The Application of Object-Oriented Views to an Engineering Environment.

Thesis

UNRESTRICTED

The Application of Object-Oriented Views to an Engineering Environment

by

Zhuang Shao

M.Phil.

Computer Science

IN PARTIAL FULFILMENT
OF THE REQUIREMENTS
OF THE OPEN UNIVERSITY

School of Information Systems and Computing

University of Wales Institute, Cardiff

2nd September, 1999

AWARDING BODY:
THE OPEN UNIVERSITY

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

With the increasing popularity of object-oriented technology, object-oriented database systems are being used in design environments as central repositories. In this thesis, we investigate the role of versioning and the characteristics of design databases in design environments. In an effort to improve the configuration management scheme in a design environment, we also investigate the use of database views as a possible configuration tool.

We propose a unified version management scheme that facilitates cooperative team work and show that the use of database views provides a powerful configuration management scheme for a design environment.

ACKNOWLEDGMENTS

Many people have influenced the work reported in this thesis, and it is my pleasure and privilege to acknowledge their contributions.

Firstly, I am indebted to my supervisors Tom Carnduff, David Ball and Alex Gray, who have helped to form many of the ideas reported in this thesis, and encouraged me throughout their development.

My love and special thanks are due to Xi for her total support throughout the course of this work. Without her encouragement this thesis would never have happened.

Table of Contents

Table of Figures

Table of Tables

Table of Lists

Chapter 1 Introduction

The development of a product is seen as the manipulation of a set of complex objects. To support complex design activities in modern society, a database is often used as central repository to store all the information. Object-oriented databases (OODB) more naturally reflect the behavior and organization of complex application domains and therefore are ideal candidates for a design database. With the increasing popularity of object-oriented database system, object-oriented views have attracted a lot of attention from the database research community. Views have been recognised as a way of virtually restructuring and customizing objects both in format and behaviour.

Object-Oriented databases are widely used in CAD/CAM/CASE. These applications require their databases to provide following capabilities[KATZ87]:

- Definition and manipulation of complex object,
- Management of variants and revisions of both the design artifact and its components.

One of the problems in a design process is that there are so many different ideas about how a product should be designed. In an uncontrolled environment, this leads to incompatibility and inconsistencies. A versioning facility in a database is there to support the tentative and iterative nature of design activities where designers are encouraged to experiment with different variants and revisions of a design [SCIORE91].

Configuration management is a form of organization that provides stability to the production of complex objects by controlling the object evolution, i.e., continued and concurrent changes. Configuration management provides a stable working environment for changing the design objects, supports the assembly of a complex design artefact from its components, and coordinates concurrent changes [FEILER91].

Configuration management ensures the consistency of and compatibility between component objects of a complex design object. A configuration can be generated by selecting component objects that satisfy some selection criteria such as configurations that incorporate specified features, or check whether a user-specified configuration is correct [AGRAWAL89].

In this thesis, we describe a configuration management mechanism in a design environment, which puts emphasis on the semantic relationship between the components and the complex object. The configuration management mechanism is based on the assumption that a component object has to meet certain design criteria before it can be integrated into a complex object's configuration. A configuration may be treated as a versioned object. The configuration management framework incorporates object-oriented views to provide an expressive and flexible scheme for defining configuration criteria.

1.1 Thesis Aims and Objectives

The aim of the research project was to investigate how an object-oriented database could provide efficient support for cooperative work in an engineering design environment with particular emphasis on the use of database views to support design activities.

To achieve the aim we will:

- Investigate the role of versioning and how it can be used to support design activities in a design environment.

- Analyse the characteristics of a design database when used as the central repository of a design environment.

- Investigate the need for, and characteristic of a database view mechanism as an integral component of a design environment.

The hypothesis of this research is that object-oriented database systems provide better support for change management in a design environment than second generation database systems, particularly through the use of database views.

Within this thesis we will prove the hypothesis by demonstrating that:

1. Object-Oriented Databases provide better support for design environments than second generation databases.

2. Object-Oriented databases are able to provide a flexible version control that suits the needs of an engineering design environment.

3. Object-Oriented views provide a flexible and efficient framework for organising a design environment.

4. View materialization provides an effective configuration management scheme.

5. Object-Oriented views provide a powerful and flexible configuration component selection scheme for configuration management.

1.2 Structure of the Thesis

The thesis is organised as follows:

Following the short introduction given in this chapter, chapter 2 looks in detail at the requirements and characteristics of a database supporting a design environment, showing how it can be used to effectively support team work in a constantly changing environment.

Chapter 3 describes the concept of versioning and what is required to capture the changes in a constantly changing world. The semantics of versioning and the need for configuration management are discussed. Versioned complex objects need

configuration management to control changes in their components. Various configuration management approaches are discussed in this chapter and the advantage and disadvantage of each approach is compared.

Chapter 4 introduces the notion of views in an object-oriented database and a taxonomy of object-oriented views is presented. The ODMG standard query language OQL is introduced in this chapter since OQL is used to define views later in this thesis.

Chapter 5 describes an object view materialization strategy capable of supporting versions of design artefacts in different workspaces. In the context of this strategy, some of the problems associated with views, such as view updates, are discussed.

Chapter 6 presents the use of object-oriented views in a design environment in order to provide a unified version management scheme and goes on to show how to use the view in configuration management. In particular, the means of using views to identify component objects are described together with the underlying structures which support these operations.

Chapter 7 presents the prototype of our proposed model for a design environment. Chapter 8 evaluates the effectiveness of the proposed model.

Chapter 9 reviews the aim of the thesis, and discusses possible further developments of the work reported here.

CHAPTER 2 Integrated Design Environment

A large engineering design project typically involves a team of designers working cooperatively on distributed workstations in order to complete a composite design task. These designers usually interact dynamically, sharing ideas, design data and general information with each other. The key element in providing efficient support within such a development environment is integration as it is essential for designers to communicate their ideas efficiently to coordinate their design efforts. To facilitate collaborative development, it is essential that the integrated design environment supports the following features [PRESSMAN94, AHMED91a]:

- Composite information modelling capabilities. Engineering data is composite in structure because of the complexity of the domain that is being modelled. Design entities may be interrelated to each other, e.g., a design artefact consists of various components where these components themselves may have components of lower complexity.

- Able to capture rich semantic information in design entities. Because of inherited complexity in engineering data, database schemes must reflect the design semantics and hierarchy. It is important that the database can capture composite inter-object relationships and dependencies in the data model.

- Provide constraint management. Due to the inter-object relationship and dependencies in the engineering data, consistency of data in the database must be maintained by enforcing design constraints and integrity constraints during the development process.

- Support information sharing. One of the key issues in collaborative engineering is the sharing of design data between teams of designers. It should be possible to partition or group data based on criteria, such as ownership, use and purpose of creation, or any other meaningful purpose.

- Provision of version control. Engineering design is an incremental process and evolves with time. A versioning facility would provide a mechanism for capturing the evolution history of a particular design over its development process. If any version were found to be faulty at any stage, it should be possible to "rollback" the design to a valid state. Versioning also promotes concurrency as designers may work concurrently on different versions of the same object instead of waiting for each other to release the resource.

- Enable changes to one item to be tracked to other related items. Composite inter-object relationships in engineering data mean that changes to one design might affect other objects. The capability will ensure that all the related objects can be identified and notified about the changes and consequently the changes would not invalidate the whole design.

Many of these are the features that are normally found in Database Management Systems (DBMS). Therefore, database management systems are often used as data repositories in design environments. The use of databases in design environments is based on the need to manage a wide range of design information efficiently, and effectively. In the following section, we will discuss the advantages of using databases in design environments.

2.1. Advantages of using a database

Using databases in design environments has the benefit that it provides centralised control over all data in the environment. For engineering design applications, the data will include all the information generated during the development life cycle, in particular, design requirements, specifications, implementation, integration, testing and error reports. In addition to data concerning the design artefact, details of the project development process itself is stored in the database, such as which designer is responsible for a particular design and how these designs are interrelated.

One of the consequences of central control of data is that we can greatly reduce redundancy in data storage. Reducing redundancy is a great help in removing inconsistencies in the data. When a change is made to a component, all larger components that have been using this object must be notified and appropriate actions have to be taken to respond to the change. This property of change control is much easier in a centrally controlled environment, especially using a database that not only stores the data items themselves, but also the relationships between them.

One of the problems in the design environment is that there are so many different ideas about how to achieve a design objective. In an uncontrolled environment, data using different design approaches can lead to incompatibility and inconsistencies between the data. This situation can be avoided by central control of all the development data. Central control is an important method of enforcing a set of development standards, providing a single point at which all design data entering the database can be validated before it is stored.

Database systems must ensure integrity of all data stored. There are two aspects to integrity. The first is that access to data can be monitored, so that each user is presented with an individual subset of the completed data, by using a view mechanism. This allows the user to get on with his/her work without distraction from irrelevant data, and provides a mechanism for restricting access to privileged information.

The second aspect is to ensure that the data recorded is accurate and conforms to constraints the designers may wish to impose on it. This is partly covered by reducing redundancy, but additional data validation is possible by defining a central set of integrity constraints, which may be applied to data before entry to the database.

2.2. Characteristics of a Data in Design Environment

The engineering design process is highly data intensive, and it involves composite data representations to model the structure and behaviour of complicated entities. Engineering information is not only complex in structure but also in terms of relationships between data. For example, a complex design object, (also known as aggregate object), may contain several components and these components may in turn have their own components. These components form *part-of* relationships with the aggregate object. Therefore, it is important to control the relationships between these objects in order to meet design constraints or requirements.

A design process is both tentative and iterative. This has a profound effect on the growth of a design database, since it is necessary to keep a record of all amendments to a design object as a new version of that object. The database should be able to get the appropriate object and the right version for each request of a design item. When changes are to be made to a design object, the designer should be able to assess and identify the effect these changes may have on other related objects. This helps prevent unexpected side effects that would otherwise cause defects and inconsistencies in a design.

The transaction is considered to be a unit of database consistency and concurrency. A typical transaction in a traditional database application is of short duration. Serializability is enforced for concurrent transactions in order to maintain database consistency. In collaborative design environments, the notion of a transaction is very different from its traditional sense, and has the following characteristics [BROWN89]:

- Conversational, requiring frequent interaction with the system before completion.

- Long-lived transactions which may leave the database in an inconsistent state for long periods of time. Cannot be conveniently used as locking units.

- May use many records, as the objects accessed may be complex and highly inter-related.

- The concept of atomic transactions is not very applicable, because rolling back a long transaction in this environment may turn out to be impractical.

In a multiple-user environment, some forms of concurrency control must be provided to prevent interference. In traditional databases, concurrency control ensures that only one user can update a particular object at any time. However, in an engineering design environment transactions are normally of long duration, refusing other user's access to a locked record for a long time is unacceptable. A more flexible locking mechanism is needed to allow greater concurrency, so that transactions do not have to wait indefinitely for each other to complete.

Relational database systems and their predecessors are designed for business applications, to store such information as personal details or bank records. These database systems are highly efficient in these application areas, but are not necessarily suitable for other application areas whose characteristics differ greatly from business applications. The relational database is good at handling data that is confined to a small number of different types of data related in well-defined ways. For design

applications, such as CAD/CAM, the relational data model has many limitations [KENT79, KIM90]:

- Relational data models are severely restricted in their modelling power. The relational model is not complex enough to capture nested entities. The relational data model does not support some of the commonly useful semantic concepts, such as generalization and aggregation relationships.

- Relational data models assume horizontal homogeneity. This means that each record of a certain record type is assumed to be composed of the exact same fields.

- Relational database systems assume vertical homogeneity, i.e., each field should be from the same domain in all the records.

- Only a fixed set of operations are allowed on atomic data values, such as arithmetic and comparison operations. It is not possible to add new operations and make those operations appear syntactically similar to built-in operations.

- Meta-information is generally not accessible. This results in a program text that includes hard coded data based on prior knowledge of the schema, making alterations to both the schema and the program difficult to manage.

- Dynamic objects such as sets have to be implemented using several records and join operations, causing inefficiency.

- Transaction time for design objects (e.g., data from CAD) is often long, spanning several hours or days, and not uncommonly, weeks or even months. This is in strong contrast to business data processing transactions, which are assumed to be short lived. Concurrency control primitives and protocols (such as two-phase locking) supported by relational databases are not particularly suitable for long-lived transactions.

- The performance of a relational database system is not satisfactory for computationally intensive applications.

Recognizing the inadequacies of relational databases, the database research community has been trying to extend database systems with enhanced semantic data modelling concepts. These research efforts have led to the development of Object-Oriented Databases which offer better modelling semantics for complex data structures such as those found in design environments.

2.3. Objects and Databases

The best candidate system upon which to base an integrated design environment is one that supports rich modelling semantics and exhibits features required by design

environments. Object-oriented databases are different from previous generations of databases in that they offer greater flexibility in new type definition and data abstraction. As well as having all the features found to be useful in relational databases [STONEBRAKER90, ATKINSON89] object-oriented databases should also offer features that are highly desirable in design environments.

Fundamental to the object-oriented data model is its ability to extend the class hierarchy with new classes. In object-oriented database systems, data types are represented as classes within a class hierarchy and can be extended easily. Extensibility is a very powerful mechanism for building and evolving large and complex design artefacts. Inheritance is the one of key features that supports extensibility in object-oriented database systems.

Apart from the purely structural data model found in previous generations of databases, the object-oriented data model embodies a more behavioural model, combining representation and manipulation of data within the same model. Each class of objects has a set of well-defined methods. Object states can only be modified through designated methods of the object. This is guaranteed by a mechanism called data encapsulation. Encapsulation not only protects data from unauthorised or unintended modification but also minimises the impact of changes in implementation.

Although there is no formal definition for the object-oriented data model, the object-oriented community has agreed that the object-oriented data models should possess certain features, as follows.

2.3.1. Object and Object Identity

The object is the basic unit of an object-oriented database. Everything is modelled as an object in an object-oriented database. An object has a number of data properties, known as attributes, associated with it which represent the current state of the object. They can be manipulated through a set of well-defined functions of the object. Each object is identified by a unique object identifier. In object-oriented databases, this object identifier is system generated and is associated with the object throughout its life time. Unlike the relational data model, the object identifier frees the user from the need to define unique keys for objects and it allows equal objects (objects that have the same attribute values) to coexist.

2.3.2. Encapsulation and Methods

Objects are manipulated by methods that are defined on their classes. Data in an object can only be accessed through these well-defined methods. These methods are invoked by messages sent to the object with which they associate. The implementation of these methods may change without invalidating their use.

2.3.3. Class Hierarchy and Inheritance

In the object-oriented data model, objects are organized in taxonomies through inheritance. In such a model, specialized objects inherit the attributes and methods of more generalized ones. The inherited methods can be modified in the subclass. This

is known as *overriding*. This feature enables the reuse and incremental redefinition of a new class structure in terms of existing ones. Similar classes of objects sharing common attributes and methods can be modelled by specifying a superclass, and then deriving specialized classes (subclasses) from the superclass.

A class may have any number of subclasses. However, some object-oriented systems allow a subclass to have only one superclass, i.e., single inheritance, while others allow a subclass to have more than one superclass, i.e., multiple inheritance. The class hierarchy captures the generalization/specialization relationships between a class and its subclasses. Figure 2.1 shows example of a class hierarchy.

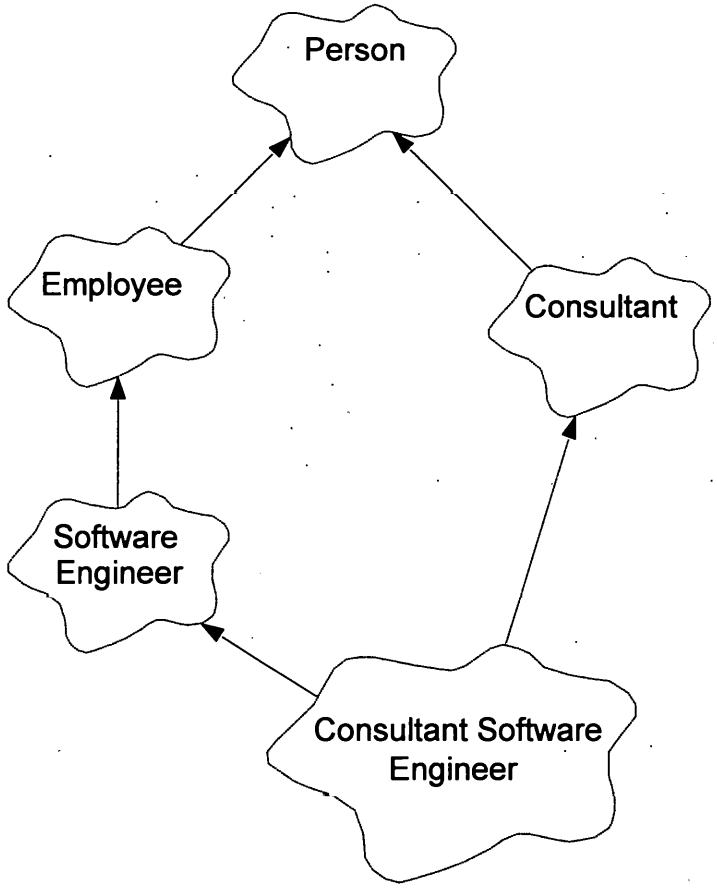


Figure 2.1 Example of Class Hierarchy

2.3.4. Composite Object

A composite object is a heterogeneous set of objects which form a part hierarchy. The part-of relationship is superimposed on the aggregation relationship between an object and the other objects it references [KIM90]. The attributes of a composite object may be objects themselves. The value of the attribute is a reference to an object An object may have a number of references to other objects.

In composite objects, the referenced object can be seen as a component of the object. For example, a Car object has the components Engine and BodyWork. The Engine object itself has a component GearBox. The structure of the composite object Car is illustrated in Figure 2.2.

