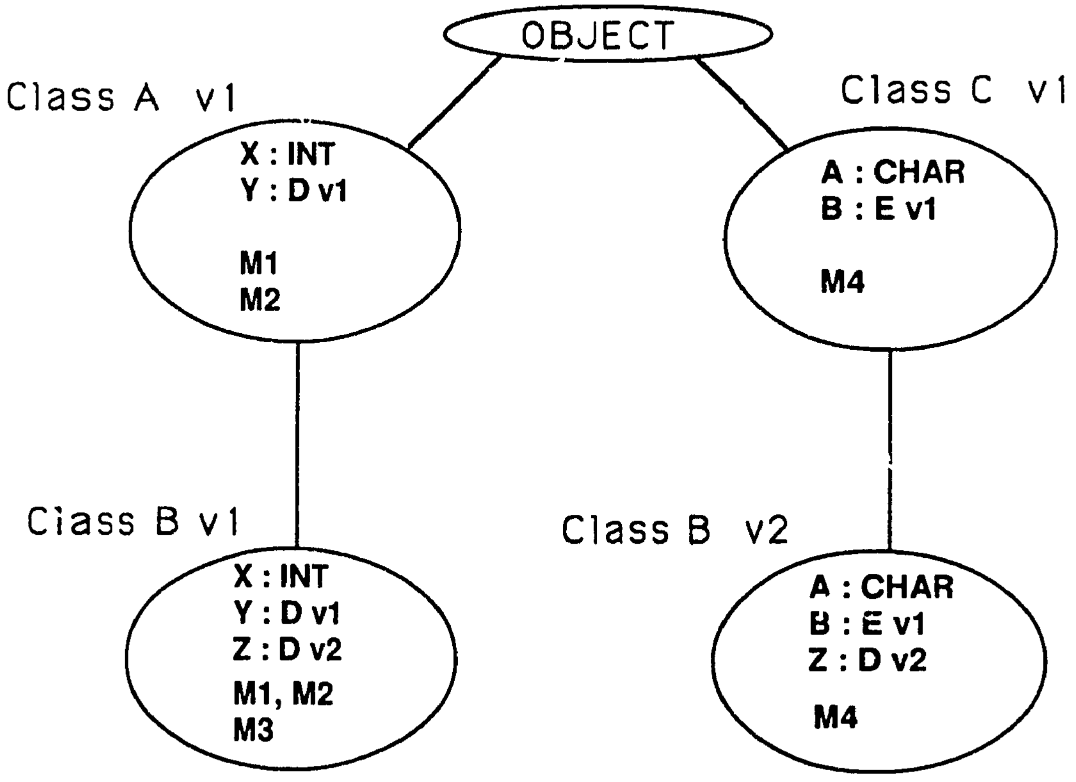


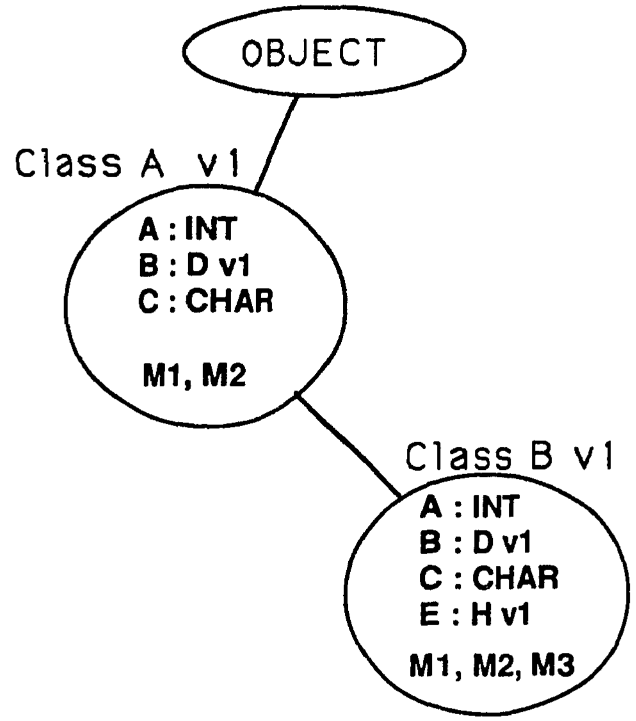
Figure 2.11 : Changes to an Edge (SCHEMA EVOLUTION)

2.3.3. ADDITION OF NEW CLASS

Addition of a new class in the class hierarchy can be done by defining a class as a subclass of the Root OBJECT, or by defining a new version (derived) of a class or by specialization of a class. A class can also be added to the class hierarchy by inserting a class in between the existing hierarchy, i.e., If class B is a subclass of class A, one can insert class C as a subclass of A and superclass of B.

Insertion of class C should satisfy the system constraints, i.e., temporal constraints, full inheritance constraint, unique name constraint and domain compatibility constraint. Class C must have all the attributes and methods defined in class A, in addition to its own defined attributes and methods. The defined attributes of C will be inherited by it's new subclass B' , if the defined attribute has the same name as that of an existing attribute in it's subclass, the user should ensure that they have the same semantic interpretation, either they may have the same domain, or the domain of the defined attribute in class C is superclass of the domain of the attribute in the existing subclass, which conforms to the domain compatibility constraint. Similarly, if class C defines a method which exists in its subclass, the method in the subclass is treated as it's redefinition in accordance to the Object-Oriented design principle.

Figure 2.12 shows schema evolution by inserting the class in the Class hierarchy. Initially the Schema consists of class B_v1 and class A_v1, where class B_v1 is a subclass of class A_v1. On inserting a class C_v1 as a subclass of A_v1 and superclass of B_v1, a new class B'_v1 is created, which is an additive clone of B_v1, a new class independent of B_v1. Class C_v1 is a subclass of A_v1 and superclass of B'_v1. The schema contains both the views of the system.



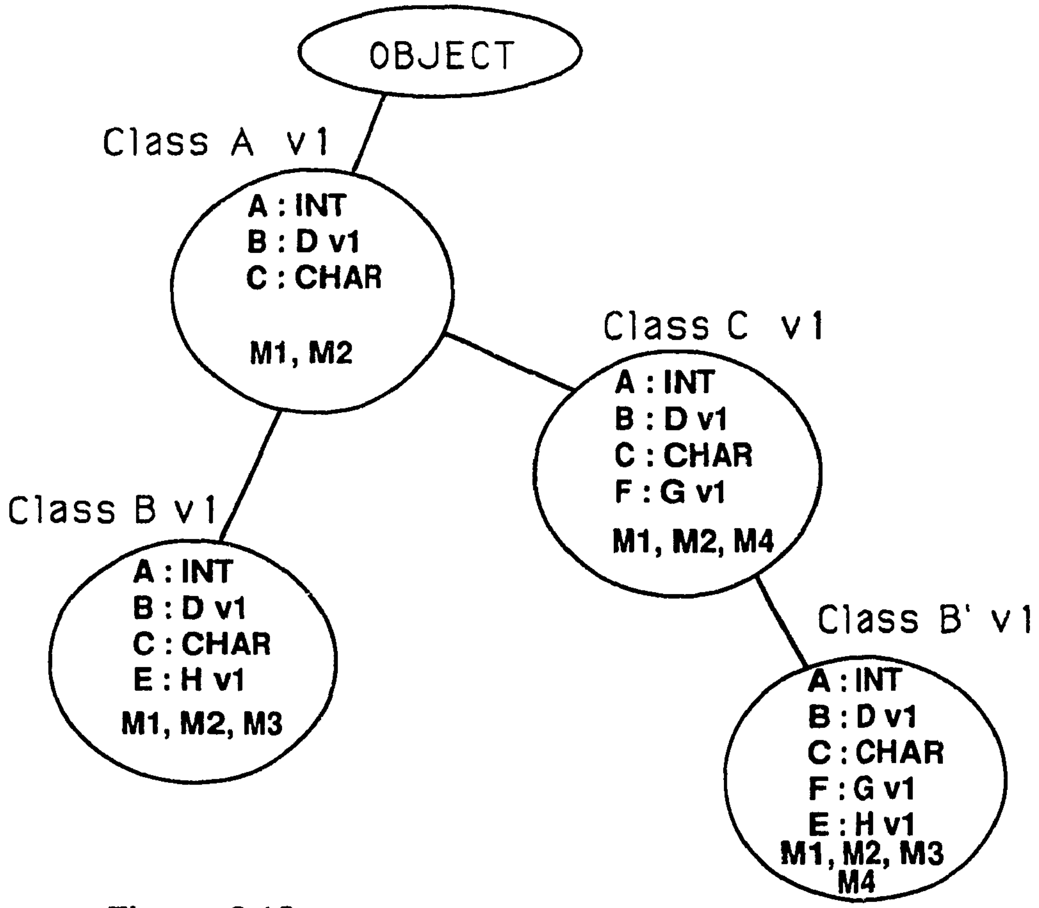


Figure 2.12 : Inserting Class in the Hierarchy

2.4. METHODS - MESSAGES

Method is the implementation of the objects behaviour, whereas *Message* is the signature describing the operation which will cause the invocation of methods. Each class defines its interface by a set of strongly typed operation signatures, i.e. messages , during method specification.

The methods capture the behaviour of the objects in a class. Methods may enforce ***integrity constraints***, which are statements that must always be true for objects in the database; ***referential integrity***, which asserts that a reference by one object indeed leads to another object; and simulate ***triggers*** to help in constraint enforcement. In general the systems should have a constraint manager, which ensures that operations on objects maintain the semantics associated with the schema. This decouples the necessity of encoding the semantic constraints within method implementation.

To enforce Encapsulation, which states that the objects are accessible through well-defined interface (operation signatures of a class), only the methods implementing operations for objects can access the representation used to store the state of the object, thus allowing one to change the representation without disturbing the rest of the system. In case of specialization, where new classes are specified by extending the existing specification, the methods for the new classes should not directly access the underlying representation of the superclass, but should access any attributes of the superclass only through methods inherited from the superclass. This approach should be adopted by the implementor of methods to honor the information-hiding principle of Abstract Data Types used in Object-Oriented Design.

Full inheritance of methods in subclasses with redefinition is permitted, i.e.

methods with the same signature as in superclass with different implementation can be specified. Moreover, all the inherited methods will conform their effective_time to be the same as that of the subclass's effective_time or later, as a method's effective_time should be same or later than its class. For example, consider class S with effective_time T, having methods M1, M2 and M3 with effective_times T_1, T_2, and T_3 respectively, which are $\geq$ T. In addition assume that $T_1 = T$, $T_2 > T$ and $T_3 > T$. Also there is a class C, which is a subclass of S, having effective_time T', $T' > T$. Assuming that $T_1 < T'$ and $T_2 < T'$, and $T_3 > T'$. When class C inherits methods M1, M2 and M3, the effective times of M1 and M2 for class C are set to T' and that of M3 remains T_3, to satisfy the constraint that the effective times of the methods of a class are same or later than that of the class.

The methods of a class are usually implemented using inherited methods of the class and can also use methods of its component objects, which constitute PART_OF relationship among classes. In Eiffel terminology a class is said to be a client of its component class. The implementor should ensure that all the invoked methods (inherited/component's) have been specified. The methods are categorized as external or internal. An *External Method* is the one which can be invoked by the user and an *Internal Method* is the one which can only be invoked by the system, e.g. indirectly invoked by a method, but not by the user. For example, consider class PERSON_v1, with attribute name of type NAME_v1. The *component class* NAME_v1 has methods specified *getfirstname* and *getlastname*, which are categorized as internal methods and the class PERSON_v1 has the method specified *getname* which is an external method invoked by the user. The method getname in turn calls methods getfirstname and getlastname defined on the component class to find the name of a given person object.