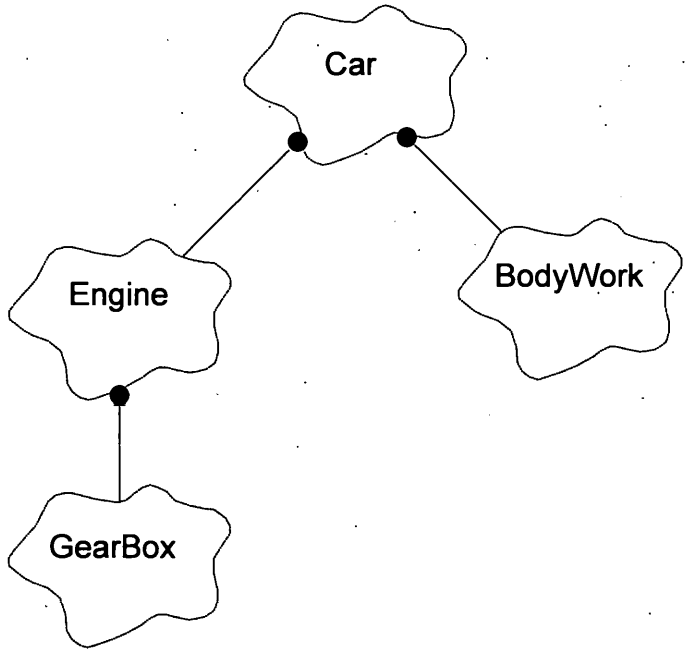


Figure 2.2 Example of Complex Object

From the above discussion, we can see that the object-oriented database system is a more suitable data repository in design environments [KIM90, AHMED91a] than previous generations of databases. In this thesis, an object-oriented database is used as a supporting data repository for a design environment. We will investigate how object-oriented databases can provide more efficient support in an engineering design environment.

2.4. The Design Environment and Databases

Central to an integrated design environment is its database management system. The database management system, often referred to as an Object Management System (OMS) in design applications, is used as the central integrating component of a design environment. The object management system handles all the information generated during the design development life cycle. Having a central database facilitates information sharing and can ensure data entered into the database can be validated to meet the design requirements or other integrity constraints.

To conquer composite design problems, people often decompose them into several smaller problems or modules which are easier to comprehend and manage. A module should be small enough for the developers to comprehend its functionality and it should be big enough to function independently. Modularity makes it possible for changes in one module not to affect other modules as long as the module interface remains the same.

In a design environment, designers work on their own workstations individually on a problem. However, communications between these designers are vital. A central database is provided to facilitate teamwork and information exchange. Project information and design data are all stored in the database. A system structure of such a design environment is illustrated in figure 2.3.

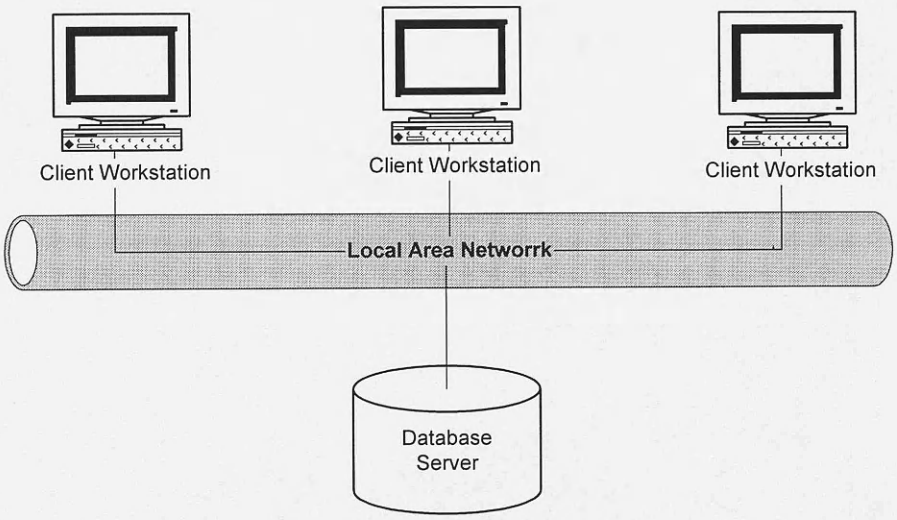


Figure 2.3 System Architecture in Design Environment

Generally, large design problems are not decomposed to a size that is suitable for one designer to work on. Rather it is firstly decomposed into smaller modules according to their functionality. These smaller modules are often organised as sub-projects within a big design project. Designers are assigned design problems within each sub-project. The solution to the original design problem occurs when the individual solutions to all the modules are put together.

To support such a project organization, the database server in figure 2.3 is actually composed of a hierarchy of databases [CHOU86]. These include a public database, project databases and private databases. This hierarchy of databases corresponds to the decomposition of the design problem. Figure 2.4 depicts the organization of the database hierarchy in a design environment.

The public database contains all the information about the whole project and also all the designs that are released from the project databases and ready to be integrated with other modules. The public database can be accessed by everyone working on the project. Before any design data is put into the public database, it must go through a validating process to ensure that it meets all the design requirement or integrity constraints of the project. All the design information in the public database is considered to be stable. The information can neither be deleted nor modified.

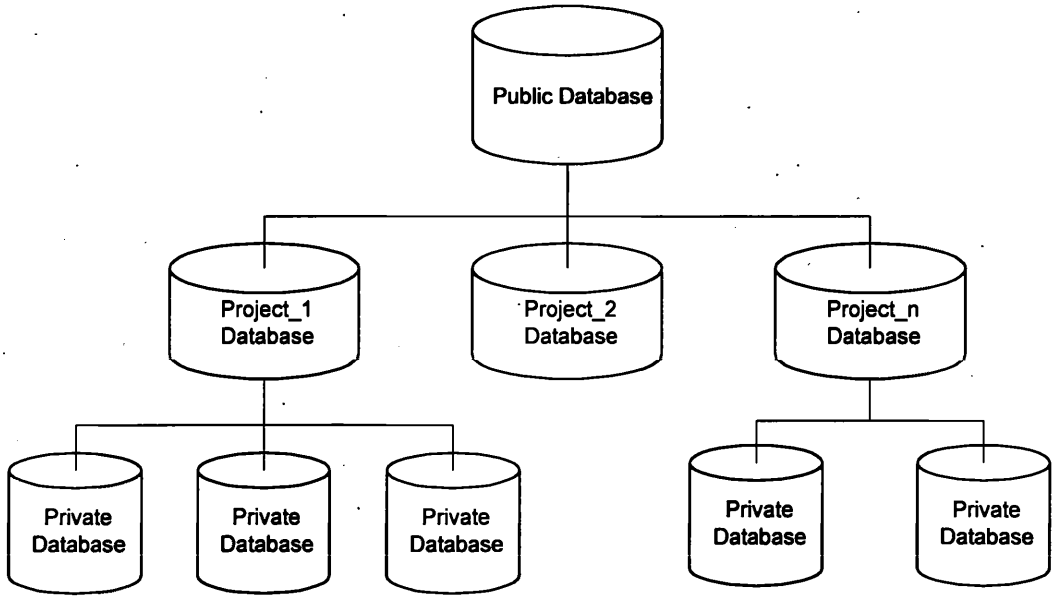


Figure 2.4 Hierarchy of Databases in a Design Environment

The project databases provide support for module development. They contain information about the modules and designs from individual designers that are ready to be used within the project database. Before the data is checked into the project database, it must be validated against the module design requirements and any other integrity constraints, e.g., other design objects referenced by the checked-in design object should also be put into the design database. Only people working on the same module are allowed access to the project database. Data in the project database cannot be modified but it may be deleted by the database administrator.

At the lowest level of the hierarchy are private databases. Generally private databases reside on the individual designers' workstations. This is where the designers perform much of their development work. The private databases can only be accessed by the designers who own it. The data in the private database is considered to be unstable. It may be deleted or updated by their owner at any time.

This organization of databases in the design environment suits the needs of different development stages. In the initial stage, design data is subject to frequent changes as designers experiment with different design ideas in their private databases. Because it is unstable, this data cannot be shared with other designers on the team.

When the design matures, the data is checked into the project database where it can be shared with other designers on the team. This data in the project database cannot be modified. If a designer wants to modify a design in the project database, he/she will

have to check out the design into his/her own private database and make the necessary changes. The modified design is then checked back into the project database as a new version. Versioning will be discussed in detail in chapter 3. When a module design matures, it can be then checked into the public database.

The check-in/check-out model suits the needs of the design environment on long transactions well. When a designer checks-in/checks-out a design data, he/she actually makes a copy of the data and installs the copy into the destination database. The designer then works on the copy instead of locking the object in the project/public database over a long period of time. This mechanism increases concurrent usage of design databases and facilitates collaborative team development.

2.5. Summary

Having examined the advantages of using databases in a design environment, the above analysis of the requirements for a design database revealed a number of problems with relational database technology. We then investigated what the object-oriented databases have on offer for design applications. We can conclude that object-oriented databases provide better data repository facilities for design environments because object-oriented database can [KIM90, AHMED91a]:

- Model and manipulate composite nested objects which allow successive refinement of composite objects.

- Allow the users to define and manipulate arbitrary data types. Object data representation is very flexible, and may be customized by users with little restriction.

- Represent and manage changes over time. This is an extremely important feature in a dynamically evolving design environment.

- Allow various semantic modelling concepts to be represented and manipulated, e.g., composite objects in an assembly-part hierarchy.

Using object-oriented databases to support composite design development is an active area of research. However, the remainder of the thesis is concerned with examining one of the mechanisms, seen as important for a database system, but which has received little attention in the context of engineering object databases - the use of the *View* mechanism. We begin by looking at versioning and configuration management mechanisms before defining and implementing a *View* mechanism for an integrated design environment.

Chapter 3 The Versioning Mechanism

In Chapter 2 we discussed the role of the database in an engineering design environment. In the first part of this chapter, we will justify the need for version management in a design environment and present a literature review of some typical version models in object-oriented databases. Configuration management is discussed in the context of composite object versioning. Finally we present our version model which we believe provides better support in composite object versioning.

3.1. Introduction

In an engineering design environment, an important requirement of its supporting database is to support incremental and cooperative design. To support such a tentative and iterative design process, the database management system must be able to capture the semantics of design evolution. Systems without a versioning mechanism keep only the most recent version of a design object. When there is an update to a design item, the old design is replaced by a new updated design. In many business applications this method is acceptable. However, in a design environment it is too rudimentary and is not acceptable because crucially the design evolution history is lost during such updates.

Versioning is seen as an important technique in managing evolution in a design environment [ZDONIK86, KATZ87]. The purpose of supporting versioning in

database systems is to capture the various states of a design object during its evolution. There are two types of changes to design objects as far as database management systems are concerned. The first is schema modification which concerns the changes in class definition of data models. The second is instance modification which concerns changes in the state of object instances.

3.2. The Changing World

The aim of introducing versioning into a database management system is to manage frequent changes to data and its schema in a dynamic world. Data in database systems are created to model only a subset of real world. When trying to capture part of the real world we would like it to stay still so we can capture a precise model of it. In reality this is rarely the case.

The data model in a database system reflects its designers' perception of the real world object. This perception reflects the designers' understanding of real world abstractions and conceptual organization. However, the designers' understanding as an abstraction may change as new insights into the application area arise.

The real world itself does not stand still either. It may evolve over time. The data in a database should adapt to the changes in the part of real world that it is modelling. The model of the real world in the database may change to better reflect the application domain, e.g., correction of errors or new requirements. From the database point of

view, there are two aspects that might be affected by these changes: the database schema and instances of schema.

3.2.1. Database Schema

In object-oriented databases, the schema defines the data structure of objects, e.g., their domain and sizes, as well as their behaviour. Objects of the same class have the same type of attribute and exhibit the same behaviour. The database schema in an object-oriented database defines classes and their inheritance structure. Objects are instances of these classes.

As the real world evolves the model in the database needs to adapt to these changes. This may mean that the database schema needs to modified. There are various approaches to schema modification [RODDICK96, LIU94]:

- ***Schema modification*** allows direct modification of a single schema. Schema modification will make any former specification obsolete.

- ***Schema evolution*** allows the modification of a database schema without the loss of existing data. Under schema evolution, existing objects must be converted to the new format and therefore existing applications are no longer compatible with the data.

- ***Schema versioning*** allows modifications to database schema without overwriting the existing schema, rather new versions of the schema are created. Versioning facilitates program compatibility by leaving the existing schema intact.

A database schema defines the contents and structure of a database. Objects in the database are created according to its schema. Upon modification of a schema, several aspects of the database may be affected and they are [ODBERG95]:

- Other parts of the schema. As database objects do not exist in isolation they inter-connect with other objects.
- Application programs. These programs are still expecting data organized according to the old schema.
- Objects in the database, which must comply with their database specification, i.e., its schema.

In object-oriented databases, modification to the schema is carried out by changing their class definitions or by creating or moving a class definition within the class inheritance hierarchy. [KIM90] summaries the taxonomy of schema modifications. [KIM90, MONK92, MONK93, BTSDYRTH92, RA95, ODBERG95] discuss schema modification and versioning in detail. Schema versioning is outside the scope of this thesis.

3.2.2. Database Objects

Another aspect that is affected by the changing world is the database object themselves. Analogously to schema changes, object modification can be achieved

through two different approaches. *Object modification* is the traditional approach to object updates, where the updated object replaces the old object and the old object ceases to exist in the database.

The second approach is *object versioning* where the updated object will be created as a new version. In a design environment, designers often want to try different approaches to a design. Ideally these different designs are grouped together therefore it is transparent that they will have some sort of connection. *Object versioning* plays an important role in such an environment as it is the object versions representing different design approaches which helps to group them. In many circumstances, modification to a design is reflected in updating the object in the database instead of its schema. Versions of an object represent different aspects of the same object and these representations are logically independent of each other [DITTRICH88].

Object versions are snapshots of an object over its evolution. Timestamped versions cannot model all of the rich semantics of versions [KATZ90]. In a design environment, a designer often has more than one design idea to fulfill a design requirement, for example, several alternatives to a given design specification. When the alternative designs are completed, revisions of prior designs are necessary because of new requirements, better ideas, or error corrections. Versioning provides the tracking of the evolution of a design object. It is crucially important to maintain versions of design objects because it provides traceability and the possibility of "going back" if a particular line of evolution does not work out. In a design environment,

versions are associated with a semantic that is known to the user and it is the user who decides which version to use.

3.3. Basic Version Concept

Versions are distinct snapshots of a design object in different states during its evolution history [AHMED91b, BEECH89]. There is a question when two instances of the same type are different objects and when they are merely different versions of the same object. Versions of the same object must share the same interface but may have different implementations [CHOU86, AHMED91b].

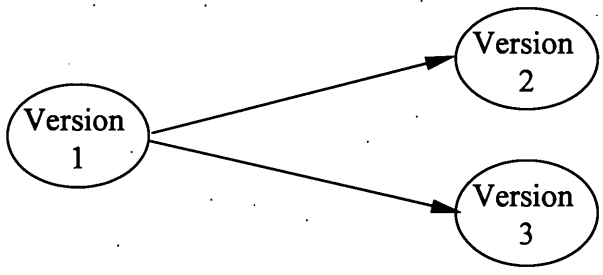
For a versioned object, each version must be uniquely identifiable through a *version identifier*. There are many ways of defining version identifiers, e.g., temporal or simple integer. The most popular one is to use the user defined unique version numbers.

A version identifier alone is not enough to fully describe the relationships between different versions of a versioned object [BILIRIS89]. Users may want to track the evolution history of a design object. Each version of an object is derived from its predecessor, except the first version. This kind of relationship between versions is typically called a predecessor/successor or parent/child relationship.

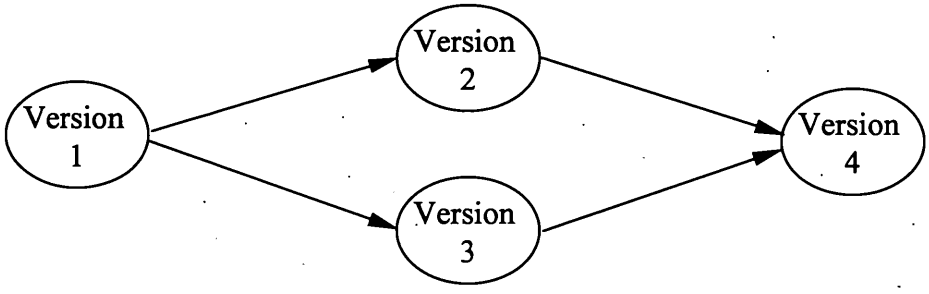
In a typical design scenario, designers often follow different development routes simultaneously starting from an initial design. The same designer might develop

several alternatives in order to study tradeoff, etc., at some stage in the evolution of a design. Alternative design versions may even need to be merged at some stage in the design process. This development scenario requires that the version history be a directed acyclic graph rather than a linear succession in order to capture the evolution history of versions of a design object. The version graph can capture different evolution patterns as shown in figure 3.1 [CARNDUFF94].

A. Linear Version Evolution



B. Alternative Version



C. Merging Version Evolution

Figure 3. 1 Version Evolution Patterns

Figure 3.1 shows examples of various version graphs. Figure 3.1a shows a linear version evolution history where versions are connected by relationships of a single

type, i.e., successor relationships. Here, version 2 is a successor of version 1. This means that version 2 has been derived from version 1. Figure 3.1b shows a two-level version evolution history where version 2 is an alternative of version 3. Figure 3.1c depicts the merging of versions where version 4 is created by merging version 2 and version 3.

A version branch in the version history may have one of the following implications:

- It represents an independent path of development
- It represents different variants of the component.
- It represents an experimental development which may be abandoned or included into the primary development at a later stage.
- It accommodates the fact that two developers were required to concurrently make changes to a component. In such a case the branch may be merged as soon as both modifications are completed. A merge combines the modifications that occurred independently in the two different versions into a new version.