2.5. OBJECT INSTANCE - INSTANCE EVOLUTION

Object instance (object) of a class is a mapping $f : A \rightarrow V$, where A is the set of attributes of the class, and V is the set of values, such that f maps an attribute A_i to a value corresponding to the type of A_i. For primitive types, the value is represented directly by a value in the corresponding domain (e.g., integer, character). For other types a value is actually an object instance on the type class, represented by an *object identifier.* Each object instance has a unique identifier (oid) assigned by the system internally.

Each instance is associated with it's valid timestamp, which is same as that of its class or later than its class. As operations (updates) are performed on the instance, it's state changes. This results in the evolution of the instance, which we consider as the creation of a new version of that instance. Each update operation is associated with it's *effective time*, the time when the update should take effect. The default situation is where the effective time is taken to be the same as the *commit time*, that is the time when the operation (method/action) commits. The use of the effective time of the operation allows one to model retroactive and post_dated updates. Updates with an effective time earlier than their commit time are *retroactive*, while those with effective time later then their commit time are *post_dated.* The validity of the instance is implicit until the existence of its next version. If the object is the only instance or the last version (in order of valid time), its validity is from its valid time onwards, i.e. their is no limit on upperbound (in general), until another version of the instance is created.

The system allows the user to stop a class, i.e. the validity of the class is upto its *stop time* only, and the system keeps track of all the stopped classes and their stop times. Once the class has been stopped no more update operations can be performed on the instances of the class and neither can new instances be created, but one can query the state of the existing instances of the stopped class.

Example 1: Consider a class Employee_v1, with effective time Jan. 1, 1990. Let John be an instance of the class Employee_v1, with effective time and commit time being Ja i. 25, 1990, (default situation), having a salary of 32,000 and holding the position of Systems Analyst in the Electrical Engineering department. Later on John is transferred to Computer Science department by an update operation having effective time and commit time Feb. 28, 1990. The update operation will result in creation of another version of John, the validity of the first instance version is from Jan. 25th to Feb. 27th 1990 and the validity of the second instance version is from Feb. 28th 1990 onward. The system records the instance evolution by keeping track of the operations performed on objects. □

The effects of retroactive updates need to be propagated to the necessary objects. The system should identify the affected objects, and the user will have to take necessary actions. Automatic propagation of the effects of retroactive updates is possible only in simple cases as discussed in Example 2. In general corrective action should be taken by the user.

Example 2: Assume that a retroactive salary increment of 5%, effective Feb. 15, 1990 is awarded to John on March 10th 1990 (commit time). The effect of this retroactive update should be reflected in the queries corresponding to John's salary depending on the observation time. John's salary on Feb. 15 as observed on March 9th is 32,000 while as observed on March 15th is 33,600, reflecting the retroactive update. In such simple cases the system may be able to propagate the effects of retroactive updates, but in cases of design environment where retroactive updates represent corrections to past errors, these may not be automatically propagated to all the affected versions. In such situations, the system identifies the invalidated objects and the user has to take the necessary corrective action as deemed by the semantics of the application. □

2.5.1. MULTITYPING

Multityping is a feature introduced in this model, it refers to the ability of an object to have more than one type, i.e., an object can be an instance of several classes. It allows one to capture multiple behaviours exhibited by the object depending upon the role associated with the object. For example, an object X may be a member of class employee, class musician and the class hockey_player. The object X will exhibit different behaviours, i.e., respond to different messages, or even respond differently to the same message, according to the role exhibited by the object X at a given instance of time. This also reflects objects evolution from one type to another.

Example 3: Consider the class Student_v1 having attributes name, idno, department, and university with effective time Jan. 15, 1990; and class Musician_v1 having attributes name, instrument and organization with effective time Mar. 20, 1990. An instance Bob of class student_v1 with effective time Jan. 20, 1990, studying at Concordia University is multityped as an instance of class Musician_v1 with effective time Mar. 25, 1990, undergoing by the pseudo_name Jack Brown, who plays the instrument piano with the organization Montreal Symphony Orchestra. The response to the message *getname* on class student_v1 is the name of the object, while the response by class musician_v1 is the pseudo_name of the object. The receiver class of the object identifies the role being played by the instance. Moreover, prior to Mar. 25, 1990 Bob exists only as a student but after Mar. 25, 1990, he exists as a student as well as a musician. □

2.5.2. INSTANCE MIGRATION

In Object-Oriented Model, all instances of a subclass are also instances of the superclass but not vice-versa. In our model an instance of a superclass can evolve and become a member of the subclass, a new class, which is another form of object evolution. The migration of instances from superclass to subclass is at the discretion of the user. In general an instance can migrate to any class.

Both multityping and instance migration are evolution of the association between an object and the class, which attempts to model the real world naturally and precisely, where the association between an object and the class is a function of time.

Each class has *membership constraints* associated with it, which are satisfied by all the instances of the class. In general a membership constraint may be a First order logic expression. A membership constraint enumerates the acceptable state of the instances belonging to the class. A special form of a constraint is a conjunct of the acceptable state of the instance, i.e.,

$(A_i = x \ .or. \ A_i = y \ ) \ .and. \ (A_j \geq z \ .or. \ A_j \leq z' \ )$

where A_i and A_j are attributes of an object and x, y, z, and z' are the values of the attributes. Membership constraints are application dependent and during schema evolution the membership constraints of the subclass subsumes that of its superclass, i.e., the membership constraints of the subclass are more restrictive than that of its superclass. Constraints of a subclass logically imply the constraints of the superclass.

Membership constraints enforce data integrity to be satisfied by the objects. Automating the enforcement of general constraints by the system is an area of future research concerning constraint management. Constraints such as the salary of the manager is greater than that of his employee would require searching the database on every update effecting salary. Usually integrity constraints are coded in appropriate update methods, to ensure data consistency.

2.6. HISTORY

In Object-oriented system, the basic operation mechanism is message passing. Where a message is an expression of the form "<receiver> <message-selector> [<arguments>]", where <receiver> is the object to which the message is sent, <message-selector> is a name of the type of interaction the sender desires with the receiver, <arguments> are the values for the formal arguments. A message requests an operation on the part of the receiver. The selector of message determines which of the receiver's operations will be invoked. The number of the arguments and their types along with the receivers type participate in deciding the method to be applied. A method may return an object, to which another message can be sent. Each message can be considered as a transaction. All operations are performed through message passing. To keep track of the systems operations and evolution, a log of these operations are maintained, which constitute the History of the System. This history is referred to answer the queries. To facilitate queries, checkpoints of objects are maintained which in conjunction with the history are used to materialize the current state of the objects and answer queries. Transaction, Checkpoints and History Log are discussed in section 2.7, 2.8 and 2.9 respectively.

2.7. OPERATIONS - QUERIES AND UPDATES

Queries are methods which retrieve information about the state of the objects, whereas updates are methods which modify the state of the objects. Due to the introduction of the notion of time in the system, the actions (events) are categorized as either retroactive, normal or post-dated, these are schematically shown in Figure 2.13 where the two orthogonal time notions, Commit time and Effective time are plotted on X-axis and Y-axis, and an event associated with both of these times is shown. Also because of the dynamic behaviour of the objects, their state is dynamic. So at different instants of time, different answers may be given to the same query. Likewise Queries have two temporal parameters associated with them, the *effective_time* and the *observation_time*, whereas the updates have *effective_time* and *commit_time* as their parameters.

The state of the object is a function of time, i.e., it may be observed to be different with respect to different effective times. The *observation_time* is the instant of time specified in the query at which the state of the object will be retrieved with respect to the effective time specified in the query. The response to a query will take into account the effect of all the actions (updates) with commit_time upto the observation time of the query and effective_time upto the effective time of the query. Any retroactive action with commit time later than the observation time of the query and effective time earlier than that of the query will not be reflected in the response to the query. This is schematically illustrated for queries Q1, Q2, Q3 and Q4 in Figures 2.14-a, 2.14-b, 2.14-c, 2.14-d, where events under the scope of each query are indicated within a box.