For a large scale design project, a typical user scenario is that the design task is decomposed into several smaller projects. This makes it easier for the people involved to understand the problem and easier for the project manager to manage. The smaller projects form sub-projects within the top level project. Within each sub-project, a group of engineers are assigned to carry out the development work. As each subproject is completed, the resulting designs are assembled and integrated together to complete the final design project.

To facilitate object sharing and management of objects in a design environment, version objects often have states assigned to them to provide update constraints. There is no consensus on how many version states are needed. Some suggested two [DITTRICH88, TALENS93], while others recommended three [BEECH88, KIM90]. We adopt the three states approach because we believe it meets the needs of most design environments. The three version states are: the released, working and transient versions. The states of the versions reflect their stability in the database.

Another concept that is closely related to version states is the workspace model which provides a mechanism through which new versions are made available to designers working on a project. Workspaces are named repositories for design objects [KATZ90]. Each type of workspace is implemented in the same way, the only difference being the status of version objects residing in them and who can access these workspaces. Workspaces are organised in a hierarchic order, as *private*, *project*, and *public workspaces* [CHOU86], illustrated in figure 3.2.

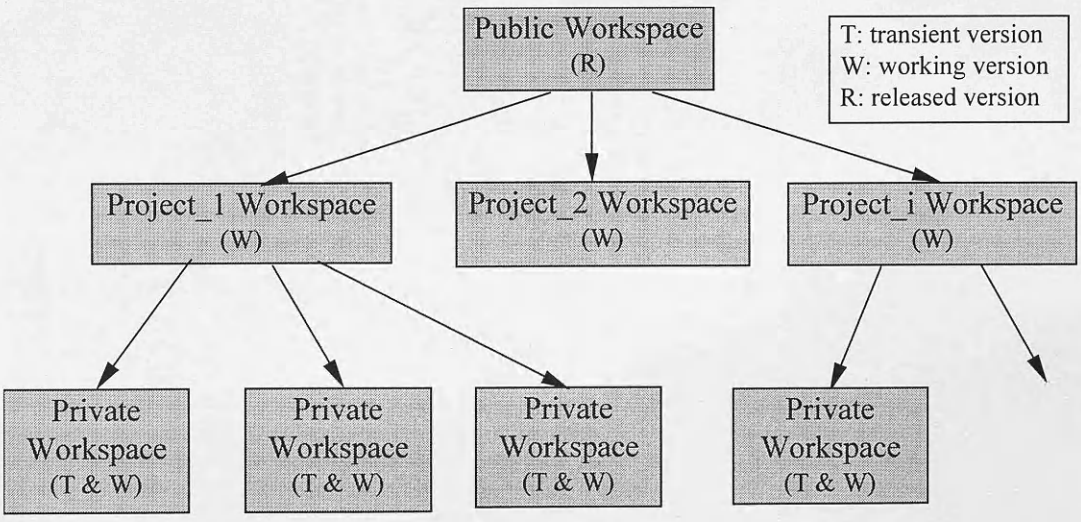


Figure 3. 2 Hierarchy of Workspaces & Distribution of versions

This arrangement of workspaces facilitates the development of large scale design projects. Large scale design projects are often too big for individual designers to comprehend fully. To make it humanly manageable, complex design problems are often broken up into smaller design problems. This hierarchy of workspaces meets the needs of such a design strategy. Designers have their own private workspace for creating, modifying and testing their design. At a certain point, the design is checked into project/public workspace where further development continues. Access control on workspaces guarantees that read and modification rights as well as rights to propagate changes to another workspace are restricted to appropriate project personnel.

All newly created versions are *transient versions.* They are owned by the designer who creates them. Transient versions are subject to frequent modification and may be even deleted by their owner. For this reason they are considered to be unstable and cannot be shared with other people. No new versions can be derived from a transient version. Transient versions reside in private workspaces where the designer performs design and validation work. The private workspace can only be accessed by the designer who owns it. This restriction enables the designer to carry out design work without fear of modification by other people. It also has the benefit that other people cannot reference unstable objects.

Working versions are promoted from transient versions. They are more stable than the transient version. *Working versions* are object versions that have passed the initial

design stage and are ready to be tested when integrated with other design modules. When a transient version is promoted to a working version a copy of the version is checked-out of the private workspace and checked-in to the project workspace. Working versions are considered to be stable. They cannot be modified but may be deleted by the project workspace administrator. There can be any number of working versions for the same design object in a project database. This arrangement enables designers within a project to experiment with different design alternatives. The project database is accessible to all the people on the same design project.

At the top of the workspace hierarchy is the public workspace. The public database holds *released* object versions. Before an object version is checked-into the public workspace it must go through a validation process to make sure that it meets all its design requirements. A released version can neither be updated nor deleted. All authorised users of the design environment have access to data in the public database.

This classification of workspaces allows the developer to be isolated from changes in other workspaces and from changes reaching higher level workspaces. Propagation of changes both out of and into workspaces are explicit operations and under the control of the developer.

3.4. Configuration Management

A composite object is a recursively defined aggregation of its constituent objects. Composite objects are configured by selecting individual component versions such

that participating versions of a component are consistent with each other [ZELLER95]. Different versions of a composite object have different configurations, each of which has various references to its component versions. An important requirement of configuration management is that it must ensure all the participating versions are compatible [AGRAWAL89].

Version management is the provision of a mechanism that can capture evolution in design artifacts. The aim of configuration management is to try to solve some of the problems pertaining to the evolution of design artefacts. These problems are caused by the lack of control and understanding of all the components that make up a design artefact. A further problem in the complex coordination of the product's evolution by its many developers. Configuration management controls the evolution of an object through the identification of the object's components and changes [HEILER91]. Configuration management provides a stable working context for changing the object.

Object configuration contains a set of references to specific versions of components. The process of selecting component versions is called binding. There are two kinds of bindings: *static binding* and *dynamic binding* [CONRADI96]. In *Static binding* versions of components have already been bound before any object is accessed. *Dynamic binding* is only performed when an object is actually accessed and the referenced objects may vary.

Configuration management allows a user to specify alternative configurations for a complex object through the selection of appropriate versions of its components.

There are two different approaches to version selection. The first approach relies on a labeling version graph. This requires that the user explicitly specifies which variant version is needed. The second approach allows the user to specify predicates on attributes. The attribute can be as simple as a version number that is associated with each of the versioned objects or as complex as a set of Boolean variables that specify some selection criteria.

The selection predicate approach provides a more general solution. It allows the designer to express selection of alternatives in a natural way and provides more flexibility and extensibility to adapt to different modelling requirements. When selecting component versions, the configuration management must provide a mechanism to ensure that all the selected versions are compatible in order to maintain configuration consistency. The underlying theories of selection predicates permit validation of consistent configurations to be expressed.

In a consistent configuration any modification to a component of a composite object may cause the consistency of the configuration to be broken. This is an issue that the configuration management must address. Many researchers have proposed various approaches on how to react to changes in a configuration. Some of these approaches are discussed in the following section.

3.5. Review of Version Models

Versioning is an important feature in third generation database systems and it has attracted a lot of interest from the database research community [SCIORE94, PARK95, CHEVAL90]. These research interests are divided into two broad areas. Schema versioning [RODDICK96, MONK93, AGRAWAL94] means different versions of an object can have different schemas. On the other hand, object versioning means different versions of an object have different values for some of the attributes. Schema versioning is an interesting research area but is outside the scope of this thesis. In the following section, we present several typical version models for object versioning.

3.5.1. Zdonik

One of the early version models for an object-oriented database was presented by [ZDONIK86]. The version model is based on the object-oriented concept using inheritance as its base for defining version capabilities for entities in an object-oriented database.

The Zdonik version model specifies a History-Bearing-Entity which is the basis of all the version control operations and attributes. As object versions evolve over time, a conceptual object is used to represent a design independent of time. A version-set is created for each conceptual object containing all its versions. The version model supports linear versioning as well as branching and consolidation. In the model a design object can be a composite object referencing other component objects that form a design hierarchy.

Zdonik recognizes the need for system controlled version percolation management. In an attempt to automate the version creation process, the model allows the user to define some references to components as version sensitive. Any change to these version sensitive components will cause new versions to be created in the upper level object. As it is not always desirable to propagate all changes at the lower level to all the higher containing objects, the concept of a *Slice* is introduced. A *Slice* is a set of versions that have been produced in a single transaction. A *Slice* also ensures that all the component versions are configuration consistent. Figure 3.3 shows an example of slices [ZDONIK86], each grouping represents a *Slice*.

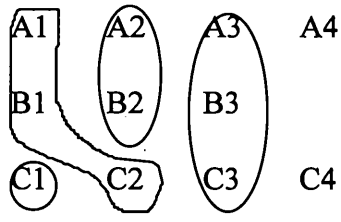


Figure 3. 3 Slices

The *Slice* concept is very similar to the group check-in/check-out model, where a set of related object versions are manipulated as a single transaction unit. The *Slice* is used as the basic unit of operation for any composite object. The model does not provide explicit support for referencing default versions.

3.5.2. Chou and Kim

[CHOU86] presented a version model in the context of a design environment. In this paper Chou and Kim considered a version model in a distributed development environment where a group of designers cooperate with each other in order to achieve a common design objective. The design environment is composed of a hierarchy of workspaces where each level allows various of degrees of sharing of information. These workspaces are the public, project and private workspace, as described in section 3.2.

Coupled with workspaces are version states where versions of different states reside in different workspaces (figure 3.2). Chou classifies versions into three states: *transient*, *working* and *released*. Versions of the same object in different states have different "*version capabilities*", e.g., which indicates whether they can be modified or deleted and by whom. Object versions are moved between workspaces by check-in/check-out operations. Version states are promoted while object versions are checked-into higher level workspaces. New versions are also created by the check-in/check-out operations. The workspace and version states provide a well-managed mechanism for cooperative design environments.

A composite object can reference other objects through static binding or dynamic binding via a context mechanism. The version model allows the user to specify a default version. This allows a more flexible dynamic binding in the design environment. They also proposed a change management strategy for composite object versions where a flag-based notification technique is used. Two time stamps are

maintained in each object version: change notification time and change approval time. If the referencing object's change approval time is later than the referenced object's change notification time, then the object is consistent, otherwise change propagation will be needed.

To avoid version proliferation, Chou limited the scope of notification only to objects that directly reference the changed version. As it is very difficult, if not impossible, to define a common policy for version propagation in a design environment, [CHOU86] left the designer with the responsibility of reacting to any changes in a configuration instead of automating the process. In Chou's model, only the designers of directly referencing objects are notified about the changes. The designers will then decide whether to react to the changes. If they decide to react to the changes then change notifications will be cascaded to the next level up. In [CHOU86], equivalent representation is mentioned but the author did not discuss its use in the version model.

3.5.3. Agrawal

[AGRAWAL91] presented a version model in the Ode object-oriented database. In this model, which is different from ones discussed in previous sections, all database objects can be versioned. The versioning capability is assigned to the persistent object instead of creating a separate version object. Therefore, the version model is embedded in the persistent object and there is no distinction between versioned object and the unversioned object.

The version model maintains temporal and derivation relationships between versions. The temporal relationship is a total ordering based on the creation time of object versions. Unlike some of the other version models, new versions can only be created by an explicit version creation function. Therefore, updates to an object do not result in new versions being created.

A logical object id is used to refer to the latest version of an object. This approach avoids using the *Generic* object as a dynamic binding to a particular object. The drawback of this approach is that the latest version does not always necessarily mean it is the most correct version especially when versions can have alternatives.

It is important the version models are extensible to best meet user needs. In Agrawal's mode, as versioning is an object property not a type property, it is difficult to add more features to the version model. The author did not discuss configuration management for composite objects, but in his early paper [AGRAWAL89] configuration management of versioned object was discussed in the context of an Ode object database. A transaction based model was proposed to ensure consistent configuration. All configurations are generated dynamically and no configuration is stored in the database.

We believe a configuration management system should provide a stable environment in a changing world. With dynamically generated configurations, the designer has no means of freezing a particular configuration which he/she might want to keep.

3.5.4. Ahmed

[AHMED91] proposed a version model for composite objects in CAD databases. The model classifies its properties into external features and internal assembly. The model explicitly defines internal assembly to identify components of a composite object and to describe their interrelationships. The external features are the non-structural features that are visible to other object.

The version model consists of three system defined types: generic, versioned and unversioned. The generic object represents the design object. It contains the invariant external features for the design object. Objects in the database can be versioned or unversioned. Unversioned objects are just like any other objects in a database without a versioning capability.

Versioned objects can contain three different kinds of attribute: invariant, version significant and non version significant. The invariant attributes remain the same across the version set of an object. Any modification of the invariant attributes will be visible to the whole version set and will not cause new versions to be generated. Unversioned objects can be converted into versioned object when needed. In contrast to Zdonik's model, updating a version significant attribute does not cause new versions to be created automatically. Instead the version significant attribute only indicates the updatability of a particular attribute in different version states. New versions can only be created by an explicit call to create function. The concept of version states used in the model is similar to that of Chou but with different names.

Version states can be promoted by explicit calls to the promote function as shown in figure 3.4. The invariant attribute cannot be modified within an object version.

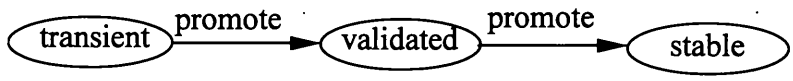


Figure 3. 4 Transition of version states

The model defines all composite aggregation as version significant. Therefore modification to any component will cause version propagation. The version model provides no comprehensive configuration management policy for controlling version proliferation in composite objects, although the use of design equivalents to avoid version proliferation is discussed.

3.5.5. Sciore

[SCIORE94] proposed a version model that places emphasis on the manipulation of object versions. The version model associates each set of design versions with a generic object. The generic object contains information that is common to all the versions of the same object. The version object can have two types of attribute: versioned and unversioned attributes. Unversioned attributes are visible in all versions. Updates to the unversioned attribute will be seen in the whole version set. The unversioned attribute is same as the invariant attribute in [AHMED91]. Mutating versioned attributes causes a new version to be created.

To further automate the process of version creation, the model allows its user to define some attributes as alternative attributes. Updates to these attributes cause alternative versions to be created therefore database designers can decide the semantics of an update operation i.e., whether it is a revision or an alternative. The version model tries to unify the various approaches used in temporal databases, historical databases and CAD/CASE databases by classifying versions into three levels - physical, conceptual and logical.

The version model distinguishes generic references from specific references. Specific references will bind to a particular version of the design object, i.e., static binding. Whereas, Generic references will be decided by a set of selection predicates, i.e., dynamic binding. These predicates are called dimensions. The use of dimensions simplifies queries to the versioned objects. The problem associated with such an approach is that the generic reference might return more than one version and it will be difficult for the user to decide whether to expect an object or a set of objects.

Sciore also explores the use of database views in the configuration management of a composite object. The view approach provides richer selection semantics than other approaches. However, the model fails to address the problem of a selection returning more than one object version for a configuration.

3.5.6. Summary of Version Models

We reviewed 5 different version models in the previous section 3.5. This review is by no means exhaustive. They are however representative of version models presented in the literature. From the review, we can see the trend is that more and more researchers recognise the need for version states in a version model to facilitate cooperative design activities and configuration management [CHOU86, OUSSALAH93, BILIRIS89, AGRAWAL91].

There are generally two approaches to version generation. One is an explicit call to a version creation function. The other is to define version sensitive attributes where any change in the version sensitive attributes will cause new versions to be generated. [AHMED91] presented a compromise approach. This version model allows the user to define version sensitive attributes but new versions will only be created by calls to the version creation function. We believe this approach not only complicated the version model but also limited the flexibility of the data model as the classification of attributes is used to limit their updatability in different version states.

Agrawal attempted a novel approach towards versioning. Instead of defining a version type as the basis for all other version objects, he embedded the version management capability inside database object - persistent object. This approach has the benefit that the user can create object versions as late as possible, i.e. whenever he/she needs it, unlike in [AHMED91] where unversioned objects have to be explicitly converted to the versioned object. Compared to the separate version object,

this approach restricts the extensibility of the version model as it is difficult to extend the version model to meet user needs.

3.6. Our Version Model

Our application aims to support a cooperative design environment for engineering design. Inevitably, there will be a lot of composite objects in such an environment. Therefore, a version control mechanism is crucial. From our previous discussions, it is essential that the version model supports the following features:

- Versioning of individual objects. This includes the maintenance of its evolution history, and the definition of default versions

- Change propagation in the composite object. This involves: how to react to changes in a lower level component and how far should the propagation go without resulting in generating unnecessary versions.
- Sharing of the design object in a design environment. This requires that we provide multiple level workspaces and version states associated with these workspaces.

3.6.1. The Versioning Data Model

Our version model is based on [CARNDUFF94] which consists of three different types of object: generic object, versionable object and descriptor object. For each set

of versions of the same design object, there is one generic object associated with them. The generic object is the conceptual representation of the design object. Attributes of a generic object are common to all versions of the design object and any update will not cause a new version of generic object to be created. For users who are not interested in versions of design object, he/she can simply reference the generic object without specifying any version specific information. The generic object will simply return the default version. Apart from default version number, the generic object also keeps a record of the last version number, the version evolution history and the methods for version creation.

The version evolution history records the derived-from relationships between versions. A version can be a refinement or alternative of its parent version. Further, a new version can be created by merging two previous versions. Therefore the version evolution history is a directed acyclic graph instead of a tree. The version graph consists of a set of version descriptors. The version descriptor has a one-to-one correspondence to the object version. The version descriptors keep a flag to indicate whether the corresponding version has been deleted. As some object versions may have other versions derived from them, it is not possible to delete them all. In our version model if the version is a non-leaf node in the version graph, the deleted flag will be set to true without actually deleting the object, otherwise the version will be deleted.

The versionable object keeps all the versioning information, a list of its component objects and also the configuration information if it is a composite object. Many

version models [SCIORE91, AHMED91] define the invariant attribute in their version model and these invariant attributes are visible throughout the version set. In our model, we put all the attributes that are common to the whole version set in a generic object. We believe this simplifies our version model without losing information.

Creating a new version is a complicated design decision. Updating the same attribute with a different value under different circumstances may have a different design implication. We consider the approach of using version sensitive attributes to create new versions as too primitive and restrictive. For such a complicated design activity, it is very difficult to define a common policy as when to create a new version and whether it is an alternative or a refinement version. We think it is more appropriate to leave the decision making to the designers. Only the designer knows the semantics of the update.

3.6.2. States of Versions and Workspaces

In a design environment, the designer's goal is to complete the design effectively. This implies that designers should not unnecessarily interfere with each other's work. But at the same time, the designer needs to communicate and coordinate efficiently. Our organization of workspaces supports this design activity well. The designers have their own workspace for carrying out their work. At a certain point, the design is made available to other designers on the team for further development and test.

The state of a version determines the stability of a particular version. In our model versions can be in one of the three states: *transient*, *working* and *released*. The basic idea behind this classification of versions is that unstable versions cannot be shared with other people as this may lead to an unstable configuration.

This provision of workspaces supports the sharing of objects among the design team. Objects and object versions are moved between workspaces as the result of check-in/check-out operations. The workspaces are arranged hierarchically as illustrated in figure 3.2. There have been various of levels of workspaces in the literature. Most researches agree that a minimum of three levels is needed to provide the necessary support [KATZ90]. In our model we classify the three levels as private workspace, project workspace and public workspace.

Versions in different workspaces reflect their states as well. Our approach to the classification of version states and workspaces is consistent with Chou and Kim's model. Each version state has a set of properties that define its behaviour. We will discuss the characteristics of each version state in the context of workspace. The state reflects that the version satisfies certain conditions.

At the lowest level of the workspace hierarchy is the private workspace. This is the private workspace for the individual designer, where he/she performs much of the design and design validation work. The private workspace can only be accessed by the designer who owns it. The states of object versions in the private workspace are transient. Transient versions in private databases are considered to be unstable

therefore they cannot be shared with other designers. They can be updated or deleted at any time by the owner of the database.

All newly generated versions are transient states. A new version is created when the designer checks out a copy of a design from a public/project database and checks it into the private database as a new transient version. No new versions can be derived from a transient version. The private workspace holds non-released designs that a designer is currently working on and any other information the designer wishes to maintain. When a design becomes stable and unlikely to be changed again, it can be checked into databases higher in the hierarchy and the state of the checked out version will be changed as well.

Figure 3.5 shows the movement of object versions between workspaces. A state transition occurs when an object version is checked into another workspace which represents the new state. Therefore, object versions migrate up the workspace hierarchy as their state is promoted.

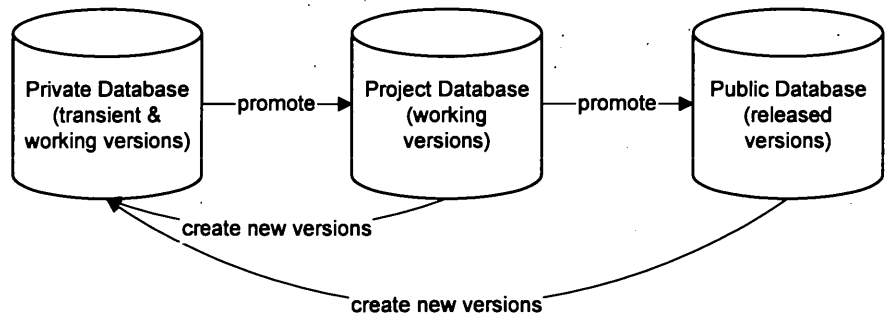


Figure 3. 5 Version movement between workspaces

Object versions checked out from the private workspace are checked-in to the project workspace, which is the next level up in workspace hierarchy. Before private versions can be checked-in to the project database, they have to go through a validation process to make sure that they meet the criteria for working versions. The project workspace is accessible to all the people working on the same project but not to people working outside that particular project. Object versions in the project database are working versions and they are considered to be stable. They cannot be updated but may be deleted by the project database administrator.

People working on the same project may check these objects into their private workspace and modify them. A modification results in new versions of a transient state being created. If any of the news version meets the design requirements, they may be checked back into the project workspace in order to allow access by other designers. There can be any number of working versions for the same design object in a project database. This arrangement allows people within a project to work on alternative versions.

At the top of the hierarchy is the public workspace. The public workspace holds released design objects. A released object can neither be updated nor deleted. All the authorised users in the integrated design environment have access to data in the public database. A working version can be promoted into the public database as a released version, if it passes the validation test for released objects. There can be any number of versions in the released state for any particular object.

3.6.3. Change Notification and Propagation

Composite objects hierarchically contain other objects as its components. If any of the component objects have been updated, the upper level object designer needs to know that this component has been modified and he needs to react to the changes in order to maintain the object consistency. Various attempts have been made to provide limited support for a system managed change propagation process [AHMED91, BEECH88, SCIORE94]. In our model we adopt the approach that the user decides when and how to react to changes in the lower level object as it is not always desirable to automate the version propagation process.

When there is a change in the component object, the change may be relevant to other objects referencing it, but it may also be likely that the change does not affect any referencing objects. For either the flag-based or the message-based change notification scheme, the upper level object will be informed irrespective of the effect of the change. In such a scheme, the designers are often inundated with many change notifications, some of which are relevant and others not.

Recognizing this problem, we have developed a new change notification scheme that can solve the above problem. Configuration constraints are used in our model to check if changes in the lower level object will affect other objects. For example, a *Car* object contains components *Engine* and *BodyWork*. Recursively, *Engine* and *BodyWork* have their own component, as shown in figure 3.6. For the composition in figure 3.6, it is the *Car* designer's responsibility to specify configuration constraints on its components *Engine* and *BodyWork*. In turn, it is the *Engine* and *BodyWork*

designers' responsibility to specify configuration constraints for their components. Further, it is the referencing object designers' responsibility to ensure that all participating components are consistent.

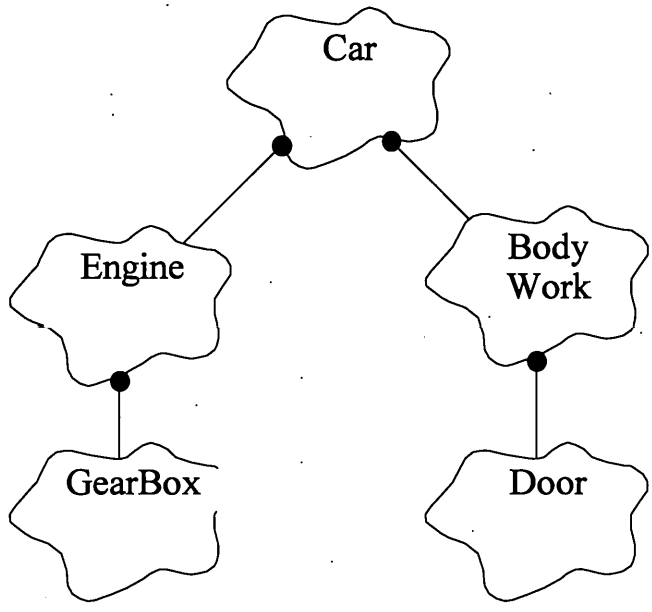


Figure 3. 6 Example of a Composite Object Configuration Hierarchy

Configuration constraints are a set of conditions set by object designers to check the respective components. Configuration constraints are specified in the version level. Therefore, they may vary in different versions. If any change in a component fails to meet its configuration constraints, then the object designer will be alerted about the change. The designer will consequently need to react to this change in order to accommodate it in the design. Otherwise, the designer will not be informed of the change. Therefore, the configuration constraints act like a filter for change notification. Only relevant ones are passed to the next level object. The use of configuration constraints also has the benefit that all the components have to meet their design requirements before they can be assembled.

For example, for a car *BodyWork* designer, it is more important to him/her that the *Doors* will fit into the B*odyWork*. If the size of window on the door has been changed, then to the car *BodyWork* designer this change is not significant. He/she may choose to set the configuration constraints in such a way that this sort of change will not be propagated back to his/her level. Therefore, the *BodyWork* designer can target the configuration constraints on aspect of the design which are important to him/her alone.

3.7. Summary

Versioning is an important feature in object-oriented database systems [CHEVL90]. In this chapter we established the requirements for a versioning mechanism in a design environment. In such an environment, cooperative and concurrent design is carried out. To support these activities, the supporting database is organised in a hierarchical form.

The provision of workspaces and classification of version states, as outline above supports composite object evolution in a natural way. It provides stable workspaces with control over isolation from external change and scopes of visibility for changes. The workspaces can support developers in an active development environment.

We discussed several version models for various environments. We presented our version model which is based on [CARNDUFF94]. The model improves change

management for composite design objects by using configuration constraints. Configuration constraints not only alert the upper level object designers (should there any change which affects the object) but also it guarantees that any component objects in a configuration have to meet their designated conditions before they can be accepted into an assembly. Configuration constraints are specified at version level. Thus, different versions of the same object may have different configuration constraints.

Chapter 4 Database Views and OQL

Object-Oriented views are more powerful than their relational counterpart [KIM88] because of the data models they are based on. In this chapter, we will discuss the semantics of object-oriented views and their roles in a design environment. A taxonomy of object-oriented views is presented. Query languages are an indispensable part of any view model. We will briefly introduce the ODMG standard query language OQL [CATELL97] which will be used as part of our view definition. Finally we will introduce our view model for a design environment.

4.1 Introduction

Views have been used for data protection and as a shorthand for queries in relational databases. They are an indispensable means of achieving logical data independence. It is recognised that views have an important role to play in object-oriented databases per se [AGRAWAL94, MARIANI93, MONK94, BRATSBERG92]. Object-oriented views should provide all the functions that are provided by relational views, plus some additional ones, which arise as a result of the higher expressive power of the object-oriented data model.

In a design environment, people often have different requirements for the data they need. The ability to provide multiple concurrent views of the same underlying information is vital to the usefulness of a database system, and means that application

programs can be written to a view of the data that is suited to that applications particular needs.

One of the main objectives of introducing an integrated design environment is to reduce the amount of redundancy in stored information, in order to maintain the consistency of information. It is inappropriate to maintain multiple copies of the same data at different abstract levels to suit different users' individual needs. It is much more desirable, where possible, to hold data in a single canonical form with different views provided to suit each end user's needs [BROWN88].

In relational databases views are exclusively defined by queries. However, in the object-oriented world, there is no agreement on how object-oriented views should be defined. Various view models have been proposed for object-oriented databases [ABITEBOUL91, BERTINO92, HEILER91, SCHOLL91]. A view mechanism should allow programmers to restructure base objects and modify their behaviors. In the object-oriented world, it should be possible to use views to provide different interfaces to the same object as a general abstraction mechanism [DAYAL89]. Generally, there are two main approaches to the definition of object-oriented views, depending on whether or not the view classes create new view objects:

- ***Object-generating views***: instances of view classes are new objects with their own object identifiers (oids). These objects are generated as a result of the view query.

- ***Object-preserving views***: no new object is created for the view class, instead existing base class(es) objects that satisfy the view query are regarded as instances of view classes.

One of the reasons for this diversity of OODB view models is due to lack of standards in the object-oriented world. Some researchers believe that the view classes should be integrated into their base classes inheritance hierarchies [ABITEBOUL91, HEILER90], to enable the view classes to use as much information from the base classes as possible. Others believe that the view classes should be orthogonal to the base classes [BERTINO92, SCHOLL91] to achieve greater data independence.

The integration approach provides a uniform structure for both view classes and base classes as semantically some view classes naturally form sub/super class relationships with their base classes. This approach enables view classes to take advantage of all the information contained in their base classes. One of the problems associated with the integration approach is how and where to position view classes in the base class inheritance hierarchy without affecting the semantics of base class inheritance structures. Integrating view classes into base classes inheritance also exposes views to the effect of changes in the base classes schema.

Proponents of the separation approach argue that views serve as interfaces to base class objects and the separation will result in complete logical data independence [KIM95]. In our view model, view classes are used to provide multiple levels of

abstraction to base classes. Therefore, we believe it is appropriate to keep view classes orthogonal to base classes.

Object-preserving views allow view updates to propagate to the corresponding base class objects unambiguously, since the view objects have the same oids as the base class objects. On the other hand the *object-generating view* provides a more flexible approach for defining view classes since view classes are not limited by the structure of their base classes. In the *object-generating view*, view updates are not always possible, as there is no guarantee that a view object always corresponds to a single base object.

View updates are an essential requirement of object-oriented views [ATKINSON89]. Many researchers believe that the support of the *object-generating view* and the unconstrained updatability of views are conflicting requirements that cannot be simultaneously met [MOTSCHNIG96]. Later in this chapter, we will introduce a view model that allows view updates in an *object-generating view*. Table 4.1 compares a few view models. These models are compared by considering:

- **What data model they are based on.** Many of the view models are designed for a particular object model and the data model plays a key role in defining views.

- **How they are defined.** Some of the view models are defined exclusively by queries. Others use a query language as well as features from their object data model in order to define views.

- **Whether they are part of the base class inheritance hierarchy.** This is a fundamental issue in the definition of a view model. Integration provides well integrated information while separation has the advantage of higher logical data independence.

- **Whether it is object-generating view or object-preserving view.** The salient point here is whether new objects are generated as result of running a view, which has considerable impact on view updatability.

The comparison in table 4.1shows the different approaches adopted by different view models. It is based on four representative view mechanisms from the literature.

View Model	Data Model	View Definition	Relation with Base	View Object	View Update
ABITEBOUL91	O2	query language, inheritance and overloading	Sub/super type and behavioural generalization	new oid is assigned	Define update method when defining the view to avoid update anomaly
BERTINO92	General	extended predicate calculus query language	Derived type	user decides whether to assign oid	Not addressed
HEILER91	FUGUE	set_oriented algebra	Sub/super type	view object uses the same oid as that of base object	Not addressed
SCHOLL91	COCOON	type system & query language	Derived type	same oid if there is a one to one correspondence between view object and base object. Otherwise, new oid is assigned to view object	Since view objects use the same oids as their base class objects, updating view objects is effectively updating base objects

Table 4. 1 Comparison of different View Models

4.2 Taxonomy of Object-Oriented Views

The aim of our project is to develop a view mechanism that can provide efficient support for a cooperative design environment. Earlier in the thesis we discussed the requirements for integrated design environments. Before going into the details of our view mechanism, we present a taxonomy of the view model. This taxonomy only considers the semantics of view operations, not any implementation details. Because there is a lack of consensus on the definition of object-oriented terminology, we firstly define the terms used in our taxonomy to avoid any confusion.

4.2.1. Definition of Terms:

Abstract Data Type: defines the interface to a data abstraction without specifying implementation details. For reasons of brevity, we use 'type' instead of 'abstract data type' in the following passages, unless otherwise indicated.

Objects: are instances of abstract data types.

Classes: are collections of objects that belong to the same abstract data type. A class can be derived from existing classes using class inheritance. The newly derived class is the sub-class of the parent class. A class hierarchy represents he relationships between parent classes and sub-classes. A class defines the object's internal state and the implementation of its methods.

4.2.2 View Taxonomy

Our taxonomy of views is similar to that in [SCHOLL91]. We extend it to handle object-versioning. The view classes can either be populated by objects that already exist in the database or by newly created ones. Since object-oriented views should at least fulfill the functionality of relational views [MOTSCHNIG96], set-oriented algebra is used to define our view semantics where people can see the relevance between the two is clear. Figure 4.1 shows our taxonomy of views.

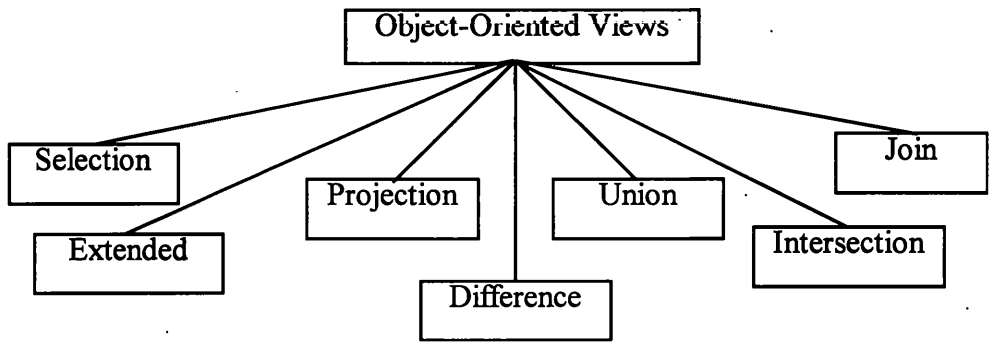


Figure 4. 1 Taxonomy of OODB Views

Before going into more detail about our view taxonomy, we identify the possible basic modifications that might happen between a view class and its base classes:

a) A view class may use different attribute names from that of its base class, e.g. a view class may change the attribute `address` in its base class `Student` to, say, `home_address`;

b) A view class may use different method names from that of its base class;

c) A view class may transform the value of its base class attribute, e.g. convert inches into centimeters.

d) A view class may have more/less attributes/methods than its base classes;

e) A view class may transform values returned by its base class methods, e.g. convert temperature from Fahrenheit to Celsius centigrade.

f) A view class may materialize the return value of a base class method and store it as an attribute value.

g) A view class may overload the methods of its base class;

h) A view class may only use part of an aggregate attribute as its attribute. For example, a `Person` class contains an aggregate attribute `address` which itself is another object. If the user is only interested in the nationality of the person, then, instead of listing their full addresses, the view class only displays their `Country` attribute in the `address` object.