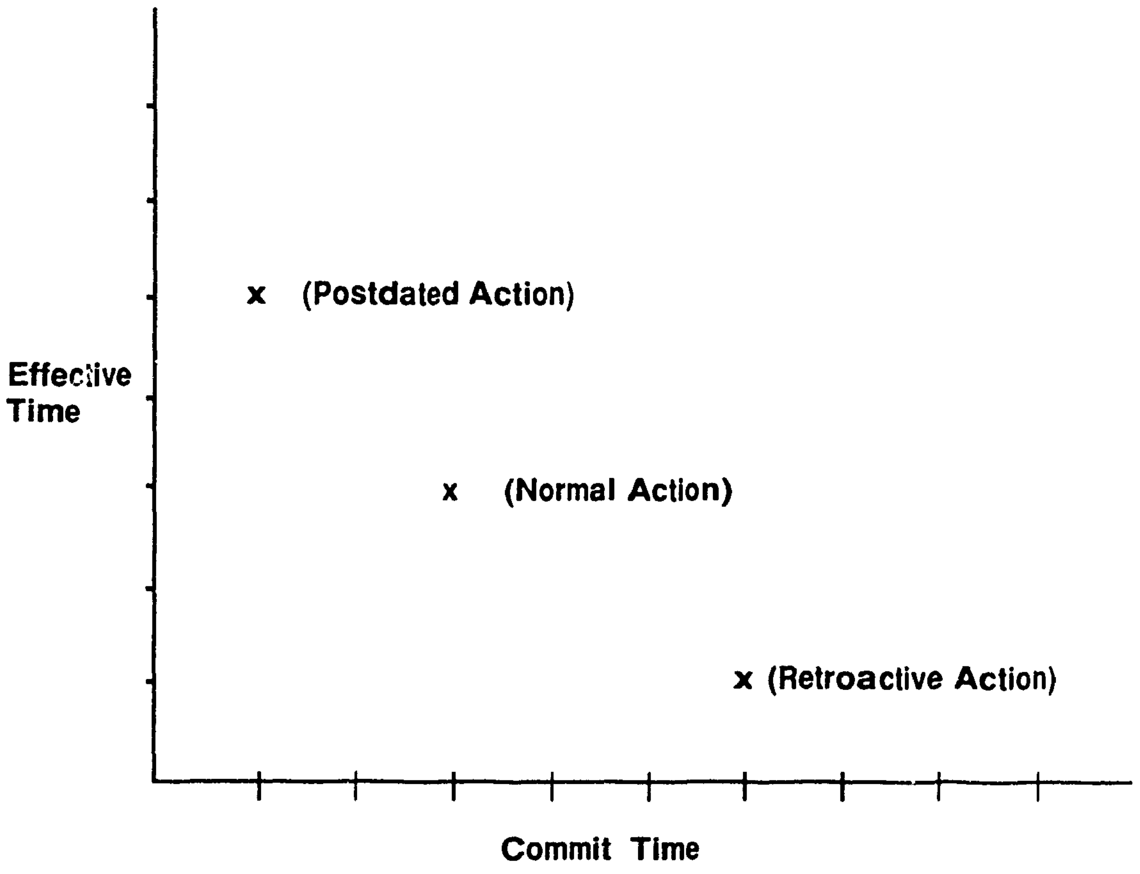


Figure 2.13 : Events Plotted against Two independent Time axis

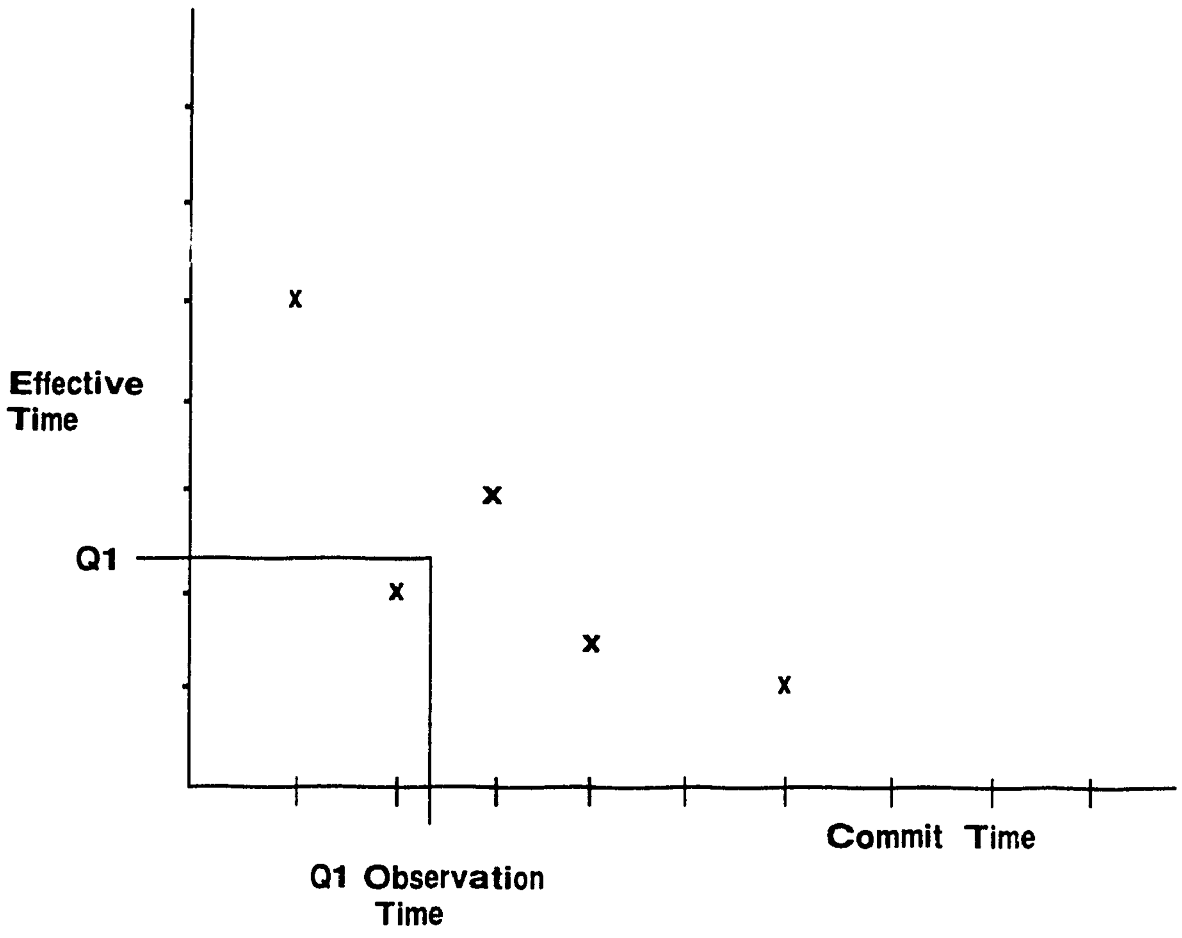


Figure 2.14 a : Update Events on an object Plotted against Two independent Time axis

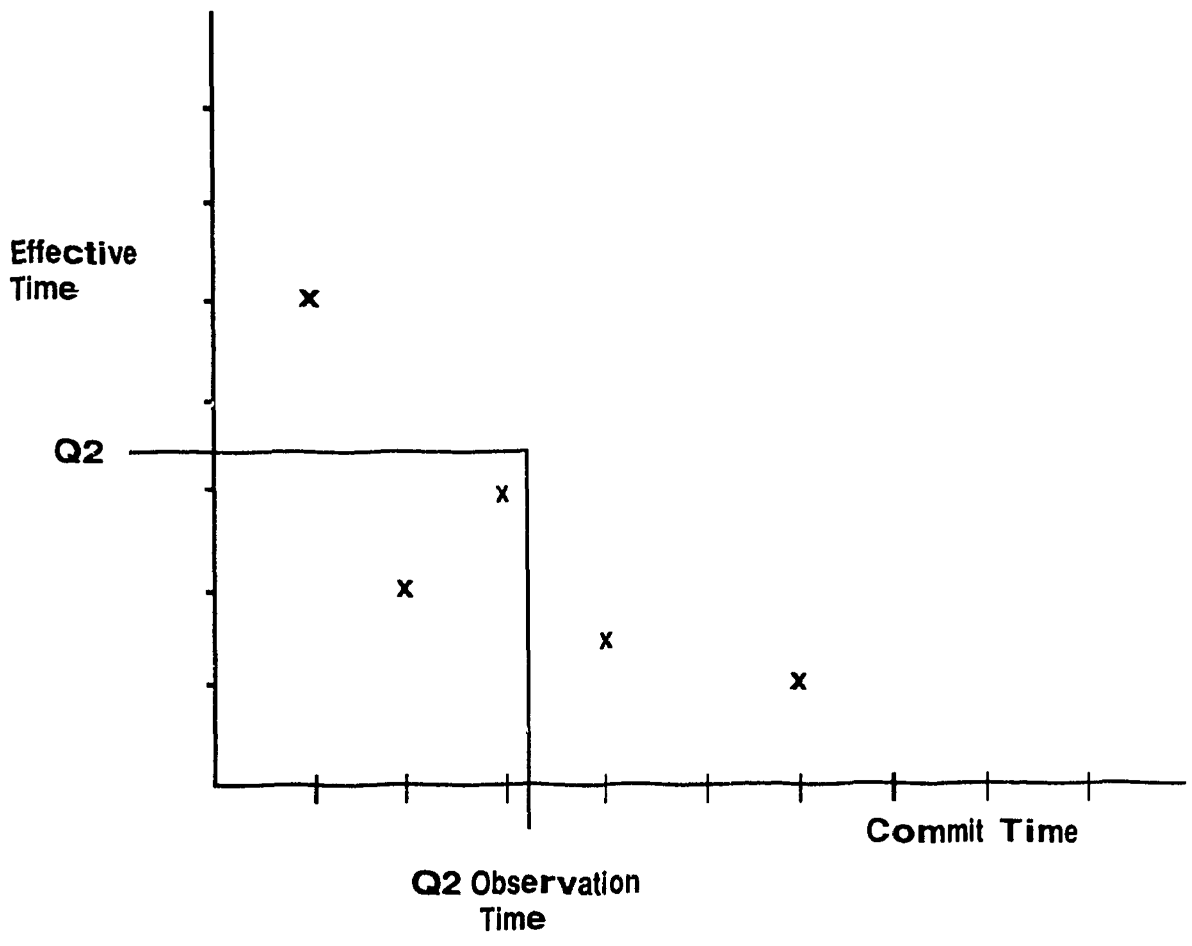


Figure 2.14 b : Update Events on an object Plotted against Two independent Time axis

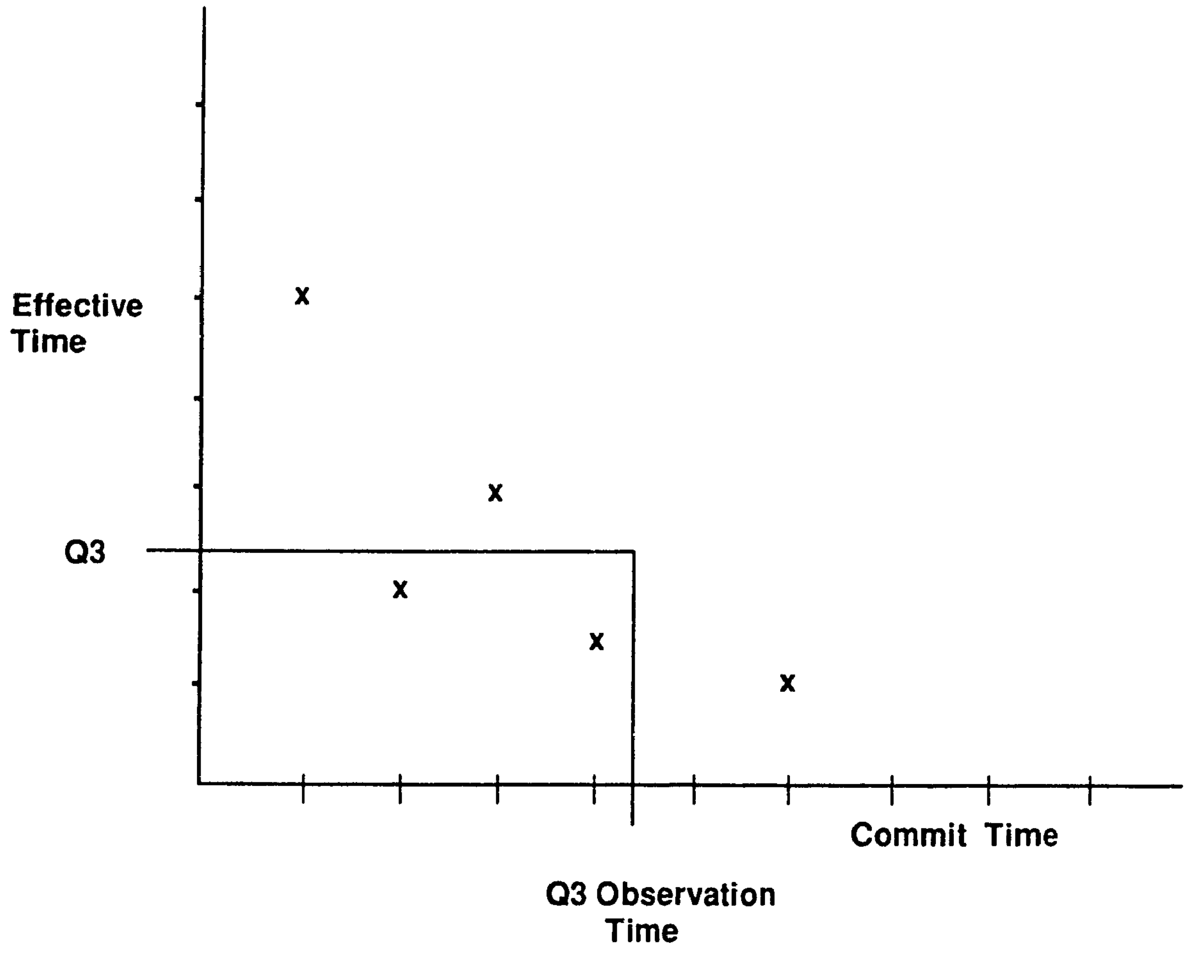


Figure 2.14 c : Update Events on an object Plotted against Two independent Time axis