- **Selection View** [SCHOLL91]

A *selection view* returns a subset of all instances of its base class satisfying the selection predicates. For example, the user may create a view `new_student` that returns all the first year students. For a composite object, a *selection view* should be able to retrieve component objects from its base class(es) without the user specifying

how. A *selection view* only applies to a single base class. For versioned objects, all versions of the base class that meet the selection predicate will be instances of the view class.

- **Projection View** [SCHOLL91]

A *projection view* returns the whole set of object of the base class with some attributes hidden in the view class. For *projection view*, any method that uses the hidden attribute should also be inaccessible. As result of projection, a new type is created for view objects. Projection only applies to a single base class. For aggregate attributes, a *projection view* should be able to either hide or retrieve the aggregate attribute as whole.

For versioned objects, projection is at the class level. Therefore, all versions of the base class object will be included in the view class. However, as a result of projection, some object versions in the view class may lose their distinctive attributes.

- **Extended View** [BERTINO92]

An *extended view* contains attributes which are not part of its base class(es). The extended attributes only exist in view objects and cannot be derived from its base class(es). The extended attribute may be another object which forms an aggregate attribute of the view class. An *extended view* allows users to augment the definition of its base class(es). A new type is created for *extended view* objects. For versioned objects, each version of the base class instances will also be versioned in the *extended view* class.

- **Join View** [SCHOLL91]

A *Join view* returns a matching pair of objects from the involved base classes. The involved base classes must share a common attribute. The result of a join is a new class that includes attributes and methods from both base classes. For versioned objects, only those versions that can be joined are included as instances of the view class. Semantically, the join view produces the same type as multiple inheritance.

- **Union View** [SCHOLL91]

A *Union view* class contains instances from both base classes. The involved base classes must be unionable, i.e., they must have a sub/super class relationship in the class inheritance hierarchy. The view class contains attributes that are an intersection of the two sets of attributes of the base classes. Semantically, a union view class is a super class of its base classes.

For the versioned object, the versions of view class instances are the sum of both base classes versions if the new instances of the view class do not have duplicate values. For example, if base class A has 4 versions and base class B has 2 versions and each view object has unique values, then the view class will have 6 versions.

- **Intersection View** [SCHOLL91]

An *intersection view* returns all the objects that are members of both base classes. An *intersection view* produces a sub-class of its base classes. The view class contains

attributes and methods that are unions of attributes and methods of both base classes. An *intersection view* class is a sub-type of its base classes.

The resulting type of *join view* and *intersection view* are very similar. For a *join view* the two base classes must share a common attribute to be joinable. The result of an *intersection view* includes objects that are members of both base classes. This implies that the base classes must share a common super-class in their inheritance hierarchy. All versions of base class objects that satisfy the intersection condition will be visible in the view class.

- **Difference View** [SCHOLL91]

A *difference view* returns all the objects that are members of the first base class but not members of the second base class. The semantics of a *difference view* requires that the base classes must have a common super-class and those members that belong to both base classes are filtered out. The view object is of the same type as that of its first base class. The view class is sub-class of the first base class. All versions of the first base class object that satisfy the difference condition will be visible in the view class.

4.2.3. Semantics of View Update

View update is a desirable feature for all view models [SCHOLL91, MOTSCHNIG96]. In an object-oriented database, because of the data encapsulation enforced by the object data model, it is not desirable to use query languages to update

view objects directly. This will infringe the encapsulation of the data model which is one of the basic principles of object-oriented theory. We believe it is more appropriate to an object-oriented database that updates are handled by methods of a view class rather than by query language.

- ***Views that Modify Attributes of Base Class***

When a view class updates an attribute of its base class, its update method should know how to propagate the update to its corresponding base object correctly. If there is more than one base class object involved in the view, the view class update method should be able to propagate the update to the correct base object. Any change made in the view is effectively updating the view object's base objects. If the view is trying to update an extended attribute which only exists in the view, the change should only be confined to the view objects and never propagate to the base objects.

If the updated attribute in the base object is an aggregate, the corresponding object should be updated correctly. If the base of a view is another view, the view update should propagate to the appropriate base class object. All the view updates are under the control of the version manager so an appropriate version may be created. This is another important reason why we do not allow direct updates from a query language as it is difficult if not impossible to enforce version control and object encapsulation.

- ***Views that Modify Methods of its Base Class/View***

It is possible that a view class has a different set of methods from its base classes. This can be because the view class has more methods than it base class or the view

class overloads some of its base class methods. Because methods belong to the data type not the individual object, changes in view class methods have no effect on individual object versions.

- ***Views that Insert Objects***

Sometimes a view class may create new objects. Because view classes are virtual, their instances should not be stored in the database. Therefore, a new view object is actually a reflection of new objects being inserted into its base class. When new objects are created, the view class should be able to insert these objects into the appropriate base class if more than one base class is involved. The creation of new view objects must be under the control of the version manager so the newly inserted object could be a version of an existing base class object.

If the base of a view class is another view, the insertion must propagate until the new objects are inserted into the appropriate base class. For composite objects, when they are inserted their aggregate attribute must also have new members inserted if they do not already exist.

- ***Views that Delete Objects***

A view class should be able to specify methods that delete objects. When deleting objects from a view class, the view class delete method must correctly remove objects from the appropriate base class. The delete operation must be under the control of the version manager and must comply with the semantics set out in Chapter 3, e.g. only versions at the leaf of the version graph get deleted, and so on.

4.3 ODMG Object Query Language (OQL)

OQL is a part of the Object Database Management Group (ODMG) standard for object-oriented database management systems. The ODMG is a consortium of object-oriented database management system (ODBMS) vendors and interested parties. The primary aim of the standard is to provide a set of standards that enable portability of customer software across ODBMS products.

The ODMG standard includes an Object Model, an Object Definition language, an Object Query Language (OQL), and Language bindings to C++, Smalltalk and Java. The object model in the ODMG standard is built upon the Object Management Group (OMG) standard [OMG97] which provides a common architectural framework for object-oriented applications. The standard also involves other existing standards, e.g., SQL-92 and the ANSI programming language standards to define a framework for application portability between object database systems. In this section, we briefly discuss the query language - OQL. For a detailed introduction of the standard ODMG 2.0 please refer to [CATELL97].

In an effort to provide a query language for object databases which is similar to the all familiar relational query language SQL, OQL is defined as a standard query language for object-oriented databases. OQL is an SQL like high level declarative query language that provides a rich environment for the efficient query of database objects. The OQL is a superset of the SQL-92 SELECT syntax. Therefore most SQL

SELECT statements can be used in object databases. To take the compatibility issue between the two query languages one step further, the ODMG is working with the ANSI X3H2 committee, which is defining the SQL-3 standard, with the aim of converging OQL and SQL-3.

To handle objects in object databases, OQL also includes object extensions that include: object identity, complex objects, path expressions, operation invocation and inheritance. To maintain the encapsulation of the object data model, OQL does not define any update operator but uses update operations defined on database objects. An OQL SELECT statement will return a collection of objects with or without object identities depending upon the way the query is specified. Should the query return objects with their identity, the user can then invoke operations defined for the object.

An object database view is not defined in the latest ODMG standard ODMG2.0 [CATELL97]. One of the aims of this thesis is to define an object-oriented view that uses OQL as part of its view definition and to explore the advantages offered by the richer semantics of object data model. In the relational world, a view is a query. In the object-oriented world, however, the situation is much more intricate because of the more complex model employed. We defined an object-oriented view which consists of a type and a query. The type defines the intent of the view and the query specifies the extent of the view class. The design andimplementation of our view model will be discussed in Chapter 7.

4.4 A Model of a View Mechanism

4.4.1. The aims of the Model

Earlier in this thesis particular problems and requirements of a design environment that support object versioning were identified and discussed. Now we describe a view model designed specifically to support design activities in an integrated design environment. We developed a view mechanism with the following explicit objectives in mind:

- To provide a flexible mechanism capable of supporting design interactions at different levels of abstraction that are suited to the individual designer's needs and support cooperative design activities in an integrated design environment.

- To provide facilities that allow users to tailor the design environment in a controlled fashion to suit a designer's individual requirements;

- To use the view mechanism as a management tool for controlling access to design data by restricting the data that each user can access, and by explicitly defining operations that different groups of users can perform on particular data.

- To use this model to assist the integration of new tools into a design environment by providing abstract interfaces through which such tools can access design data in a design environment.

- To use the view as a mechanism that facilitates controlled information sharing between teams of designers, maintaining the integrity of design data and also provide a unified versioning framework throughout a design environment.

4.4.2. The View Model

To achieve the above objectives in our view model, we cannot limit ourselves to existing view approaches. New objects are needed to provide the extra modelling power required by design environments. Meanwhile we want to maintain the convenience of an *object preserving view* where the user does not need to worry about creating a view object schema and view updates. Therefore, it is our intention to combine both *object-generating* and *object-preserving* strategies in our view model in order to achieve the maximum flexibility required by an integrated design environment. The definition of our view mode (figure 4.2) is composed of two parts:

- A view schema definition. This specifies the schema for view objects. The users can either use existing base class objects as view objects, i.e. an *object preserving view*, or they can define a new schema for a view class. A view schema is defined in the same way as its base class. This will invoke new objects being generated by the view class. The view schema specification defines the intent of a view.

- A view query definition. This specifies the condition whereby the base class objects can be selected to initialise view objects. The view query definition defines the extent of a view class.

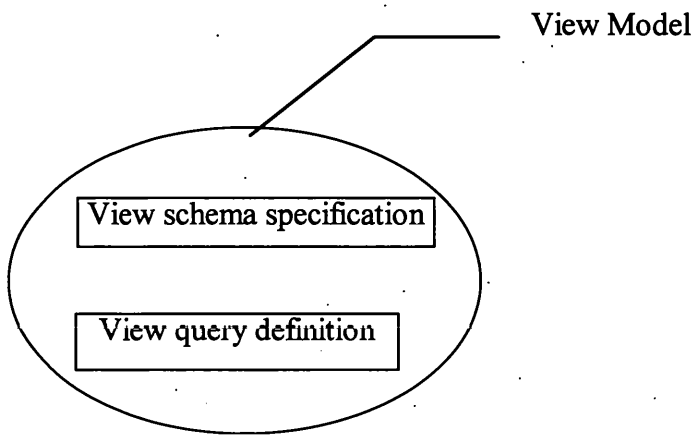


Figure 4. 2 View Model definition

The query language, OQL, used to define the extent of our view is not computationally complete. Therefore, it is difficult to use it to define a view object's behaviour without extending the query language. The advantage of separating the definition of view intent and extent is that it allows the user the freedom of using either existing objects or creating new view objects, should the user need it. The generated view objects are defined just like any other base classes in the database. Therefore, the objects behave exactly like any other database objects such that we can take full advantage of the richer semantics provided by the object-oriented data model.

The view query decides the number of instances in a view class. The view schema definition decides the characteristics and behaviour of view objects. The view designer may choose to use an existing schema as view schema. In this case, no new

object is created by the view class. This is called an *object-preserving view*. In an *object-preserving view*, there is no need for the view designer to specify how to update the base object through the view as the view object is the same as its base object. If the application requirements cannot be met directly by base objects, the view designer can define a new schema for the view class which uses base class objects to instantiate view objects. In this case, new objects are generated by the view class. The new view objects are not stored in the database. They are dynamically created when accessed. Therefore they reflect any changes in the base objects.

For the *object generating view* in our model, while designing the view schema the user needs to specify how to instantiate view objects from base objects. Every view object maintains the object identifier of its base object. This will allow view updates to propagate to the correct base object even if the view objects have more than one base objects. The view designers can explicitly define update methods for view classes by specifying which base object need to be updated for a particular update operation. Figure 4.3 shows the relationship between the view object and its base objects.

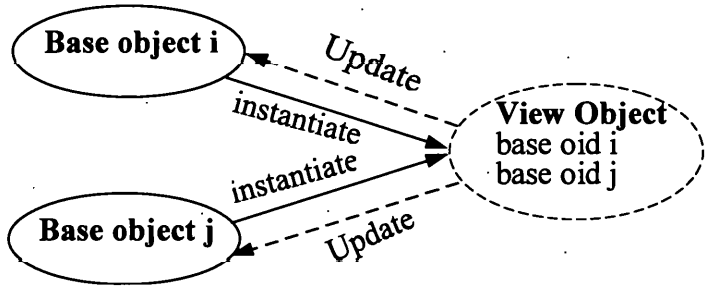


Figure 4. 3 Relationships between the View Object and its Base Objects

4.5. Summary

In this chapter we presented a taxonomy of view semantics. The taxonomy specifies the full semantic requirements for a view model. The pros and cons of an *object-generating view* and an *object-preserving view* were discussed. We developed a view model which consists of two separate definition schemes that allows the model take full advantage of the object-oriented data model and gives us the freedom to choose either an *object-preserving* or an *object-generating view* in a single view model. To achieve logical data independence, the view class is orthogonal to the base class inheritance hierarchy. We contend that that our view model is able to attain the objectives stated earlier in this section.

Data encapsulation is maintained in our view model to ensure that other user special requirements, e.g. version management, will not be violated by a direct query update as in a relational view.

Chapter 5 Materialization of The Object-Oriented View

The object-oriented paradigm provides a more powerful view model than its relational counterpart. From the discussions in chapter 4, we can see that the object-oriented view model can be used as a suitable means to provide multi-level abstractions in a design environment. Views have been recognized as an effective mechanism to virtually restructure the database schema [ABITEBOUL91, BERTINO92].

In relational databases, views are typically defined by stored queries. Each time a query is issued against the extent of the view, it is translated into a query against the view's base tables [DATE95]. Although object-oriented views differ from relational views, they will still inevitably impose some performance overhead because of the recomputation involved upon accessing the view.

5.1 Introduction

View materialization has long been used by the relational database community as a means of performance enhancement. Materialized views store the extent of the view in the database as opposed to recomputing them upon demand [Gupta93]. One of the basic requirements of views, whether they are materialized or not, is that they must reflect changes in its base classes. This means that view objects must be consistent

with their base class objects. This requirement presents a challenge to materialized views, as their instances are physically stored in the database.

The objective of maintaining a materialized view is to keep the view objects consistent with their base class objects with the least maintenance overhead. The maintenance overhead includes the re-evaluation and re-materialization of the materialized views when inconsistency between a view and its base occurs. The question to answer is when to evaluate and how much to update? Many techniques for improving the efficiency of relational view maintenance have been reported in the literature [LU95, GUPTA93, GUPTA95, PIROTTE94, STAUDT96, COLBY96].

Although the object-oriented data model is different from the relational data model, we can still learn some view maintenance techniques from the relational database community. Currently materialized object-oriented view have not received much attention from object database community. MultiView [KUNO95a] is the only object-oriented view model that we have come across, discussing the issue of object view materialization.

View maintenance techniques are classified into two categories depending upon when the view is refreshed. If a view is refreshed within the transaction that updates its base, it is called *immediate view maintenance* [COLBY96]. Otherwise, a view can be refreshed periodically or on-demand when certain conditions arise. This is called *deferred view maintenance* or *lazy view maintenance*. The immediate view maintenance approach increases the overhead of updating the base as the view needs

to be updated at the same time. This overhead increases with the number of views and their complexity. On the other hand, deferred view maintenance may increase the view access time. This occurs when views are accessed. Each view object has to be checked against their base objects or even recomputed if necessary to maintain view-base consistency.

5.2 Object-oriented View Materialization

The view mechanism offers greater flexibility in organising schema and managing data in database systems. Each time a view is accessed, its extents will be re-computed. The recomputation process will induce some performance overhead. View materialization is a well-known optimization technique in relational database systems [HANSON87]. View materialization is used to store the extents of a view class in a database. Because the view extents do not need to be computed upon access, access to materialized views may be substantially faster than non-materialized views. However, we must maintain the consistency between the view class and its base classes upon updates to bases.

MultiView [KUNO95a] supports view materialization in object-oriented databases. MultiView uses an object slicing technique [KUNO95b] to define its object data model. MultiView adopts an object-preserving approach where view objects are the same as base objects. In MultiView, each object is composed of two parts: a conceptual object that decides the type of the object, and an implementation object which is used to represent an object's membership in a class. An implementation

object can be associated with more than one conceptual object. Therefore, an object can gain membership to more than one class which means that an object can gain or drop a type dynamically.

In [KUNO95a], view-class consistency upon update is achieved by propagating updates to both the base and view class at the same time. This is basically an immediate view maintenance strategy. To achieve this simultaneous update to both base and view class each view class is registered with those base classes whose updates might affect the view class. When a base class is updated, its registration table will be processed. Every view class that has an entry in this table will be updated as well to maintain the view-base class consistency.

As pointed out earlier in this chapter, the immediate view maintenance incurs an update overhead when each view class is processed. Although view materialization is based on the assumption that the materialized view will be used, it may happen that a view is not accessed between two updates to a base class. In such a circumstance, the update overhead is not justified and is not necessary. The MultiView view maintenance approach is based on the so called object-slicing technique where an object can gain or drop a type dynamically. Therefore its materialization approach cannot be applied to other object-oriented data models.

[CARNDUFF93, KEMPER94, KEMPER91] presents a strategy for function materialization in object database. We believe function materialization can be part of view materialization as a view designer may decide to add an attribute in the view

class to store the value returned by a base object function. [KEMPER94,91] exploits object encapsulation in his strategy where objects can be updated through a designated channel. Associated with these update methods are triggers which will invalidate the materialized value. A table is created to keep track of the relation between an update method and a materialized function value.

In [KEMPER94, KEMPER91] the user can choose either immediate or deferred maintenance. However, extra effort and overhead are needed to maintain the update table which is crucial to the maintenance of materialization. The table will inevitably grow larger when many materializations take place and the maintenance overhead will increase.

5.3 Object Identity and View Materialization

In this section, we present our view materialization technique for object-oriented databases. Our object materialization strategy enables efficient view maintenance and is not specific to any particular object data model. Thus it can be applied to other object-oriented data models.

Efficient view maintenance is achieved by incremental maintenance [GUPTA93], in which only the changed base objects are evaluated and computed, without extensive evaluation and full recomputation of the whole view class. View maintenance happens only when base class object updates occur. When an update on base classes occurs, we need to know which object has been updated so that the appropriate view

object will be refreshed. It does not affect the base class in any way, e.g., the base class update method does not need to trigger any function.

We adopt the deferred view maintenance approach where views are only re-evaluated on-demand. The benefit of this approach is that people do not use the view will not pay any penalty, i.e., those not using view will not have to worry how to keep them up to date.

Updates to base classes have considerable impact on the performance of materialized views. To minimize the impact on performance of materialized views: on the one hand we adopt an optimized view materialization technique whereas on the other hand we only want to materialize those views whose base classes are in a stable state, e.g., not subject to frequent changes. Transient versions in our object database are considered to be unstable and subject to frequent update operations. We make the restriction that only those view objects based on working versions and released versions can be materialized. This limitation means that sometimes our materialization is a partial one.

5.3.1 Taxonomy of Base Class Update Operations

When update operations are performed on a view's base class, we would like to know how it affects view objects based on it. However, update operations on a view's base object do not always have the same effect on the view class. For example, we define a view 'luxury cars' which has the extent of all the cars valued over £20,000. Now

suppose two update operations are performed on Car1 and Car2. We increase the price of Car1 from £19,000 to £20,500 and reduce the price of Car2 from £21,000 to £19,000. Car1 was not in the view class. After the update Car1 is inserted into the view class, while the effect of the second one is to remove Car2 from the view class.

We classify update operations into the following categories and discuss what effect these operations have on the view class:

- **Insert:** this operation adds new instances into the base class. If the newly added objects meet the view query predicate then another *insert* operation will be performed on the view class, otherwise, no action will be taken.

- **Delete:** this operation removes an instance from the base class. If this instance was involved in the view class then it is removed from the view class as well, otherwise, this operation has no effect on the view class.

- **Set:** this operation updates the value of the base class attribute through the update method of the class. We may classify base class attributes into two categories:

  (i). relevant attributes are those used as part of the view class properties or as part of the view query predicate, and

  (ii). irrelevant attributes, e.g. base attributes projected out in the view.

- **Promote:** version status plays some role in our view materialization. We stated that view objects based on transient versions cannot be materialized. If the transient version is promoted to a working or released version, then we need to *insert* it into our materialized view class.

If the updated attribute is a relevant attribute then the following scenario will lead to different operations being performed on the view class depending on the attribute's role in the view:

(i) If the attribute is part of the view property then this view object needs to be re-materialized.

(ii) this attribute is used as part of the view query, if its value does not cause the view predicate to become false, then the change will be propagated to the view object, otherwise, the corresponding view object will be inserted/deleted from the view class.

For the irrelevant attributes, as its name suggests, these attributes are not involved in the view class in any way, therefore, changes in these attributes will have no effect on the view class.

In [CERI91] all the update operations on the base relations that affect the view are translated into *insert/delete* operations. Since our view model is object-generating, if we adopt the same approach, the object ids of the affected view objects will change which is undesirable.

We have the restriction that only view objects that are based on working versions and released versions can be materialized for performance reasons. For these two types of object versions, updates will create new versions rather than change the base version. Therefore, the *set* operation is not considered in our view maintenance. In our database, only the delete and update operations are considered on the base classes and they are translated to an *insert/delete* operation on the view classes of the materialized view.

5.3.2 The View Maintenance Manager

A view maintenance manager (VMM) has been developed to act as a mediator between view classes and their base classes in order to maintain their consistency. A VMM keeps information about views and their base classes in a database. When a view is created, it registers with the view manager together with its associated base classes.

The base class has a flag indicating whether a view has been derived from it. The base class will send a message to the VMM when an update operation occurs if the view flag has been set to true. The message contains the information of the base class id, the updated object id, and the type of update operation. If the updated base class id is registered in the VMM as an associated base class, the VMM will keep that information, otherwise the VMM will set the view flag in the base class to false.

Generally there are two different materialization strategies for the timing of view updates:

- *Immediate mode*: the view update will be carried out immediately after a base class update.
- *Deferred mode*: the view update will only be carried out when it is required.

In the *immediate mode*, a view is kept consistent with its base class all the time. An update to the base class will trigger the update operation on the view. Therefore whenever we access the view, we know it is consistent with the base class(es). This will improve the performance of the materialized view access. However, the *immediate mode* will increase the update overhead to base classes as the system needs to not only update the base class objects, but also update all the view classes that are affected by the update.

In the *deferred mode*, the update view will only be carried out when the view is accessed. The disadvantage of this approach is that consistency evaluation must be carried out before the view is accessed, or even worse it may be necessary to re-materialize the view if an inconsistency is found. This will hamper the performance of view access.

Different materialization strategies perform differently under different situations. [BOTZER96] has a detailed discussion of when to use which materialization strategy for functions in the object-oriented data model. For the framework we have set up for our view materialization, we believe the deferred materialization mode is more

appropriate to our application domain as it achieves the balanced of performance for both view classes and base classes.

In our *deferred mode*, the view will interrogate the VMM upon being accessed to check if any of its base classes have been updated. The type of corrective action taken will depend on the type of update operation on the base class, as discussed above. After a view update, the corresponding message will be removed from the VMM to avoid redundant update operations on the view. Figure 5.1 shows the structure of the VMM.

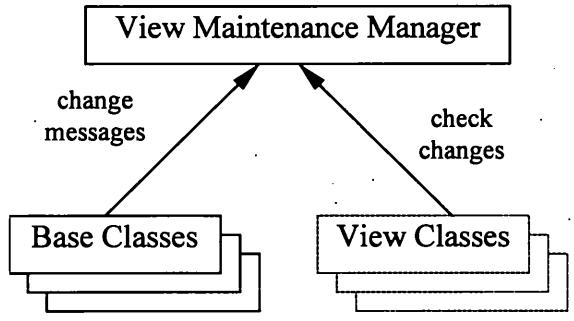


Figure 5. 1 Structure of VMM

Because a base class may be involved in more than one view definition, we keep one copy of the change notification in the VMM for each view class to avoid possible inconsistency between view classes and base classes.

The view manager provides an incremental maintenance of the materialized view in an engineering database. It enables the view maintenance be carried out at the object level instead of class level. Although we limited our view maintenance only to

insert/delete operations, our materialization strategy can easily be extended to cover the situation where an attribute update is required.

5.4 Summary

The view mechanism offers greater flexibility in organising schema and managing data in database systems. However, view classes are computed upon access. This incurs performance overhead on views. The materialized view is seen as an optimization method which can improve the performance of views. The great challenge in view materialization is to maintain the materialized views consistent with its base when the base is updated.

We have presented a view materialization strategy that is applied to a design database where objects in the database may be versioned. The application of versioning implies that objects in the database may endure frequent changes. For view objects frequent updates to base classes will greatly decrease the benefit of view materialization. For optimal performance of our materialized views, we only allow view objects based on working versions and released versions to be materialized.

To facilitate the incremental maintenance of materialized views, we introduced a view maintenance manager to mediate between view classes and base classes. The view maintenance manager approach enables us to transfer the task of maintaining consistency from base class to the VMM. Our argument is that the base class designer

does not know which view will be using the class and it should not the base class designer's responsibility to maintain consistency.

Unlike the MultiView approach, our approach is applicable to other general object-oriented data models. Because of the framework of our view model: we limited the update operation on the base class(es) to insert and delete, we discussed the impact of modification on base class attributes and we believe the function of the VMM can easily be extended to cover such update operations.

The VMM allows view maintenance to be carried out at the object level instead of the class level. This avoids extensive re-evaluation and re-materialization which can improve the performance of the materialized view substantially.

Chapter 6 Views in Integrated Design Environment

Object-Oriented views provides a powerful re-structuring tools for design environments. In previous chapters, we presented a view model that is developed for design environments. In this chapter, we will discuss how to use our view model to provide a flexible design environment and we argue that object-oriented views provide a powerful technique for configuration management.

6.1 Introduction

In a product development environment (e.g. software development) , engineers normally work in groups. These engineers cooperate with each other in order to achieve the products design goal. While at the same time, they need to work on their own un-interrupted by other team members. Normally databases are used support design activities at different levels. When engineers are working on a product, not all the information of each individual's work is relevant to other people on the team. One team's design data may not be relevant to another team. For these reasons, there is a need to divide a design database into different partitions. Now the research community come to consensus that three levels of workspaces provide sufficient support to design activities [KIM95].

Object versioning provides the ability to keep track of an object's evolution path. A complex object is composed of simpler component objects. For complex objects, a configuration management tool is need to help designers to choose correct component objects. The role of configuration management is more complex in a version capable environment. *A configuration is created by composing the system from its components and selecting individual component versions such that the resulting systems is consistent* [ZELER95]. Although the user may get an object without specifying a version number, e.g. through default version, it is desirable that the user is able to select a particular version of an object to configure a complex design object.

Object-oriented views can be used in configuration management to identify the appropriate component object through query predicate. Query languages are generally more expressive than other means of selection used by other configuration management tools. In chapter 4, we have presented an object-oriented view model that can be used in configuration of a complex object. We will show in this chapter that our technique offers a flexible mechanism towards configuration management.

6.2 Object Versions and Workspaces

Versions are distinct snapshots of a design object in different states [AHMED91b]. Version management involves the definition of versioned objects, version identification and organization, and operations for creating new versions and retrieving existing versions. Object versions are organised in version space. A version represents a state of an object during its evolving process. Each version

within an versioned object, must have a unique version identifier. There are many ways of naming a version, we adopt the one that use consecutive integers as our version identifier. Detailed semantics of versioning has been discussed in Chapter 3.

6.2.1 Partition of Design Database

Our database system is partitioned into three workspaces, i.e. private workspace, project workspace and public workspace. The private workspaces are managed by individual designers and project workspaces are associated with each projects. The public workspace is where all released versions are located and can be accessed by people from different projects.

In [CHOU86] the private workspace, project workspace and public workspace each maintains their separate versioning system. The version numbers of a design artefact are independent of each other in different workspaces although they are versions of the same object in different workspace. The separate versioning scheme in different workspaces introduces added complexity into version management and may introduce inconsistency between versions in different workspaces.

In the separate versioning scheme, the user has to assign an appropriate parent version to an object version when it is checked out one workspace and checked into a new one. There is not any mechanism in the database that ensures appropriate parent version is assigned to the object version. This provides a chance of introducing inconsistency into the object's evolution history.

As most engineering artefacts are complex objects, these complex objects have references to other lower level component objects. There are two ways that a complex object can be bound to its versioned components: *static* and *dynamic*. *Static binding* means that the reference to a component object is bound before any object is accessed, e.g. the full path name is included. *Dynamic binding* means component binding is only performed when the component object is actually accessed.

Because of the separation of workspaces, the binding of an object version requires not only its version number but also the name of the workspace it is located in. In [CHOU86] a triplet of <object name, workspace name, version number> is used to name a version. The separate versioning scheme used in [CHOU86] means that when an object migrates between workspaces any static references it has to other component versions have to be converted to new static references that is meaningful in the new workspace.

6.3 Unified Version Management

We have developed a unified version management mechanism that allows efficient and consistent version management throughout the workspaces in a design environment. Consistent version numbers are used for versions of a versioned object throughout different workspaces, i.e., an object version maintains its version number no matter in which workspace it resides. We also unified the version graph in different workspaces which means integrity of an object's version history is maintained.

The Version Maintenance Manager(VMM) that allocates version numbers and maintains version graph is located in public workspace. This arrangement facilitates the share of designs between different projects and it also allows supervisors to examine the entire version set without being confined to a particular workspace.

Each workspace has virtual version graphs which are database views on the main version graphs in public workspace. The private workspace version graph refers to all versions developed by the workspace owner. The project workspace version graph has all the working versions designed by the project team and the public workspace version graph includes all released versions which may be released by different project team. Figure 6.1 to Figure 6.4 illustrate the structure of version graphs in different workspaces.

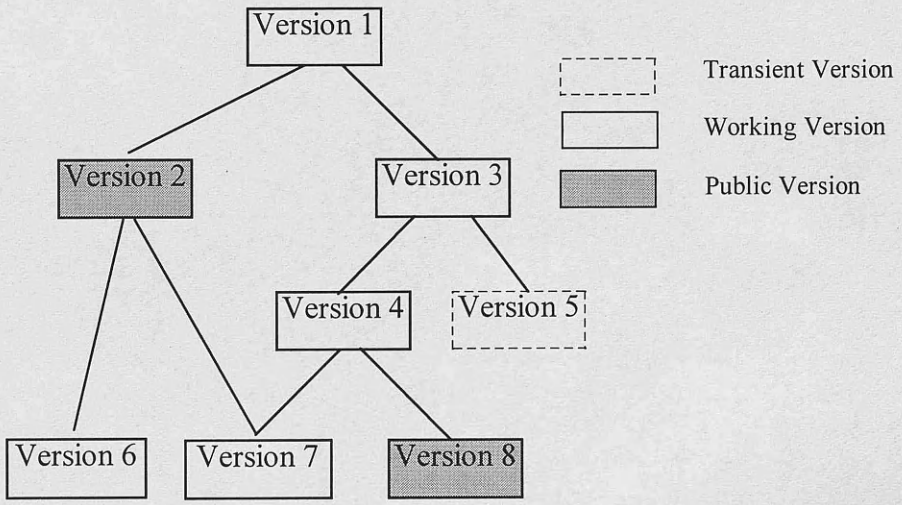


Figure 6. 1 Main Version Graph in Public Workspace

Figure 6.1 shows a main version graph for a versioned object in public workspace. It contains the complete version derivation history for the object. Because all new

versions are generated from private workspace, the private version graph contains all the versions created in this workspace irrespective of their current states in the environment.

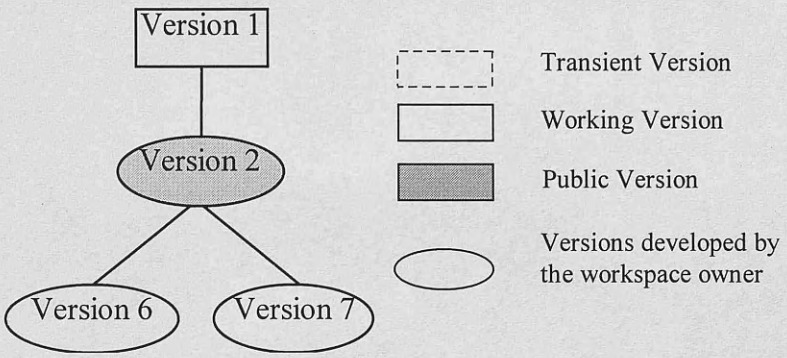


Figure 6. 2 Version Graph in Private Workspace

Figure 6.2 shows a version graph in a private workspace. This is a virtual version graph. It is defined by a database view on the main version graph for the same object with all the versions developed by the workspace owner. These versions may be in transient states or working state. The ovals in the diagram represent versions developed by the workspace owner.

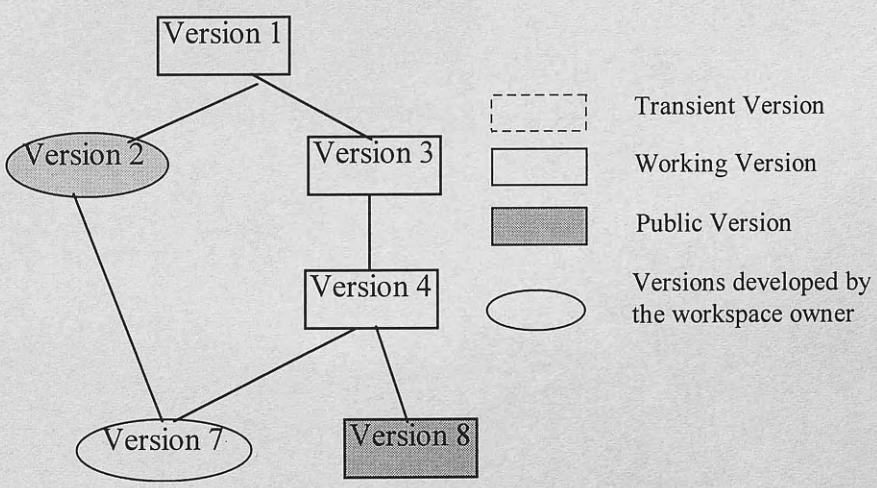


Figure 6. 3 Version Graph in Project Workspace

The version graph in project workspace (shown in Figure 6.3) is also a database view. This view is based on the object's main version graph with all the versions in working states. As shown in 1c, version 3, 4, 8 are not in the private workspace shown in figure 6.1. These versions are developed by other designer working on the project.

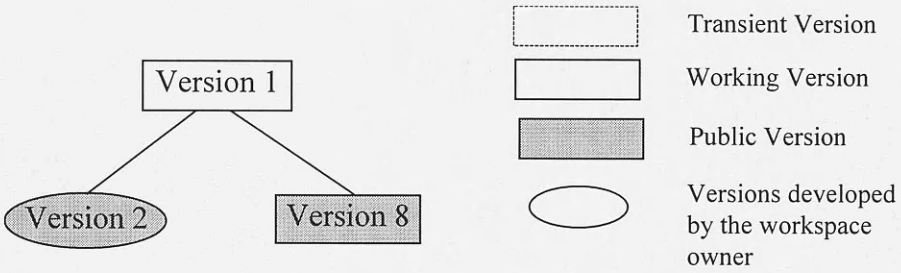


Figure 6. 4 Version Graph in Public Workspace

Figure 6.4 illustrates the version graph for public workspace. It has only released versions. A released version may be created by another development team working on a different project.