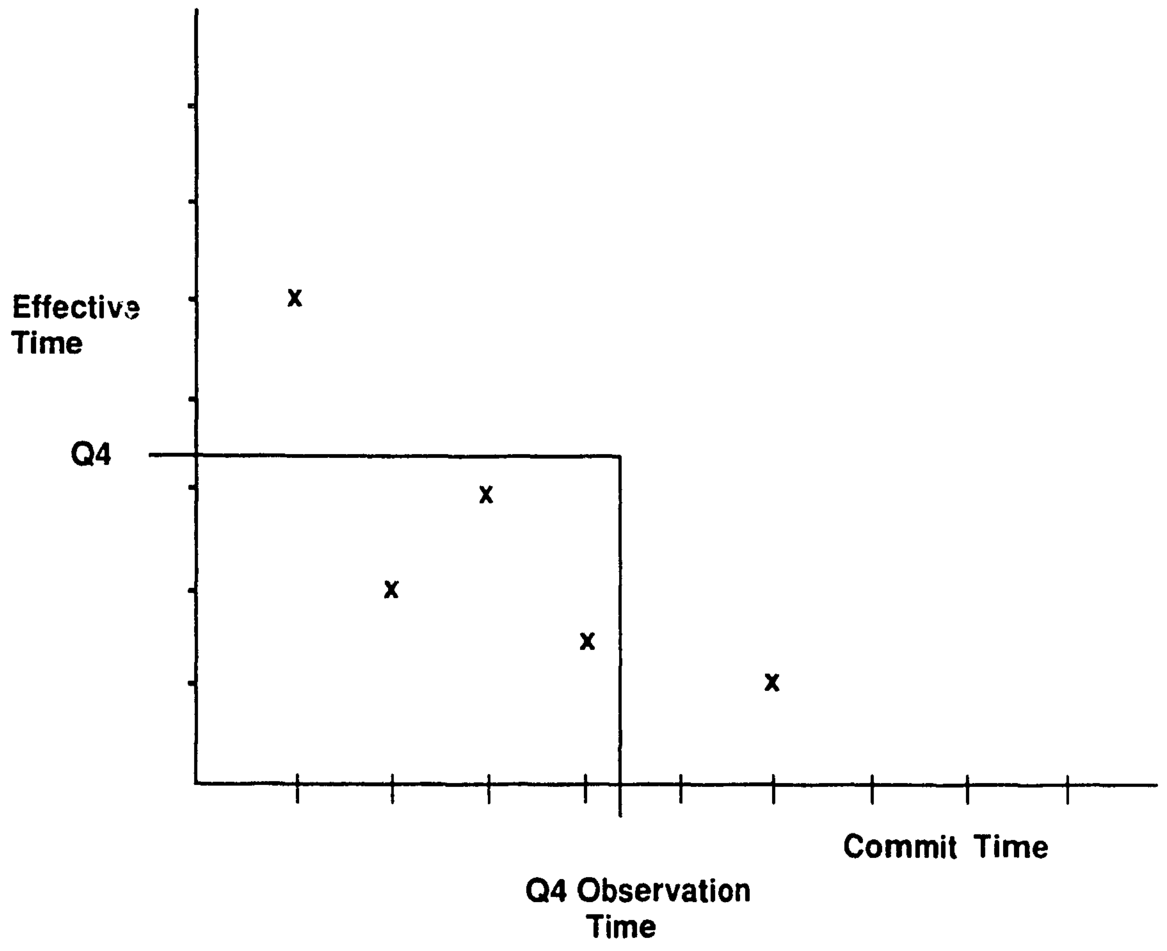


Figure 2.14 d : Update Events on an object Plotted against Two independent Time axis

The response to the query requires identification of the events within the scope of the query in the history and executing these events in the order of their effective time, to recreate the effective state of the object at the effective time of the query as viewed at the observation time. To facilitate temporal query processing, the current state of the objects are materialized at selected time instances, which serve as checkpoints. The checkpoints along with the history log are used to answer the queries (Discussed in section 2.8).

The following examples illustrate temporal query processing which involve corrections as well as proactive and retroactive updates.

Example 4: A transaction T1 is an event which hires John with a salary of 45,000 on March 1, 1989, and the information is entered into the database also on March 1, 1989, through transaction T1. It was subsequently discovered that John was hired with a salary of 40,000 and this information was entered into the database by another transaction T2 on March 15, 1989. As a result of ongoing negotiations with the administration, an agreement was reached that all the employees will get a 5% increment effective March 1, 1989. This information was entered into the database on June 15, 1989. The materialized database has the following information about John:

Transaction			Effective Time	Commit Time
T1	John	$45,000	March 1, 1989	March 1, 1989
T2	John	$40,000	March 1, 1989	March 15, 1989
T3	John	$42,000	March 1, 1989	June 15, 1989

Transaction T2 is a retroactive correction of the salary of the object John. the validity of any correction must not be prior to the creation of the object. In this case the validity of both the creation of the object and the retroactive update is same. The validity of the update can be same as that of the object or later but not earlier than the object's creation. The response to the query "What was John's salary on March 10, 1989 as observed on March 13,1989?", will be $45,000. While the answer to the query "What was John's salary on March 10, 1989 as observed on March 17, 1989?", will be $40,000. The response to the second query will require processing transactions T1, and T2 with commit time less than the observation time and effective time less than that of the query, in the order of their commit times. Transaction T3 is not within the scope of the query as its commit time is later than that of the observation time. □

Example 5: A transaction T4 is an event which hires David with a salary of 45,000 as a Systems Analyst on Feb. 1, 1989, and the information is entered into the database on Feb. 6, 1989, through transaction T4. It was subsequently discovered that David was hired as a Systems Analyst on Feb. 10 1989 and this information was entered into the database by another transaction T5 on Feb.15, 1989. Transaction T5 revises the effect of transaction T4 completely. In order for the system to reason and respond to queries, a complementary Transaction T5′ which invalidates T4 is entered into the database prior to T5. It has the effect of deleting the existence of David created by T4. The database has the following information:

Transaction		Effective Time	Commit Time
T4	David, Systems Analyst	Feb. 1, 1989	Feb. 6, 1989
T5′	Invalidate T4		Feb.15, 1989
T5	David, Systems Analyst	Feb.10, 1989	Feb.15, 1989

The answer to the query "What was David's position on Feb. 8, 1989 as observed on Feb. 12, 1989?", will be Systems analyst but the response to the query "What was David's position on Feb. 8, 1989 as observed on Feb. 20, 1989?", will be that David is not a valid object on Feb. 8, 1989. The Transactions enable the system to reason temporal belief periods. It is the users responsibility to ensure that the transactions emulate the semantics of the application. In this example it was believed that David was a Systems Analyst from Feb. 1 onwards which was revised by transactions T5′, and T5. □

Example 6: Susan is hired as a Supervisor in June 1, 1988, and the information is entered into the database through transaction T6 on June 1, 1988. A Decision is taken in August 1, 1988 to promote Susan to Assistant Manager for a period of 2 years effective October 1, 1988 and then to promote her to the Managers position at the end of that period. These proactive updates are recorded in the database through transactions T7 and T8. Susan is found to be extremely brilliant and therefore promoted to Managers position in December 1, 1988 with effect from July 1, 1988 through transaction T9. Transaction T9. is an retroactive update affecting earlier proactive updates, some of which have already taken effect. For the system to determine the belief periods and the relationships that hold, compensatory transactions T7′, and T8′ , which invalidate transactions T7 and T8 respectively, have to be entered into the database. The database will have the following information:

Transaction			Effective Time	Commit Time
T6	Susan	Supervisor	June 1, 1988	June 1, 1988
T7	Susan	Asst. Manager	Oct. 1, 1988	Aug. 1, 1988

T8	Susan	Manager	Oct. 1, 1990	Aug. 1, 1988
T7′	Invalidate T7			Dec. 1, 1988
T8′	Invalidate T8			Dec. 1, 1988
T9	Susan	Manager	July 1,1988	Dec. 1, 1988

The response to queries "What is Susan's position on Aug. 1, 1988 as observed on Aug. 1, 1988?", is Supervisor, and the response to queries "What is Susans position on Nov. 1, 1988 as observed on Nov. 1, 1988?", is Assistant Manager, while the response to queries "What is Susan's position on Dec. 1, 1988 as observed on Dec. 1, 1988?", is Manager. The response to the third query takes into account the affect of transaction T7′ , which invalidates the affect of T7. □

An external method invocation is treated as a *Transaction*, it comprises of all the method invocations generated by the initial invocation and subsequent invocations. Thus the transaction is a nested transaction comprising of subtransactions. The history keeps track of the transaction invocations. For example, consider an invocation of method M1, which in turn invokes M2 and M3. The method M2 in turn invokes calls to methods M4 and M5 which do not invoke any more calls, whereas method M3 invokes method M6 which calls method M7. The method M7 does not invoke any more methods. Thus a transaction T1 consists of method invocations M1, M2, M3, M4, M5, M6, and M7. This is illustrated in the figure 2.15.

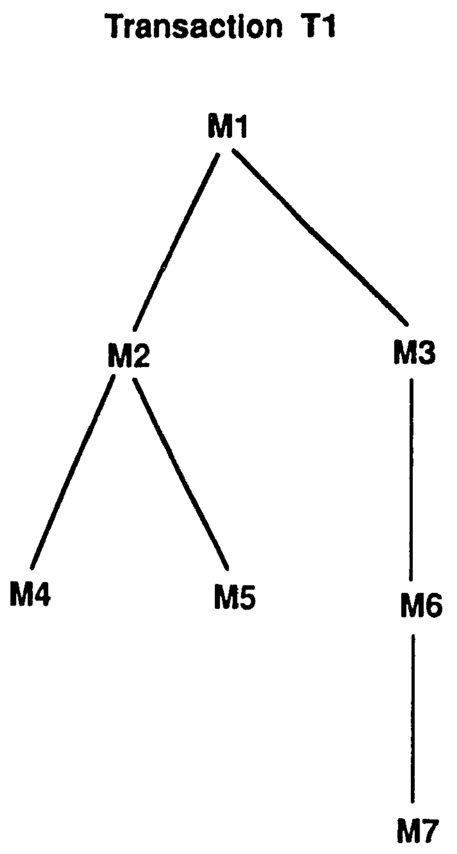


Figure 2.15 : Nested Transaction

2.8. CHECKPOINTS

To facilitate query processing, the system materializes the state of the objects at selected instances of time, which serve as checkpoints for the object. One approach towards generating checkpoints for an object is to specify at the time of its creation that after how many versions should it be materialized. Each instance is materialized on its creation, i.e., the first version of instance, and the trigger associated with the object will materialize every n'th version of the instance, specified by the user.

Suppose an object has checkpoints Ost_i, Ost_j, Ost_k, etc, representing the effective states of the object at time instances t_i, t_j, and t_k, where $i < j < k$. Consider a query about the state of the object with effective time e_q and observation time o_q, where $t_j < e_q < t_k$, and $t_j < o_q < t_k$. To process the above query the system makes use of the checkpoint Ost_j as the initial state of the object and consults the history log for events (actions) with effective time greater than t_j and less than equal to e_q with commit time upto the observation time of the query o_q, to determine the effective state of the object at the observation time of the query.

Retroactive updates may invalidate some of the checkpoints generated. Query processing will account for invalidated checkpoints and use the most recent unaffected checkpoint as its initial base. Consider a retroactive event $R_{i'j'}$, where i' is the effective time and j' is the commit time of the retroactive event and $j' > j$, $i' > i$. A Query with observation time $o_q < j'$ will use Ost_j as the initial base where as the query with observation time $o_q > j'$ will use the most recent unaffected checkpoint Ost_i, as its base. This is schematically illustrated in figure 2.16.

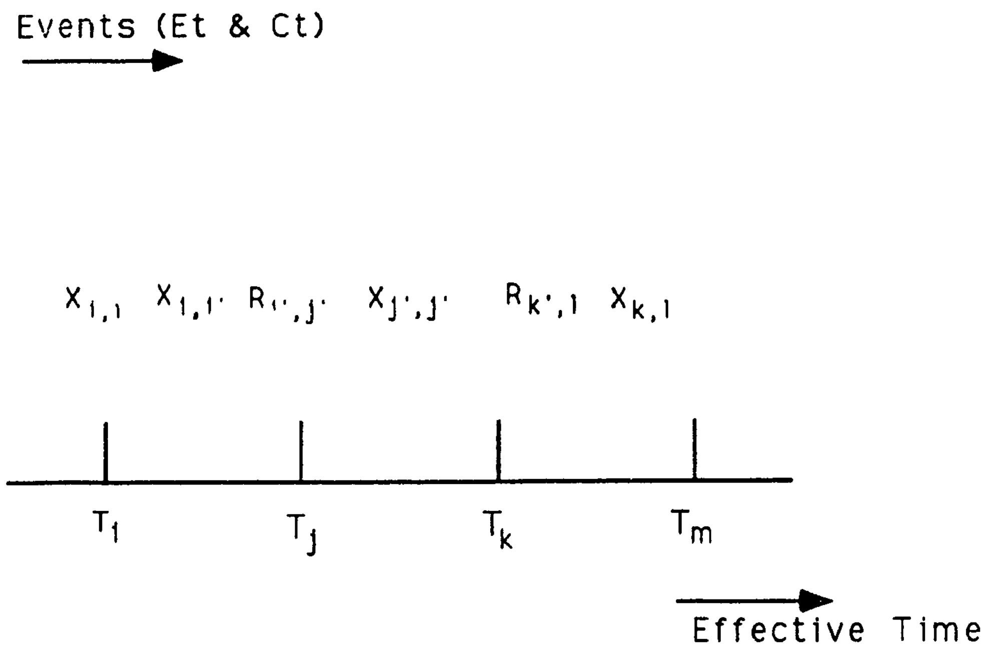


Figure 2.16 : CHECKPOINTS OF AN OBJECT

2.9. HISTORY LOG

The system keeps track of all the operations performed on the objects. This history of actions is conceptually stored as log entries. Each entry includes the following information:

Transaction id. (Tr. Id.)

Sequence no. (Seq. No.)

Invoker id. (Sender)

Receiver id. (Recv. Id.)

Method invoked.

Commit time.

Effective time.

Parameters (Arguments)

Read set.

Write set.

Observation time.

When a transaction is executed this information is entered into the history log for each method invoked on behalf of the transaction. The methods invoked by the transaction have a unique transaction id and the sequence number. The sequence number indicates the order of execution of the methods within a transaction. The invoker id, i.e. the sender of the message, and the receiver id, i.e., the receiver of the message is recorded for each method invocation. The name of the method invoked and the values of the formal arguments passed to the method are also recorded. During the execution of the method the system keeps track of the attributes read and written by the method. These constitute the Read set and Write set of the method invocation. Parameters, Read set and Write set are set valued attributes of the entity history log. We will consider them as such in the following discussion, though the log entry may be implemented using its first normal form representation. The commit time indicates the commit time of the transaction. All the methods invoked on behalf of the transaction have the same commit time in this model. Similarly the effective time indicates when the transaction takes effect. It is assumed in this model that all subtransactions comprising the transaction take effect at the same time. There are situations where the subtransactions may have different effective time, depending on the semantics of the operations and application. Such situations should be managed by the concurrency control mechanisms adopted by the system. The issues of concurrency control are not discussed in this thesis. The observation time is associted with queries and is also recorded in the log.

Example 7: Consider the class PERSON.1 which has the following definition:

CLASS	:	PERSON.1
SUPERCLASS	:	OBJECT
sin	:	**SIN.1**
dob	:	**DATE**
name	:	**NAME.1**
salary	:	**FLOAT**

The class PERSON.1 has attributes *sin*, *dob*, *name*, and *salary* of object types SIN.1. DATE, NAME.1 and FLOAT respectively. The classes DATE and FLOAT are system defined classes whereas SIN.1 and NAME.1 are user defined classes. The class PERSON.1 has a method *crperson1*, defined which creates an instance of class PERSON.1. The method *crperson1* in turn invokes calls to methods *crsin1*, *crdate*, *crname1* and *crfloat*, which create an instance of classes SIN.1, DATE, NAME.1, and FLOAT respectively. Each of these methods returns an internal object identifier of the instance created, which is recorded in the log. The invocation of method *crperson1* will generate the following log entries. The parameters and the read and write set are not shown below.

Tr. Id.	Seq. No.	Invoker Id.	Recv. Id.	Method Invoked	Effective Time	Commit Time
1	1	*uid*	*oid*	*crperson1*	Jan. 1, 1989	Jan. 1, 1989
1	2	*crperson1*	*oid*	*crsin1*	Jan. 1, 1989	Jan. 1, 1989
1	3	*crperson1*	*oid*	*crdate*	Jan. 1, 1989	Jan. 1, 1989
1	4	*crperson1*	*oid*	*crname1*	Jan. 1, 1989	Jan. 1, 1989
1	5	*crperson1*	*oid*	*crfloat*	Jan. 1, 1989	Jan. 1, 1989

Conceptually the receiver of the create message is the class, (In our model Classes are also treated as objects), which on receiving the create message creates an instance of the class. For the purpose of query processing it is necessary to know the instance's identifier. The instance's identifier is returned by the method which implements the create message. The *oid* of the instance created is recorded in the log as the receiver of the message. The *oid* are used by the system for query processing, and the user do not have access to the internal object identifiers. The *uid* represents the user identifier, the id of the user who initiated the transaction. The system identifies each user with a unique id. □

2.10. INVALIDATION OF LOG ENTRIES

Retroactive updates have an effective_time earlier than its commit_time. Such a transaction will affect the transactions (queries/updates) that has read/written a retroactively updated value. Recursively other transactions that read/write an attribute from an affected transaction are also affected. The system identifies such affected transactions and informs the user to take appropriate action.

Consider a retroactive update transaction T_i, with a write set W_i, having effective_time t_{ei} and commit_time t_{ci}. Consider a update transaction T_j, with a write set W_j, having an effective_time, t_{ej} and commit_time t_{cj}. The transaction T_j will be affected by the retroactive update transaction T_i when the following holds. If $t_{cj} < t_{ci}$ and $W_j \cap W_i \neq \emptyset$ or $R_j \cap W_i \neq \emptyset$ and either

i. $t_{ei} \leq t_{cj} \rightarrow T_j$ is invalidated

ii. $t_{ej} < t_{ei} \rightarrow T_j$ is valid in time t, ej $\leq t <$ ei

CHAPTER 3

IMPLEMENTATION

A Temporal Object Oriented Data Model is proposed and a prototype has been implemented. The programming language used for the implementation is C, and the system is implemented on Apollo Workstation running under Domain OS and SUN workstation running under SUN's UNIX. The system provides a structured menu-based interface which assists users in defining and accessing the database. The data operations supported by the system include schema definition (including evolution), database creation, and data manipulation i.e., retrieval and updates by invoking operations on objects. The system offers the basic functionalities of database system and object-oriented system: persistence, concurrency and recovery, static type checking, encapsulation, object identity, complex objects management, single inheritance, and overloading. In the following sections, the structure and implementation details of the system are described.

3.1. SYSTEM STRUCTURE

The System conceptually comprises of *Schema Manager*, *Method Manager*, and *Data Manager*, which interact with the underlying System Hardware and Software, to provide an Object-Oriented Environment. *Schema Manager* is responsible for creating, storing and maintaining the (type) structure description of the Classes and method specification (interface). *Method Manager* is responsible for storing and maintaining the source and executable codes of methods. *Data*

Manager is responsible for creating, storing and maintaining objects, i.e., instances of Classes. These components interact with each other to provide a cohesive Object-Oriented Environment.

The major implementation issues of the system are to facilitate Class specification, Method specification, their storage on secondary devices, creation and storage of object instances to provide persistence of objects, and ease of accessibility to objects.

3.2. SCHEMA MANAGER

Schema Manager keeps track of the Classes and Method specifications. of the system via system files classname.sys, classhirchy.sys, verhirchy.sys, time.sys, classver.def.

The system file *classname sys* acts as the Class name server of the system, ensuring that all the Classes being defined have a unique name. It also records the number of different versions of a Class existing in the system. When a class is being defined for the first time, it is the first version, any subsequent (schema) change to the class definition results in a new version of the Class. This class evolution is captured by the system file *verhirchy sys*, which keeps track of the parent deriver version and derived version of the Class. The System Class Hierarchy is recorded in *classhirchy.sys* file, which records the superclass of a class, and the class specific information (discussed below).

The system files make use of System Defined Class Time, to record the valid_time and commit_time of the classes, which are the instances of Class Time and are stored in system file *time.sys*. The Class Time has the following structure:

time.sys

year	:	**INT**
month	:	**INT**
day	:	**INT**
hour	:	**INT**
minute	:	**INT**
seconds	:	**INT**

It permits one to have a wide range of granulity of time scale. The temporal operations on Class Time (Before, After, Equal) may use different granularities of time scale.

All instances of user defined classes are internally identified by system assigned object identifiers, *oid*. The *oid* serves to identify the class of the instance, the instance id in the class, and version id of the instance, so cid:iid:vid make up the oid. A class can have many instances, and each instance can have many versions (object evolution), the system files *classhrchy.sys* and *oidinf.sys* keep track of this information. The generation of oid is discussed in the section of Data Manager.

The structure of the system files *classname sys* and *verhrchy sys* is shown below.

classname sys

Object_Name	:	**string**
Version_Count	:	**INT**

The above record structure is used to capture list of Classes in the system and their version count. INT is the integer type supported by C and string is the character array.

```
verhirchy.sys
Object_Name     :   string
Object_Version  :   string
Derived_From    :   string
```

The system file *verhirchy.sys* keeps track of derived version hierarchy of the Class. Object_Name concatenated with Object_Version gives the Class name, and Derived_From indicates its parent deriver version.

3.2.1. CLASS SPECIFICATION

To specify the description of the class structure, a Class Definition Language is proposed. The Class Definition Language, CDL, has the following textual format.

```
/* hierarchy status */

OBJECT_NAME  : <string>
SUB_CLASS OF : <string>
[VER. NO. : <integer>]

/* state definition */

List of Attributes:

ATTRIBUTE : <string>
TYPE      : <string>
[VER. NO. : <integer>]

/* Attributes Status */

KEY/NON_KEY [K/N] : <string>
CATEGORY [V/N]    : <string>

/* Classes Validity */

VALID_TIME : <yy/mm/dd>
```

The Class specification is captured by system files *classhirchy.sys* and class definition file, class.def, one for each class specified. In CDL, each class being specified is either the subclass of the root OBJECT or an user defined class whose name and version no. is to be specified. The attributes of the class can be either primitive types (CHAR, INTEGER) or of user defined types, which are specified by the class name and its version no. In order to uniquely identify the object the attribute may be specified as key attribute, the user can refer to the object by specifying this key value. The Category indicates whether the attribute is volatile or non-volatile, all key attributes have to be non-volatile. Volatile means that the attribute may be a void object when the instance is created, all non-volatile attributes have to be specified on creation of the instance.