In our version model version 1 is the root version from which all versions of an object are derived. If version 1 is not included in figure 6.3 then to the users of the public workspace that version 2 and version 8 are not unrelated. In such a circumstance some semantic information is lost during the conversion process. To avoid the loss of semantic information in a unified version graph, version 1 is always included in all version graphs. If version 1 is not visible in a particular workspace, it will not be accessible from the version graph in the workspace. If the users have to create a new version graph in the public workspace as described in [CHOU86], then they would

face a very difficult decision on how to relate version 2 and version 8 in their version graph.

As can be seen from figure 6.1, version numbers in different workspaces are not necessarily consecutive. In a version model, version identifier represents the partial time order of creation of each version. These version identifiers in our environment still reflect the partial orders of these versions, i.e., no semantic information is lost during the unifying process. Using unified version identifiers means that only one version manager is needed for each object in the develop environment. This will make our version management less complicated than that of Chou's. More importantly all the versions maintain the intrinsic relationships with their parent versions. We say an object version x is a parent version of version y if there is a path from x to y in the version graph of the object.

6.3.1 Version Names

In our version model, integer numbers are used as the version identifier. Versions are assigned consecutive integers in the order of their creation. As there are various levels of workspaces in our development environment, an object's name and its version identifier may not provide enough information to locate the object.

We use a name tuple <workspace, version number> to identify an object version in our environment. The workspace indicate which workspace the version is located. The version number is the version identifier of the object version.

6.3.2 Object Version Migration

As unified version identifiers are used throughout our database environment, when an object version migrates from one workspace to another, the only information that needs to be updated to keep object references meaningful is the workspace name.

The workspace name and its category can be updated automatically when the object is checked into a new workspace. Because an object in the lower category workspace may reference objects in the higher category workspaces, e.g., an object version in a private workspace has reference to a component version in a project workspace, it is not always necessary to update the reference to a component.

For example, a version of Car object in private workspace has references to a version of Engine object and a version of Bodywork object (Figure6.5). The Engine object version is a working version and is located in a project workspace. In this example, the designer who is working on the Car object happens to be working on the Bodywork as well. The version of Bodywork object, therefore, is also in his/her private workspace (figure 6.5a).

When the designer decides to release the version of Car object into the project workspace, the referenced Bodywork object version needs to be released as well. When the Car object is checked into the project workspace, the Bodywork object version is also released into project workspace and the reference from the Car object

version to the Bodywork object version has to be updated at the same time. Since the Engine object version is already in the project workspace, this reference to the Engine object version will remain unchanged (figure 6.5b).

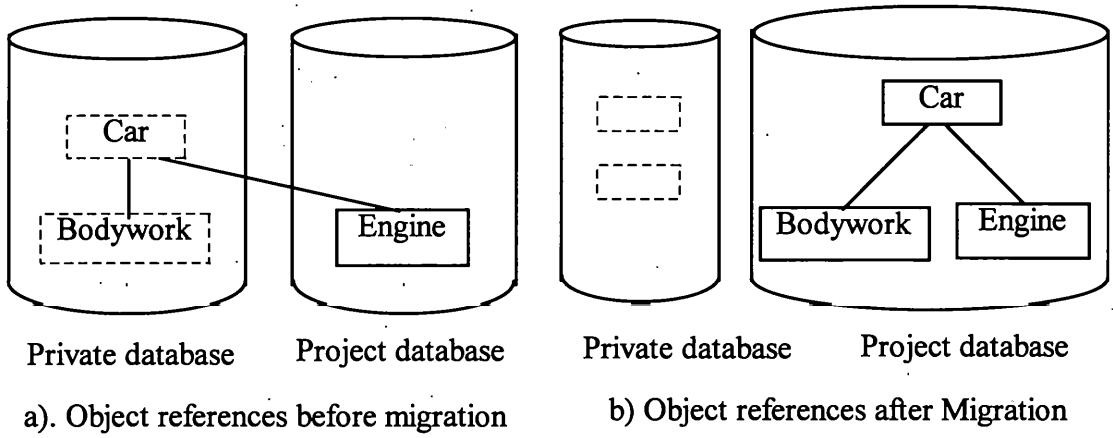


Figure 6. 5 Object References during Object Migration

Unified version management in a multi-level workspace environment provides consistent version identifiers throughout the whole environment without further complicating the management of version identifiers. Consistent version evolution history is maintained in all the participating workspaces. The use of database views for version graph in project workspace and public workspace provide up to date information about the changes in these workspaces and no extra work is needed to maintain separate version graphs in project and public workspace.

6.4 Configuration Management

A complex object comprising a set of components is configured by selecting a version for each of the component objects that constitute the complex object

[AGRAWAL89]. If any of the component has been modified, the database needs to react to the changes that have impact on the complex object. Version management defines the object to be versioned, version identification and organisation, as well as operations for retrieving existing versions and constructing new versions [CONRADI96] while configuration management is the art of selecting and controlling modifications to a complex object.

Configuration management allows a user to specify alternative configuration of a complex object through the selection of appropriate versions of its components. A configuration can be specified and constructed by describing a set of desired attributes. The attribute can be a version number that is associated with each of the versioned object.

6.4.1 Views in Configuration Management

Tradition configuration management uses version label to select components for a complex object (as discussed in Chapter 3). This approach has limited expressive power as in many circumstances the user might want to use selection criteria other than version numbers. We see that a database view as an ideal tool for selecting component objects in a configuration management as it uses query language, e.g., OQL, to define the selection criteria. Query languages are more expressive than other means of selection.

We have developed a view model that supports configuration management in a design environment. It provides a flexible approach towards complex object configuration. The use of database views not only facilitates the selection of components but also allows us to adapt the complex object towards new user requirements.

For example, in a design object Car has components Bodywork and Engine as shown in figure 6.5. The designer wants to try a version of Engine from another source. The Engine object may be belong to a different abstract data type than the in-house one. Normally a new Car object would need to be created to cater the change in design. With capacity for an augmented view in our view model, the designer only needs to define the view object other than creating a new Car object. With the view approach the user can use any other selection predicate to choose object configuration that is supported by the query language.

The query language can be used to define a selection criteria which describes a set of desired features of components versions. The selection may return an empty set, indicating that no component that meets the configuration requirement. The selection may also return more than one versions of a component that meet the configuration requirement. When such a case arises, the user need to develop a mechanism that allows the user to specify more specific selections to make it unambiguous.

The system has several options when faced with multiple choice of components:

1) Choose one at random;

2) Create the cross product of all possible configurations;

3) Provide the user with the appropriate operational mechanisms to describe the desired configuration.

Choice (1) provides a simple but limited solution to get a single version for a configuration. Choice (2) will result in exponential explosion of configurations when faced with multiple choice of component in more than one components. [SCIORE94] allows multiple component for a configuration. Choice (3) allows the user to refine the query and select a single version from the result list.

6.4.2 Identifying Object Components

Configuration constraints are rules to check against when a component is included into a configuration. If an object fails its configuration constraints then it means that the object does not meet the design requirement and its inclusion in the configuration will be rejected.

Configuration constraints are conditions specified by designers to ensure the consistency of a design object configuration. Our version model allows designers to specify configuration constraints for lower-level objects. In our model, it is the responsibility of higher-level object to ensure that its components meet their configuration constraints.

When a configuration query returns more than one versions of a particular object, the configuration view will check these objects against configuration constraints. If any of them fail the test, they will be removed from the result list thus reduce the number

of available object versions for a configuration. If there are still more than one versions for any component, then the user will have the choice to view through these objects and manually pick up one for the configuration or revise the query condition to provide stricter criteria.

If the user chooses to manually select the available component, then the configuration criteria will be modified to include the selected object's version number. Thus guarantee the configuration with one version for each component object.

For a configuration specification, when multiple versions for a component are returned by a query, if the user does not have any preference among these versions, he/she could simply choose the most recent version. This is supported by our view model and the specification for the configuration will be automatically updated. Should the designer choose to revise the configuration specification, i.e., the query condition, we have to go through the above procedure until a single version is selected.

The view approach is similar to that of dynamic binding. Selected object versions might be different each time the configuration is accessed. This implies that a complex object configuration can automatically take advantage of new versions of its components. At some stage, however, the user may need to freeze a configuration. The frozen configuration will become a new version of a complex object. All dynamic bindings in the configuration will be converted to static ones, i.e., all the

selection criteria will change to version numbers that uniquely identify the components, e.g., configuration numbers.

From the above discussion, we can see that the view approach towards configuration management not only facilitates the users in experimenting with all possible configurations but also allows the user to take advantage of latest development. On the other hand it still allows the user to freeze a particular configuration whenever needed.

This part of the work has not been implemented due to the limitations of object-database being used in this project. The reason POET was chosen as development tool at the begining of the project was because it was the only object database available for PC platform and it supports ODMG standard.

6.5 Summary

In this chapter, we presented the unified version management in design environment. The unified version management maintains consistent version numbers throughout the develop environment. Although version numbers in each workspace may not consecutive, no semantic information is lost during the process.

For each versioned object, only one version manager is needed in the environment. This reduces the complexity of version management in the environment. The relationships between versions are maintained even when an object version migrates

from one workspace to another. We believe our approach provide consistent version management in a develop environment. The user no longer needs to assign a parent to a new version when it migrates into a new workspace which carries the risk of introducing inconsistency.

We argue that database views are a better tool for configuration management. The use of query language in the selection of components have more expressive power than the use of version number list and can assure the consistency of the configuration. Database views provides a more flexible approach towards configuration management.

Chapter 7 Prototype Design and Implementation

This chapter presents the design and implementation of a prototype for unified version management in a design environment. The prototype includes the version model presented in chapter 3, the view model presented in chapter 4 and the architecture of view maintainence manager discussed in chapter 6. The prototype presented in this chapter is simplified as it is used to demonstrate the feasibility of our design.

7.1 Introduction

The primary objectives of the prototype are:

- Demonstrate the feasibility of our VMM model. There are novel contributions of the model and it is important to demonstrate that they are actually applicable in practice.

- A prototype can give some indications of the usefulness of theory used in the research. The prototype provides a testbed to experiment with various scenarios and give some indications of their value.

- A prototype may help to detect possible design flaws in the model. This will provide valuable feedback to the development of the model. The implementation

may also highlight possible areas of future work and it provides an environment for experimenting with different design possibilities.

The prototype has several components:

- **Version Model** provides the versioning capability to objects that are derived from versionable classes. It provides functionality for keeping the semantic relationships between versions of an object. In a design database, a design artefact is generally multi-versioned.

- **View Model** is composed of two parts. A view query definition class provides storage for query conditions that define the extent of a view class and it keeps the information about which schema to use to initialise view objects. The view schema definition class, as it name implies, defines the schema for view objects.

- **View Maintenance Manager(VMM)** provides a mechanism for efficient maintenance of materialized views. The VMM acts as a mediator between view classes and their base classes to keep them synchronised upon view access.

7.2 The Version Model

The version model consists of three classes: `Generic`, `Descriptor` and `Versionable` classes. Figure 7.1 shows graphically the relationship between these three classes. The `Generic` class provides the version management functionality for

the version model. There is only one `Generic` object (List 7.1) for each set of versions. It will allocate the next available version number to new versions and put the new version onto an appropriate place on the version history graph according to its relationships with its base version(s).

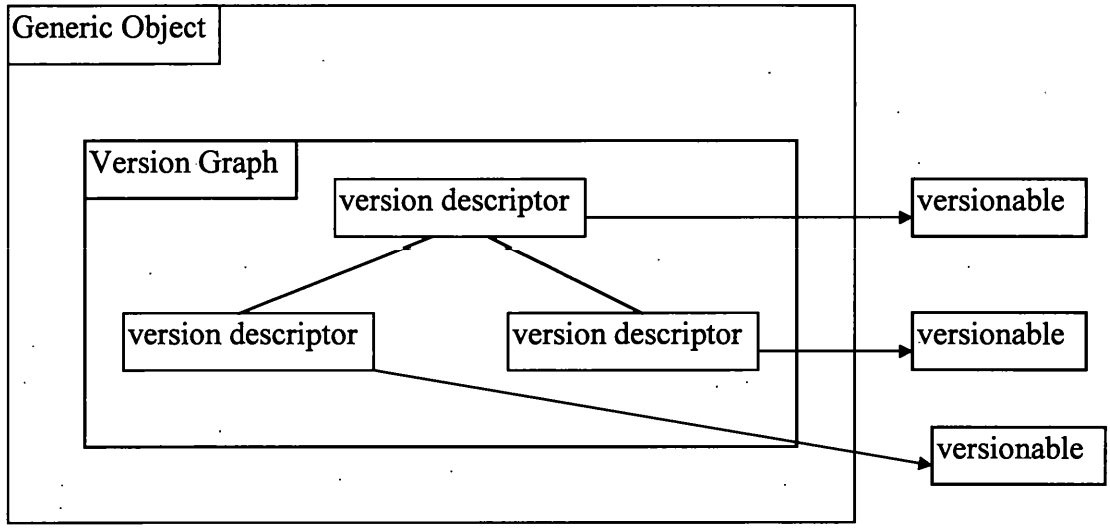


Figure 7.1 Class Relationships for Version Model

The version graph is implemented as a Direct Acyclic Graph due to the nature of version semantics. For each version set, there is a default version. The default version number is stored in the `Generic` object for easy access. The `Generic` object provides methods for creating new versions (e.g. `merge_versions()`).

```
persistent class Generic
{
   private:
     int   last_version_no;
     int   default_version_no;
     DAG   version_history;
     ......
   public:
     Generic();
     ~Generic();
     .......
};
```

List 7. 1 Class for Generic Objects

The `Descriptor` class (List 7.2)acts as a flag for instances of a `Versionable` class. Each instance of a Versionable class has a corresponding `Descriptor` object. The main function of a `Descriptor` object is to hold a flag to indicate whether the corresponding `Versionable` object has been deleted. If a Versionable object has other versions derived from it, it cannot be physically deleted. The flag in the `Descriptor` object will stop new versions being derived from it. It is the `Descriptor` object that is actually stored on the version graph of a `Generic` object. The reason that a separate descriptor object is kept on the version graph instead of the version object itself is because the descriptor object is a lot smaller than a vesionable object.

```
persistent class Descriptor
{
   private:
     int             version_no;
     bool            deleted;  //deletion flag
     Versionable*    ver;
     ......
   public:
     Descriptor();
     ~Descriptor();
     ......
};
```

List 7. 2 Class for Descriptor Objects

The `Versionable` object (List 7.3) is at the core of the version model. All classes that require versioning capability are derived from `Versionable`. Each `Versionable` object has a reference to its generic object.

```
persistent class Versionable
{
   private:
     STATE      state;  //version state transient,...
     int        version_no;
     Generic*   pGen;
     ......
   public:
     Versionable();
     ~Versionable();
     ......
};
```

List 7. 3 Class Versionable Objects

The design of the version model allows the user to impose versioning capability by deriving their classes from the Versionable class. The Generic class provides the basic version management scheme. This version model can be easily extended should the user require more than what is offered by the base model.

7.3 The View Model

For relational databases, a view is a stored query. A view set is returned by running the query against base table when the view is accessed. In object-oriented databases, a view is composed of two parts. The first part is inherited from the previous generation database, as a stored query. The second part is unique to the object data model - the schema definition for the view class. This split of function is due to the need to define methods for view objects in view schemas. The query languages such as OQL, are not computationally complete. It is very difficult to define methods using just a query language. In our view model, the two parts are implemented as two separate classes. This approach gives the user greater flexibility in defining a view. The user can either define a new view schema or use an existing schema for their database views, which in turn means a view can either be object-generating or object preserving.

The query part of the view only defines the extent of a view. Because the object-oriented database stores everything as objects, the stored query part of a view is stored in the form of an object. It is also possible to tell the database which schema to use to populate the view class.

This design of a view model not only provides full support to existing view semantics but also enables the user to extend view semantics from those in the previous generation database. Chapter 4 has a detailed discussion of view semantics. In the

following section, we use several examples to demonstrate how various view semantics are implemented in our view model.

7.3.1. Example of View Definition

To illustrate the structure of our view model, a few examples are developed to demonstrate how our view model works. We use the POET™ object database to implement our view model. The POET database is ODMG compliant therefore we are able to use OQL to define our view query. C++ is the data definition language in POET. Because only a subset of the ODMG OQL is supported by POET, the view model is implemented using query functions provided by POET. However, the semantics of these queries is fully supported by ODMG OQL and we use OQL in our examples to specify our view query definition.

Example 1: Selection view using an Object-Preserving View

Firstly, we illustrate how to create an *object-preserving-view* in our view model. Generally an *object-preserving view* is used in SELECT view where the view objects are a subset of the base objects. This is because SELECT view does not need a new schema for its view objects since the view object is of the same type as its base objects.

```
//view query definition
class HatchBacks
{
  private:
     view_class Car
     string query_spec =``select car from cars
                         in all car
                         where BodyWork=\''Hatch\''''

  public:
     ..........
     Activate();
}
```

List 7. 4 Object-Preserving View in SELECT View

In our example, the designer wants to look at all the cars that are hatchbacks. Because in an *object-preserving view*, the schema of the view object is the same as that of base objects, the schema definition part in this example is not needed. The user only needs to specify that the view class is a set of cars and then specify the query condition . List 7.4 shows the view definition.

In an *object-preserving view*, there is no need for the user to create a view schema definition as it is the same as its base class schema. The view objects share the same oids as their base objects and the view class contains a subset of objects from their base class. Thus our approach provides a simple solution to a *selection view*.

Example 2: Projection view using an Object-Creating View

In our second example, we illustrate how to use an *object-generating view* in a *projection view*. Suppose, in our database there is a `Car` class which has four attributes: `make`, `model`, `engine`, and `carBody` as shown in figure 7.2.