3.3. METHOD MANAGER - METHOD SPECIFICATION

The Method Manager keeps track of the methods specified for a class and its details. Just like class specification there is a Method Definition Language, MDL, for specifying the method name and its parameters, which makeup the interface. MDL. has the following textual format.

```
METHOD NAME : <string>
ARITY : <integer>

For i:= 1 to arity
  ARGUMENT TYPE : <string>

METHOD TYPE : <Inherited/Defined/Redefined>
METHOD CATEGORY : <Internal/External>

VALID TIME : <yy/mm/dd>
```

The Arity indicates the number of parameters to be passed on method invocation. and the type of each method is specified, where the type is any primitive type or user defined class version. In addition, the method type, method category and method's valid time are specified.

The method specification is captured in system files *classver.mhd*, and *classver.pds*, one for each class. The structure of classver.mhd and classver.pds is as follows:

classver.mhd

Method_Name	:	**string**
No_of_Parameter	:	**INT**
Exec_code_file	:	**string**
Valid_Time	:	**time**
Method_Type	:	**string**
Method_Category	:	**string**
Implemented	:	**INT**
Method_No	:	**INT**

classver.mhd records the general method information whereas the parameter description of the methods is recorded in classver.pds.

classver.pds

Method_No	:	**INT**
Parameter_No	:	**INT**
Parameter_Type	:	**string**

When the user has coded the implementation of the method, then the user specifies the name of the executable filename which implements the method. This is recorded in classver.mhd and the flag Implemented is set on indicating that the method is implemented. It is the users responsibility to ensure that all the auxiliary internal methods have also been implemented which will in turn be invoked by the method implemented. At the time of the specification, the system ensures that the valid time (effective time) of the method is same as that of the class or later.

3.4. DATA MANAGER

Data Manager keeps track of the instances of each class. It is responsible for their creation, storage and retrieval. To identify each instance it generates a unique object identifier. The instances of each class are materialized on creation and stored in *classver.ins* file, one for each class.

3.4.1. OBJECT IDENTIFIER

Each object is internally identified by the system by its unique system generated identifier. The identifier has three components, cid - the class identifier, iid - instance identifier, i.e., instance of a class, and vid - version identifier, i.e., the version of the instance.

For each class the system generates a unique class identifier, when the class is specified, and initializes the instance count to zero. Both the instance count and class code are stored along with the other class information in the system file *classhirchy.sys.* When an instance of a class is created the instance count is monotonically incremented, and a version count one is assigned to the instance. The class code, instance count and version count which make up the identifier are stored in the system file *oidinf.sys* On any update operation on an existing object a new version of the instance is created, which is assigned a new identifier by incrementing the version count of the instance in system file oidinf.sys. Both classhirchy.sys and oidinf.sys record the current count of instances and versions respectively. Their structures are shown below.

classhirchy sys

Object_Name	:	**string**
Object_Version	:	**string**
Super_Class	:	**string**
Effective_Time	:	**time**
Commit_Time	:	**time**
Stable	:	**INT**
Class_Code	:	**string**
Instance_Count	:	**INT**

In addition to the identifier's information, the system file classhirchy.sys records the System's Class Hierarchy by storing information about the classes (Object_Name concatenated with Object_Version) Superclass. It records the classes Effective_time and Commit_time which are *oid's* of time instances. There is a flag Stable which is used for indicating whether the Class has been stabilized for usage or not. Stabilization is discussed in sections 2.2 and 3.7.

oidinf.sys

Class_Code	:	**string**
Instance_Number	:	**INT**
Version_Count	:	**INT**

The system is a collection of utilities, which are discussed in Appendix A. The system is setup by executing the system utility *initdb*, which creates the necessary system files *classcnt.sys*, *classhirchy.sys*, *odinf.sys*, *time.sys*, *verhirchy.sys*, and *classname.sys*. This utility is to be executed only once in the beginning, when the system is initialized.

The user can build the system hierarchy by specifying the classes and their methods. The system allows the user the flexibility to specify the method interface of a class independent of the class specification. If the user wants, he/she can specify the methods along with the class specification. To specify the class user makes use of the system utility *define*, which interactively prompts the user to enter the classname, its superclass, its parent class version if it is a derived class, followed by attribute names and their types (Class). Lastly the user enters the classes valid time. This information is recorded in appropriate system files listed above.

To specify the methods for a class the user makes use of the utility *mthspec*, it prompts the user to enter the name of the class, the methods name, its arity and the type of each parameter. It also prompts the user to enter the methods valid time. The method interface specified by the user is recorded in *classver.mhd*, and *classver.pds*. When the user has implemented the method, he/she can use the utility *mthexec*, to record the executable file name in the system and indicate that the method in the class has been implemented.

During Schema specification, the system provides users the flexibility to specify attribute types, i.e., classes, which have not been specified yet. These unspecified classes are assigned version number zero by the system, indicating that they are yet to be specified. The user can make use of the utility *asgnverno*, to assign a version number to the type of an attribute in a class, when the attribute type (class) has been specified. The asgnverno is described in detail in the forthcoming section. The user can use the utility *Stabilize*, to ensure that the necessary temporal constraints mentioned in Chapter 2, are satisfied and the class is available for use. The Stabilize utility and the define utility are discussed in the following sections.

The user can use the utility *browse*, to display the class specification, i.e., it's superclass, its parent class version if it is a derived version, and list of attribute names and their types. It also indicates the category of the attribute, i.e., defined, inherited, or redefined. The display has the following format:

```
CLASS: <classname.verno>
SUPERCLASS: <superclassname>
[DERIVED FRM: <parent classname.verno>]

{ <attributename : attributetype> (defined/inherited/redefined) }
```

3.5. CLASS DEFINITION

The user makes use of the *define* utility, in the current implementation to define, i.e. specify, the class specification. The system assigns a version number to each class defined by the user. The version numbers of a class are monotonically incremented and assigned by the system. The system keeps track of the version count of each class in the system file *classname sys*. When a new class is defined, it is assigned version number one, on subsequent definition of the class it is assigned a new version number by the system. Each version of a class represents a different view of the Object. Classes with versions greater than one are called derived classes, i.e., representing a different view. Each derived class is a result of a change to existing class version, during schema evolution. Hence each derived class version has a parent class version, from which it is derived. This class version derivation hierarchy is recorded in systemfile *verhirchy.sys*. All the versions of a Class have the same superclass.

The user can use the utility *listclassses*, to find out the classname and its total versions specified and the utility *Dispverhirc*, to display the Classes version derivation hierarchy.

The class being defined, can be categorized to belong to one of the following groups in the system class hierarchy.

I. The class is a subclass of the system class hierarchy Root OBJECT, and is the first version of the class.

II. The class is a subclass of an existing user defined class, and is the first version of the class.

III. The class is a subclass of the system class hierarchy Root OBJECT, and is a derived class, i.e., version other than the first.

IV. The class is a subclass of an existing user defined class, and is a derived class.

This is schematically illustrated in Figure 3.1.

The attributes of the classes belonging to Group I, are categorized as defined attributes. Defined attributes are those which are specified by the user while specifying the class. The attribute of the classes belonging to Group III, are categorized as either defined or retained. In group III, derived versions are specified, these may have some new attributes defined by the user and may share some of the attributes from its parent deriver class. The attributes shared between the derived class and its parent deriver class are called retained attributes, their semantics are similar to that of defined attributes. The attributes of the classes belonging to Group II, are inherited or defined. Full inheritance is being modeled in our implementation, and classes belonging to the Group II, inherit all the attributes from its superclass, these attributes are called inherited attributes. The attributes of the classes belonging to Group IV are also either defined or inherited. In addition, the system permits the user to redefine the types of inherited attribute belonging to classes in Group II and IV. The redefined attributes must have the type which is a subclass of their original type.

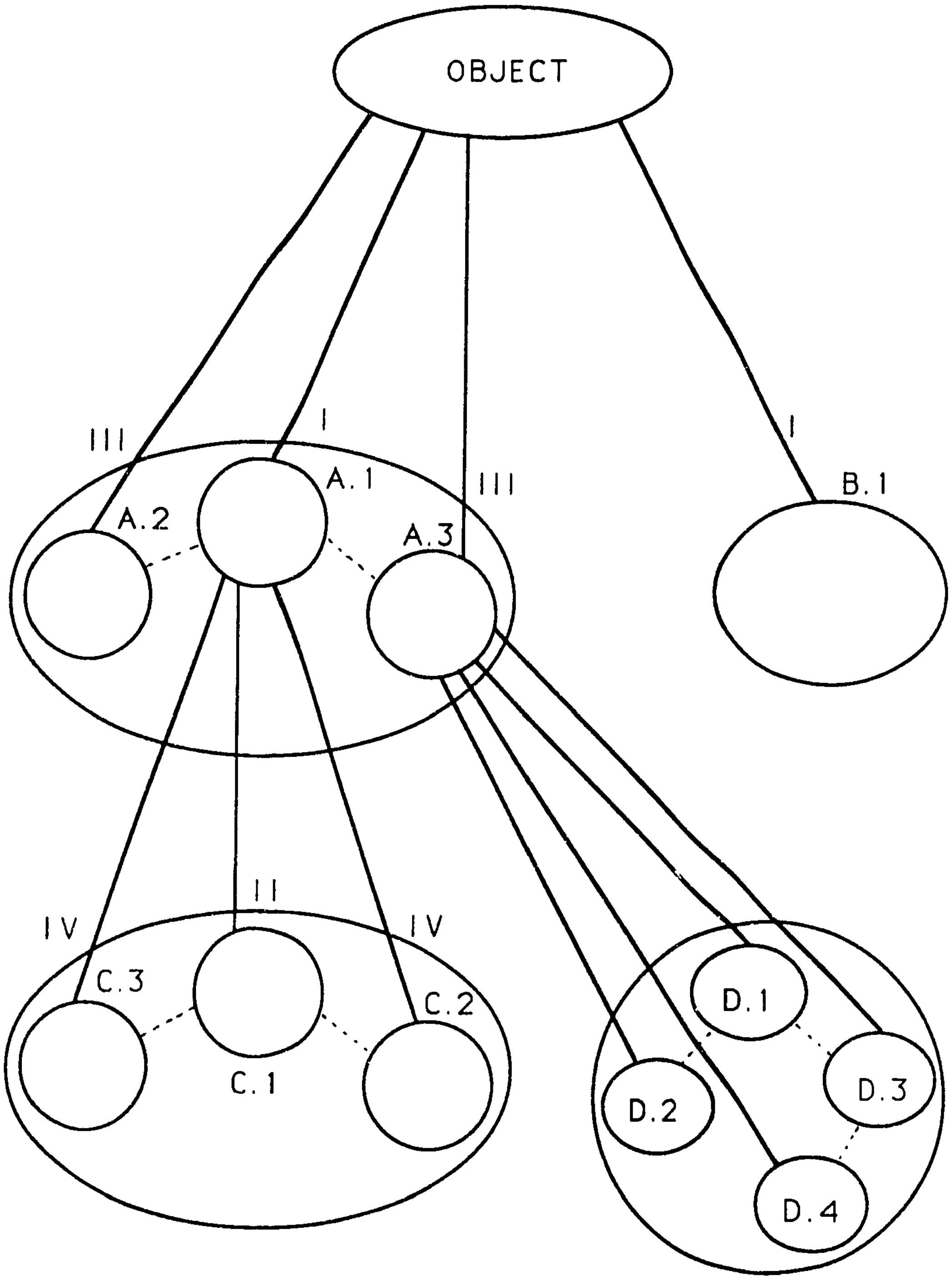


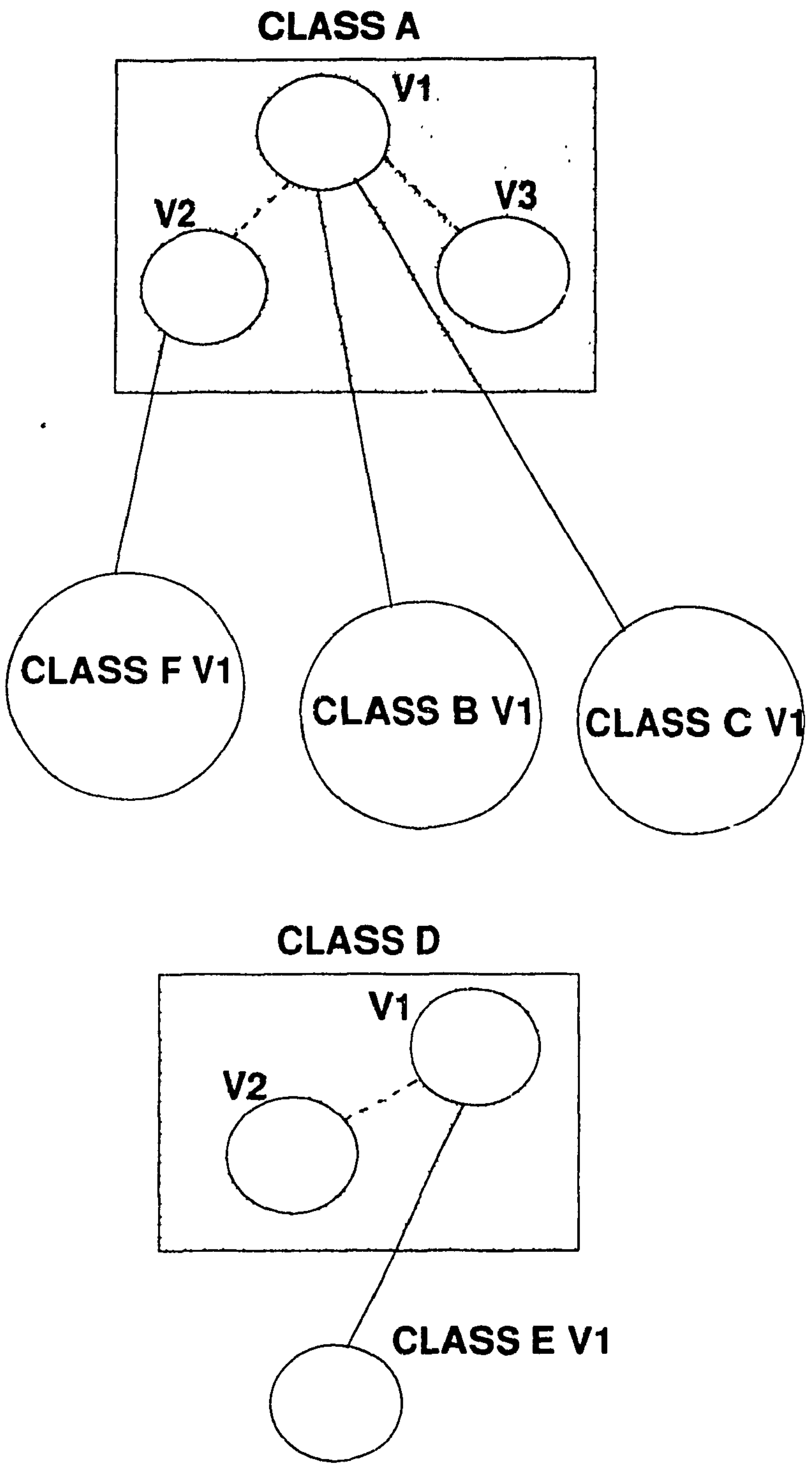
Figure 3.1

3.6. ASSIGNING VERSION NUMBERS

The utility *asgnverno*, allows the user to assign version number to the type (class) of an attribute in a class. The assigned version number is propagated to the existing system hierarchy. The system ensures that the class to which the attribute belongs is unstable, and the attribute is not an inherited one. The utility assigns the version number to the attribute type (class), and then propagates the assigned value to all the subclasses of the class to which the attribute belongs, which have inherited the attribute and not redefined it. In addition to propagating to the subclasses of the class to which the attribute belongs, the version number is also propagated to the derived classes of the class to which the attribute belongs if they have retained the attribute, and to their subclasses if they have inherited and not redefined the attribute.

The version assignment is said to be inconsistent when the inherited attribute is redefined and the type of the redefined attribute is not a subclass of the assigned type. *asgnverno* identifies such assignments and undo's the assignment to maintain a consistent schema.

Consider Classes D_v1, D_v2, E_v1, A_v1, A_v2, A_v3, F_v1, B_v1 and C_v1 shown in Figure 3.2. Class E_v1 is a subclass of class D_v1. Class F_v1 is a subclass of class A_v2. Classes B_v1 and C_v1 are subclasses of class A_v1. Class A_v1 has an attribute w of type D_v1, which is retained in derived versions A_v2, and A_v3. It is inherited by subclasses F_v1 and B_v1. It is redefined in class C_v1 to be of type E_v1. The attempt to assign version D_v2 to attribute w of class A_v1 is an inconsistent assignment because the type of the attribute in the subclass C_v1 is redefined to E_v1, and E_v1 is not a subclass of D_v2, the new type of the attribute in the superclass.


CLASS A
V1
V2
V3
CLASS F V1
CLASS B V1
CLASS C V1
CLASS D
V1
V2
CLASS E V1

Figuer 3.2

3.7. CLASS STABILIAZTION

The utility *stabilize*, ensures that a class satisfies the necessary temporal constraints incorporated in the model and is available to the users. It tests for the specification of the component classes of the class being stabilized. It builds a directed graph, with a directed edge between the class and the component class, as illustrated in Chapter 2. It finds the strong component of the graph, to identify cyclic dependent classes. Then it tests for the following constraints which are to be satisfied by a class.

1. Test whether the superclass of the class being stabilized is stable. If so, it test whether the Effective_time(Superclass) $\leq$ Effective_time(Subclass).

2. Determines whether the class being stabilized is a derived class. If so, it tests whether the driver class (parent class version in derivation hierarchy), is stable .

3. Tests for the constraints to be satisfied by the component classes. Determines whether the superclass of the component class is stable. If so, tests whether the Effective_time(components superclass) $\leq$ Effective_time(component class). It tests whether the Effective_time(component class) $\leq$ Effective_time(class). It determines whether the component class is a derived class. If so, it tests whether the driver class of the component class is stable.

4. Lastly, it tests whether the cyclic dependent class have the same Effective_time.

CHAPTER 4

CONCLUSIONS AND RESEARCH DIRECTIONS

Database applications may be classified into two categories: traditional "commercial" applications and newly emerging data-intensive applications. The former is characterized by large amount of data with relatively small and static conceptual structure (schema). Examples of such applications include banking, reservation, personnel, inventory systems, etc. The database systems used for such applications are dominated by those with record-based modeling technology (hierarchical, network, and relational database models). The latter is exemplified by applications such as office information systems (OIS), design of engineering databases (CAD/VLSI), artificial intelligence (AI) systems. These application environments are more dynamic in nature than those of traditional business applications. In these application environments, the amount of structural information is large and much complex than that of traditional business applications. Moreover changes to the conceptual structure of the database for these applications is quite common phenomena. The conceptual structure of the database changes when the application environment that the database models evolves. Such application "dynamics" are not adequately supported by existing database systems. Object-based modeling approaches are being used for these applications, but to capture the behaviour and evolution of design applications, it is essential to have the notion of time, which is lacking in the existing Object-

Oriented Data Models.

In this thesis we have presented a Temporal Object-Oriented Data Model. which captures the inherent dynamic changes of the conceptual structure (schema), of the non-traditional data-intensive design applications. The model incorporates the notion of time into object-oriented databases to capture object evolution. Schema evolution has been proposed and the temporal constraints to be satisfied by each class and instance is mentioned. Constraints for schema validation are identified and Version management is supported by the system. Methods are used as a means of query processing, which uses history log and checkpoints. Notion of Multityping is introduced to capture the dynamic behaviour of objects, which is a function of time. Membership constraints for each class are proposed which require further study.

Temporal Object-Oriented Databases is an active area of research. Efficient mechanisms for persistent object management is another area of research. Currently work is being done on developing indexing techniques for efficient access to the objects and support queries over complex objects. Representation of constraints and constraint management is an open research issue for TOODBMS. In our model we are limiting the instance evolutions of a class to one default version. In can be extended to allow multiple versions of instances to be derived from potential instance active at any instant of time.

Further work is required to efficiently access and manage the history log. The database may be periodically materialized, thus reducing the size of the log. This implies a restriction on the period of retroactive updates which are usually dependent on the application semantics.

REFERENCES

[1] James F. Allen, "Maintaining Knowledge about Temporal Intervals", in Communications of the ACM, Nov. 1983, pp. 832-843.

[2] James F. Allen, "Towards a General Theory of Action and Time", in Artificial Intelligence 23, 1984, pp. 123-154.

[3] T. Andrews, C. Harris, "Combining Language and Database Advances in an Object-Oriented Development Environment", in OOPSLA, 1987.

[4] F.Bancilhon, "A logic programming object oriented cocktail", in ACM Sigmod Record, 15:3,pp. 11-21, 1986.

[5] F.Bancilhon, "Object-Oriented Database Systems", in Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 1988, pp. 152-162.

[6] Banerjee,J., et al. "Data Model Issue for Object- Oriented Databases Applications", in ACM Trans. on Office Information System, Jan 1987.

[7] Banerjee, J., W. Kim, H.J. Kim, ant H.F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", in Proc. ACM SIGMOD Conf. on the Management of Data, San Francisco, Calif., May 1987, pp. 311-322.

[8] Elisa Bertino, "Issues in Indexing Techniques for Object-Oriented Databases", in Proceedings of Advanced Database Systems Symposium'89, Kyoto Research Park, Kyoto, Japan, December 1989, pp. 151-160.

[9] Brad J. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology", in IEEE Software, Jan. 1984, pp. 50-61.

[10] Brad J. Cox, Object Oriented Programming, An Evolutionary Approach: Addison-Wesley Publishing Company, ISBN 0- 201-10393-1, 1986.

[11] Chou,H.T., and W.Kim, "Versions and Change Notification in an Object-Oriented Database Systems", in Proc. Design Automation Conference, June 1988.

[12] D.H.Fishman, D.Beech, H.P.Cate, E.C.Chow, T.Connors, J.W.Davis, N.Derrett, C.G.Hoch, W.Kent, P.Lyngbaek, B.Mahbod, M.A.Neimat, T.A.Ryan, and M.C.Shaw, "Iris: An Object-Oriented Database Management System", in ACM Transactions on Office Information Systems, January 1987, pp. 48-69.

[13] Garza,J.F., and W.Kim "Transaction Management in an Object-Oriented Database System", in Proc. ACM-SIGMOD Intl. Conf. on Management of Data, Chicago, May 1988.

[14] Goldberg, A. and D. Robson. "Smalltalk-80: The Language and its implementation, Addison-Wesley, reading, MA 1983.

[15] P.Goyal, M.Okada, Y.Z.Qu, F.Sadri, "Temporal Object-Oriented Database:(I) Data Model and Formalism" in Proceedings of Advanced Database System Symposium'89, Kyoto Research Park, Kyoto, Japan, December 1989, pp. 121-128.

[16] Kim, W., H.T. Chou, and J. Banerjee, "Operations and Implementation of Complex Objects", in Proc. Data Engineering Conference, Los Angles, Calif., Feb. 1987.

[17] Kim, W., et al. "Composite Object Support in an Object-Oriented Database System" in Proc. Object-Oriented Programming Systems, Languages, and Applications, Oct. 1987, Orlando, Florida, pp. 118-125.

[18] Kim, W., H.T. Chou, "Versions of Schema for Object-Oriented Databases", in Proceedings of the 14th VLDB Conference, Los Angeles, California 1988, pp. 148-159.

[19] Kim, W., Elisa Bertino, J.F.Garza, "Composite Objects Revisited",in Proc. ACM-SIGMOD June 1989.

[20] Q.Li, "Accommodating Application Dynamics in an Object Database System", in Proceedings of Advanced Database System Symposium'89, Kyoto Research Park, Kyoto, Japan, December 1989, pp. 97-104.

[21] Lung-Chun Liu, Ellis Horowitz, "Object Database Support for a Software Project Management Environment.",in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 1988.

[22] Maier, D., J. Stein, A. Otis, and A. Purdy, "Development of an object-oriented DBMS", in Proc. ACM Conference On Object Oriented Programmimg Systems, Languages, and Applications, Portland, Oregon. September 1986.

[23] Josephine Micallef, "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages", in Journal of Object-Oriented Programming, April/May 1988, pp. 12-38.

[24] D.Jason Penney, Jacob Stein, "Class Modification in the GemStone Object-Oriented DBMS", in OOPSLA 87, pp. 111-117.

[25] Skarra, A.H., and Z.B. Zdonik, "The Managament of Changing Types in an Object-Oriented Database", in Proc. ACM Conference On Object Oriented Programming Systems, Languages and Applications. Portland. Oregon, September 1986.

[26] Andrea H. Skarra, Stanley B. Zdonik, "Type Evolution in an Object-Oriented Databases", in Research Directions in Object-Oriented Programming, 1987, pp. 393-414.

[27] Snodgrass, I. Ahn, " A Taxonomy of Time in Databases", in Proc. Int'l Conf. Management of Data, ACM SIGMOD, Austin, TX. May 1985, pp. 236-246.

[28] Richard Snodgrass, Ilsoo Ahn, "Temporal Databases", in IEEE Computer, Sept 86, pp. 35-42.

[29] Michael Schrefl, Erich J. Neuhold, "Object class definition by generalization using upward inheritance", in IEEE Proceedings Fourth International Conference on Data Engineering, Feb 1-5, 1988, pp. 4-13, 1988.

[30] S.M. Sripada, "A logical framework for temporal deductive database", in Proceedings of the 14th VLDB Conference, Los Angeles, California 1988.

pp. 171-182.

[31] Francois Bancilhon, Gilles Barbedette, Veronique Benzaken, Claude Delobel, Sophie Gamerman, Christophe Lecluse, Patrik Pfeffer, Philippe Richard, Fernando Velez "The Design and Implementation of O2 ,an Object-Oriented Database System", Advances in Object-Oriented Database Systems, Lecture Notes In Computer Science 331, pp. 1-22.

[32] Woelk,D., Kim,W., Luther,W. "An object-oriented approach to multimedia databases", in Proceedings of ACM SIGMOD Conference on the Management of Data, ACM, New York, 1986.

APPENDIX A

TEMPORAL OBJECT-ORIENTED DATA BASE

User's Manual

A.1 User Commands

1. Initdb

Purpose: To initialize the system and create system files.

Usage: initdb

This set's up the system by creating the system files *classcnt.sys*, *classhirchy sys*, *odinf sys*, *time.sys*, *verhirchy.sys*, and *classname.sys*, in a subdirectory *sysfiles*. During system initialization, the specification of the System Defined Class TIME is recorded in classhirchy.sys. A class id is generated for the class TIME, and its system default valid time Jan 1, 1900, is recorded in time.sys.

2. Define

Purpose: To specify class description.

Usage: define

This utility allows the user to specify the structure of a class. It records the structural information in system files *classver.def*, *classhirchy.sys*, *classname.sys*, *verhirchy.sys*.

3. Dispcvtime

Purpose: To display the valid time of the given class.

Usage: dispcvtime <classname.verno>

Searches the system file *classhirchy.sys* for the existence of the class. Scans the system file *time.sys* and displays the valid time of the class. The time object identifier from the classhirchy.sys is used as search key in time.sys. The system displays "class vtime is DD MM YYYY".

4. Changecvtime

Purpose: To change the valid time of the given class.

Usage: changecvtime <classname.verno>

Searches the system file *classhirchy sys* for the existence of the class and ensures that it is not stabilized. Accepts the new valid time for the class and replaces the previous valid time in *time sys* with the new one using time instance oid from the classhirchy.sys as the key.

5. Mthspec

Purpose: To specify the method interface of a class.

Usage: mthspec

The user enters the class for which the method interface is to be specified. The system file *classhirchy.sys* is scaned to ascertain that the class exists. The user is prompted to enter the method name, its formal parameter types, method type, method category and effective time of the method. This information is stored in system files *classver.mhd* and *classver.pds.* It also checks for the temporal constraint between the class and the method. If the method is older than the class, the user is informed of it.

6. Mthexec

Purpose: To specify the executable code file name which implements the method in a class.

Usage: mthexec <classname.verno methodname>

Searches the system file *classhirchy sys* to ascertain the existence of the class whose method has been implemented. If the class exists then the *classver.mhd* is scaned to determine that the method has been specified. If the method exists, the executable file name which implements the method is accepted and recorded in *classver.mhd.* Appropriate flag is set indicating that the method has been implementd.

7. Dispmvtime

Purpose: To Display the valid time of the method in a class.

Usage: dispmvtime <classname.verno methodname>

Searches the system file *classhirchy.sys* to ascertain the existence of the class, and search for *classver.mhd* to determine that the methods have

been specified for the class. Scan the file classver.mhd for the methodname, if the method exists scan the system file *time sys* and display the time using *toid* (time instance object identifier) from classver.mhd as the key. The display has the following format:
Valid time of method <methodname> in class <classname.verno> is DD MM YY.

8. Changemvtime

Purpose: To Change the valid time of the method in a class.

Usage: changemvtime <classname.verno methodname>

Searches the system file *classhirchy sys* for the existence of the class and ensures that it is not stabilized. Search for *classver mhd* to determine that the method have been specified for the class. Scan the file classver.nhd for the methodname. if the method exists, accepts the new valid time for the method and replaces the previous valid time in *time.sys* with the new one using time instance oid from the classver.mhd as the key.

9. Asgnverno

Purpose: To Assign a version number to the type of an attribute in a Class, and propagate it to the system hierarchy.

Usage: asgnverno <classname.verno attrname attrtype >

Searches the system file *classhirchy.sys*, to determine that the class whose attributes type has to be assigned a version number exists and is not stabilized. Searches the file *classver.def*, to ascertain that the attribute is not inherited. It assigns the version number to the attribute type, and finds the subclasses of the class and derived classes of the class. It propagates the version number to the subclasses which have inherited the attribute and to the derived classes which have retained the attribute.

10. Browse

Purpose: To display the structural specification of a Class.

Usage: browse <classname.verno>

Searches the system file *classhirchy.sys* for the existence of the class, and finds the superclass of the class. Scans the system file *verhirchy.sys*, to determine if the class is a derived version or not, and scans the *classver.def* file to find the structure of the class. It displays the Classname, its superclass, and parent class version if it is a derived class version. It also displays the list of attribute names and their types and category of the attribute (defined, inherited or redefined).

11. Disphirc

Purpose: To Display the System Class Hierarchy.

It scans the system file *classhirchy.sys* for the information about the classes specified and constructs a multiway tree. Each node of the tree represents a class and its children represent its subclasses. The subclasses of a class are sorted by name. To display the class hierarchy the depth-first search of the multiway tree is performed. For example, consider a system consisting of classes A.1, A.2, A.3, B.1, and C.1, where A.1, A.2 and A.3 are subclasses of the root OBJECT. B.1 and C.1 are subclasses of A.1. The display of the System Class Hierarchy has the following presentation.

```
OBJECT
  A.1
    B.1
    C.1
  A.2
  A.3
```

12. Stabilize

Purpose: To Stabilize a Class for user usage.

Usage: stabilize <classname.verno>

It scans the *classver.def* file and constructs a directed graph with edges between the class and component classes. It repeatedly scans *classver.def* file of all the classes involved in the graph until no new nodes are inserted in the graph. Once the graph is constructed, the nodes in the strong component of the graph are identified. These nodes represent cyclic dependent classes. To stabilize the class the following constraints are checked. The superclass of the class being stabilized must be stable. The effective_time of the superclass of the class being stabilized must be earlier or same as that of the subclass. If the class being stabilized is a derived class then its deriver class must be stable. The superclasses of the component class must be stable, and the effective_time of the component classes must be earlier or same as that of the class. The cyclic dependent classes must have the same effective_time.

13. Liststable

Purpose: To list the stabilized class in the system.

Usage: liststable

Scans the system file *classhirchy.sys* and displays the classname's of the classes which have their stable flag set, i.e., they are stabilized.

14. Lismth

Purpose: To List the methods specified in a class.

Usage: lismth <classname.verno>

Searchs the system file *classhirchy.sys* for the existence of the class. and searches *classver.mhd* to determine whether methods have been specified for the class. Displays the methods specified using the following format:

{ Method: <methodname> }

15. Disverhirc

Purpose: To Display a Classes version derivation hierarchy.

Usage: disverhirc <classname>

Searches the system file *classname.sys* to determine whether the class has derived versions. It scans the system file *verhirchy.sys* to display the list of derived versions of the class and their parent class version.

The display has the following format:

{ CLASS VERSION: <classname.verno> DERIVED FRM: <parent classname.verno> }

16. Listclasses

Purpose: To display the classes specified and their version count.

Usage: listclasses

Scans the system file *classname.sys*, and displays the classname and the version count of the versions of the class specified. The display has the following format:

{ OBJNAME: <classname> VERSIONS: <versioncount> }

17. Resetverno

Purpose: To Assign a version number to the type of an attribute in a Class.

Usage: resetverno <classname.verno attrname attrtype newverno>

This utility is used for testing during system development. It allows the user to assign any version number to the type of an attribute of a class. The user specifies the class and the attribute of the class and its type. The system scans the *classver.def* file for the existence of the attribute with the specified type. If the attribute with the specified type exists, it is assigned the newverno. It is the users responsibility to ensure the validity of the schema after the assignment of the new version number to the type of the attribute.