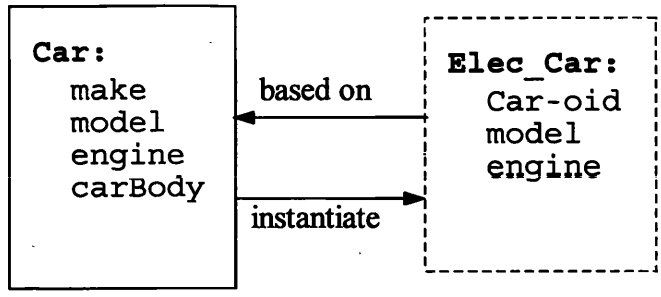


Figure 7. 2 Car Object and Its View Elec_Car

The view `Elec_Car` is a *projection view* based on `Car`. The view class contains all cars that are powered by electric engines. The user of the view is not interested in the bodywork of the car so the attribute carBody is projected out of the view. The definition of the view is shown in List 7.5:

In the view query definition class, the method `Activate()` is called automatically to instantiate view objects each time the view query class is accessed. All methods of views are defined in schema classes. As all update operations are carried out through view methods instead of query statements we can eliminate the update anomaly associated with the traditional view approach.

```
//view schema definition
class Elec_Car
{
 private:
      oid  car_oid;
      string model;
      Engine elec_engine;

 public:
      //constructors and methods defined for the view
      ......
}


//view query definition
class elec_car_view
{
 private:
      view_class Elec_Car;

      string query_spec=``select car
                        from cars in allCar
                        where car.engine.type =
                        \''Electric\''';

      Elec_Car_Set view_objects;
 public:

      Activate();
      ......
}
```

List 7. 5 Object-Generating view in Projection View

Example 3: Join View

In our third example, we present a *join view* which combines two base objects to create a new one. For a join view, an *object-generating view is used* as semantically a *join view creates* a new type for its view class. Figure 7.3 shows the two base classes, `Engine` and `GearBox`. In figure 7.3 the bold attributes in both base classes are used to join two base objects.

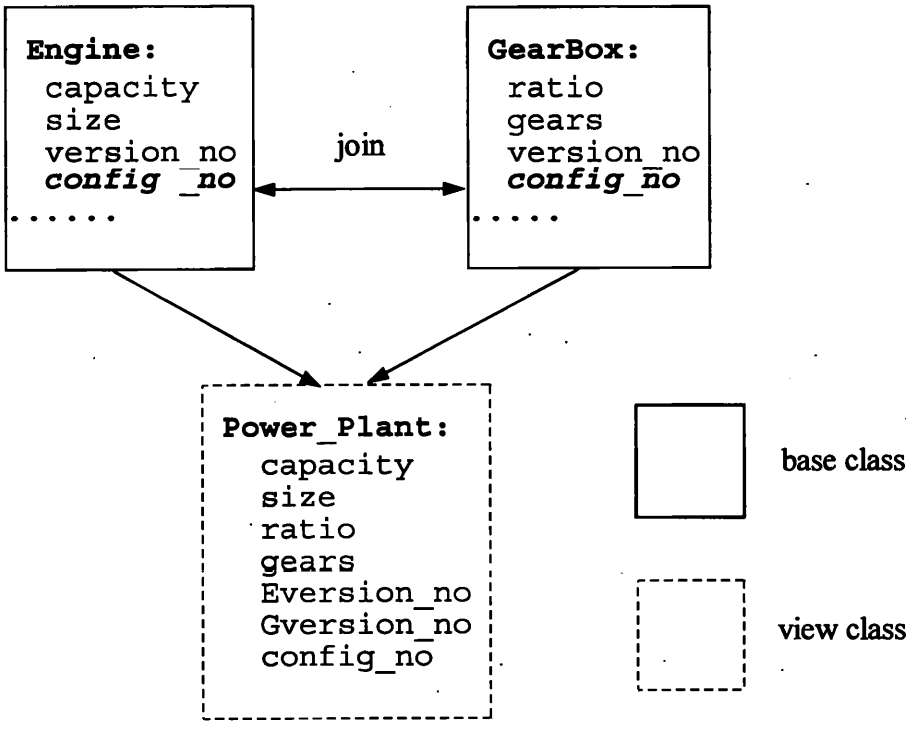


Figure 7. 3 The Join of Engine Object and GearBox Object

In the example, the result of our join created a new view class, `Power_Plant` which contains the attributes and methods from both base classes. While specifying the schema for a view class the user can project out any attributes or methods if he/she wishes to. In this example, there is a name conflict for `version_no` in both classes. The view designer is responsible for resolving the name conflict in view classes. The `config_no` seems like a conflicting attribute name as well, but it is used as the joinable attribute which is treated as a single attribute in the view class. The definition of the view query is shown in List 7.6

```
//view query definition for Power_Plant
class Power_Plant_view
{
   private:
     view class Power_Plant;
     string query_spec = ``select engine, gearbox
                         from engines in allEngine
                         gearboxes in allGearBox
                         where engine.config_no =
                         gearbox.config_no;

     Power_Plant_Set  view_objects;
  public:
     Activate();
     ........
}
```

List 7. 6 Query Definition for Join View Power_Plant

The similarity between a relational join view and an object-oriented join view can be seen from the query definition. In an object-oriented view the user has to define the view schema before any view object can be used, although semantically the result of the join view is the same as that of deriving a subclass from both base classes. For reasons discussed earlier in this chapter, the view classes are part of the inheritance hierarchy of its base classes.

Example 4: Extended View

From the last 3 examples, the reader can find corresponding views in relational form. This next example (figure 7.4), however, is unique to the object-oriented paradigm. An extended view contains attributes that are not part of its base class objects and cannot be derived from its base class object attributes. The extended part of the view is created as a new object and stored in the database to enable the view to be initialised each time it is called.

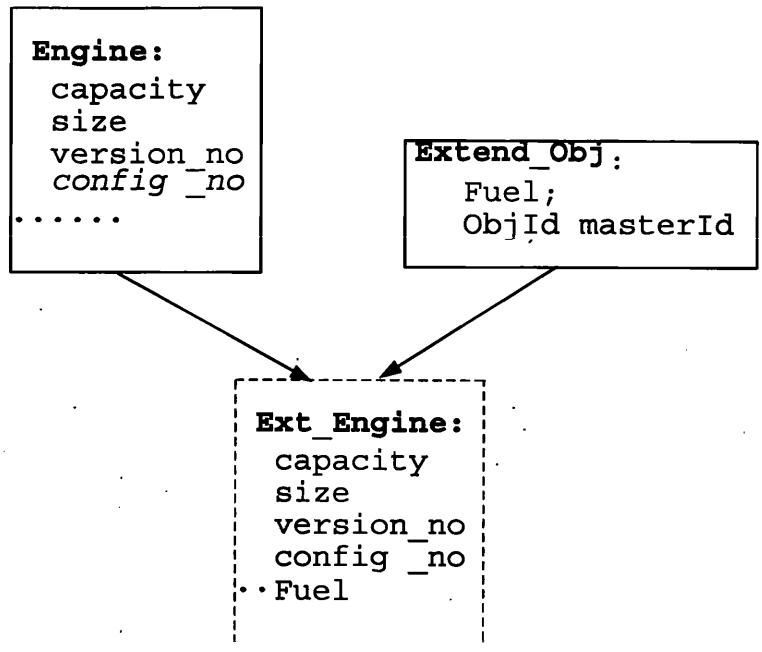


Figure 7. 4 Example of Extended View

The extended part of the object contains extra attributes, plus the object Id of its associated base object in order to guarantee that the extended view is initialised correctly each time it is accessed. Since there is no existing schema in the database which can be used as the view schema, the new view schema has to be specified as in previous object-generating examples. The view query specification is basically a join query, to join the base object oid with the extended object `masterId` as shown in List 7.7.

```
//view query definition for extended view
class Ext_Engine_view
{
  private:
     view_class Ext_Engine;
     string query_spec=``select engine extend_obj
                        from engines in allEngine
                        extend_objs in allExtendObj
                        where engine.oid
                        = extend_obj.masterId
  public:
     Activate();
     ............
}
```

List 7. 7 Example of Extended View

Example 5. Union View

Union view is quite straight forward in the Object-Oriented paradigm. From the semantics of union, the result of a union includes instances of both the classes involved. The semantic constraint on union means that the resulting class must be the superclass of both classes taking part in the union. We use a typical college database to illustrate how a union query is formulated in our database (Figure 7.5).

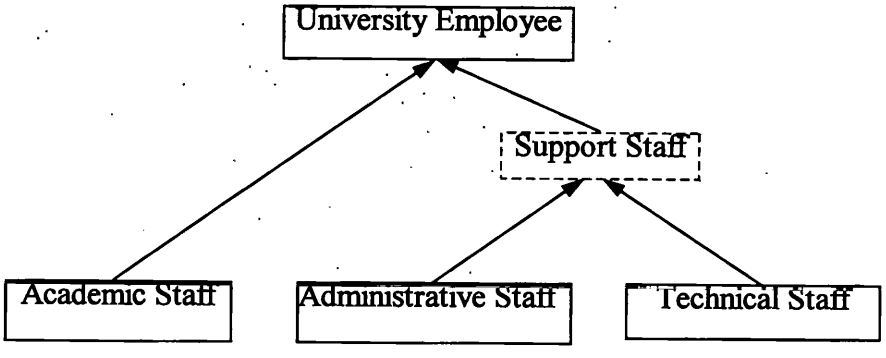


Figure 7.5 Class Hierarchy for Union Example

For a query to concerning support staff, we need the union of administrative staff and technical staff. The view query would involve putting instances from both classes into the view class as shown in List 7.8.

```
//view query definition for union view
class Support_Staff_View
{
     private:
          view_class support_staff
     public:
          Support_Staff_View();
          ~Support_Staff_View();
          Activate()
};

//implementation of Activate method
Support_Staff_View::Activate()
{
  //Sets contains all instances of each classes
  Administraive_StaffAllSet admin;
  Technical_StaffAllSet tech;
  University_Employee* pEmployee

  for(int I=0; admin.Get(pEmployee, i, PtStart);
     I++)
  {
     support_staff.Append(pEmployee);
  }
     .......
}
```

List 7. 8 Example of Union View

In the above example, the view class `Support_Staff` could be `University_Employee` or the user may wish to define a new view class which would be the super class of `Administrative Staff` and `Technical Staff`. Because POET only supports a subset of OQL [CATTELL97], this example looks more like a C++ program than a demonstration of view semantics. This is due to the limited support of set operations in the version of POET we used to implement our

view model. If future versions of the POET were to support binary set operations, then the above union would look a lot simpler as shown in List 7.9.

```
Class Support_Staff_View
{
     private:
          view_class support_staff
          Administraive_StaffAllSet admin;
          Technical_StaffAllSet tech;

          string query_spec = ``support_Staff union
                              tech'';
     public:
          Support_Staff_View();
          ~Support_Staff_View();
          Activate()

     ....

};
```

List 7. 9 Union View in OQL

For *Intersection View* and *Difference View,* we had to implement a very crude C++ solution to achieve the semantics of these views. The operation basically involves getting the OID of the first class and then finding out if it is in the second class. In the ODMG OQL the intersection operator is `intersect` and the difference operator is `except`.

7.4 View Maintenance Manager

The main objective of the View Maintenance Manager (VMM) is to provide an efficient view maintenance mechanism for materialized views. Ideally the VMM should impose no impact on those base classes that have no views based on them. We

have added an extra attribute which acts as a flag indicating whether a view is derived from it. A simple check on this flag will indicate whether to update the VMM. This will minimize performance overheads on those base classes with no view derived from them.

The View Maintenance Manager acts as a mediator between views and their base classes. For each base class, there will be a list that contains all the view classes that are based on it (List 7.10). For each list there is a time stamp associated with it indicating the time of base updates.

```
class base_view_list
{
     private:
          Date timestamp;
          Cstring base_class_id
          List ViewClassIds
          .....
     public:
          base_view_list();
          ~base_view_list();
          Add(view_class_id);
          Remove(view_class_id);
     .....
};
```

List 7. 10 View Maintenance Class

Each time a view is accessed, the VMM checks its time stamp against the time stamp of the base_view_list. If the time stamp in the base_view_list is later than that of the view then a re-materialization will occur.

As discussed in chapter 5, this is a very primitive view maintenance approach. A more elaborate approach is envisaged and discussed in chapter 5 as future work. Due to time constraints, only the primitive time stamp approach has been implemented

Using the VMM, we can implement different view maintenance approaches, i.e. immediate or deferred. This is achieved by adding a mode attribute indicating whether the user wants the immediate view update or deferred. If it is set to the immediate mode, the VMM can be set to call a method to re-materialize the views. If it is set to the deferred mode, the view class will only check its consistency upon view access.

7.4.1 Implementation of the View Maintenance Manager (VMM)

The VMM has been implemented using Borland C++ and the POET object database. As an earlier version of POET has very limited functionality in supporting object queries and other aspects of object storage, we were forced to use demonstration versions of POET because this allowed us to take advantage of the latest functionality offered in new versions of POET database.

The VMM keeps the view maintenance information in the database. When a base class object is being updated, the new update message will simply replace the old one if a view class has not been accessed since last base update. When a view class is accessed, the VMM will re-materialize it to ensure consistency with its base and remove the message entry for this view class. At the moment, we have only

implemented a simple message scheme to indicate an update in the base. As regards future work, the message should contain more information about the base update, e.g. a list of attributes that have been updated. The extra information would enable the VMM to reduce unnecessary re-materialization of the view class and thus further improve the performance of our view model.

7.5 Summary

This chapter has presented the design and implementation of the main components in our design environment. We have demonstrated that the whole model can be implemented using an appropriate programming language. However the effectiveness of our model has not been properly evaluated. Only extensive experimentation with a large, evolving application could contribute in this aspect as discussed in the next chapter.

Chapter 8 Evaluation

8.1 Introduction

This chapter evaluates the version and view models in a design environment. The evaluation considers the achievements of the system. Consider how our models might be used in a design environment to support the process of engineering development. The ideal would be to allow the models to be used in a project of realistic size and complexity, and to investigate their performance in terms of ease of use, physical performance statistics, and so on. However, for a project of this scale, it was difficult to set up a realistic test environment due to limited resources and time. In the following sections of this chapter, we compare our work with existing work to demonstrate that our approach provides better support for design activities.

8.2 Object View Model Development

The view mechanism provides the presentation model for our system. The successful development of a comprehensive object view model provides a solid foundation for a flexible system. Our view model is defined by two separate classes. The first class defines the schema of a view class and the second class defines the extent of a view class. This strategy allows the user to either use an existing schema as the view schema or to define a new view schema. Unlike many other view models which are either object-generating or object-preserving [ABITEBOUL91, BERTINO92,

HEILER90, SCHOLL91] this view model allows the view object to be created either with a new object id or using its base objects' id. This means that our view model is not limited to one view semantics rather it can take advantage of both semantics. An *object preserving view* simplifies the view definition and allows direct view updates. On the other hand, an *object generating view* allows an extended view schema to be defined which gives greater flexibility to the view model. Therefore, our view model can provide all the view semantics set out in our view taxonomy.

As in the relational world, defining object-oriented views involves a query language. Because the object-oriented data model offers richer semantics, it is natural to expect object-oriented views to offer richer semantics than relational views. On the other hand, the object query language standard (OQL) does not offer any more than a relational query language (SQL). To circumvent this limitations of OQL, two different approaches were adopted to extend the semantics of object-oriented views.

[BERT91, HEIL91] both only use a query language to define their view model. To overcome the limited semantics of the existing query language, Bertino extended the semantics of the query language to provide support for schema changes. [HEIL91] simply provided a view model that could do no more than relational views.

The other camp [ABIT91, SCHO91] combine the query language and object data model to define their view models. As their object data model offers more than the relational model, object views using the object data model can be easily extended. The disadvantage of this approach is that it has loses the simplicity of using a query

language to define the schema of a view, and data models must be used as a part of view schema definition.

[ABIT91] defines his view model with virtual classes. Instances of the virtual classes are assigned their own Oids. The virtual classes are integrated into the inheritance hierarchy of the base classes. Abiteboul incorporates the view into a coherent framework. There are a few problems with the integration approach. Firstly, it exposes views to the effects of schema changes in the database. In an inheritance hierarchy of only base classes, any schema change is propagated down the inheritance tree automatically. However, it is generally impossible for a database system to change a view specification in a view definition to reflect schema changes. Secondly, the semantics of the inheritance hierarchy may be violated if a view class is placed without first considering its type definition and extent.

Scholl's [SCHO91] view model allows view updates which is a desirable feature in a view mechanism. The view model is defined using a query language and the type system for the object data model. The view class is virtually populated by the same objects from the base class which means it is an object preserving view. For this type of view, it is difficult to modify the behaviour of a view object.

To overcome the above existing problems associated with object generating and object preserving views, our view model allows the user to specify which one to use at design time. In the meantime it allows view updates to be propagated to their base classes regardless of whether new Oids are generated for view objects.

8.3 Version Management and Design Environment

8.3.1 Version Management

Version management involves the management of version creation and the relationships between the versions. There is no consensus on when a new version of an object should be created. [AHME91, BEEC88, SCIO94] proposed to use version sensitive attributes to control the creation of new versions. A new version is created when any of the version sensitive attributes have been updated. This approach may automate the process of creating new versions but we think it is too restrictive.

Creating a new version is a complex design decision. To say some attributes are more version sensitive than others is an oversimplification. If some version sensitive attributes are defined, then a new version is created whenever a version sensitive attribute has been modified irrespective of the semantics of the change. Defining version sensitive attributes also forces the database designer to decide how a new version can be created at an early stage in development which may later prove to be inappropriate. We argue it is more appropriate to let the application user decide when a new version needs to be created, as it is complicated design decision. In our version model we adopt the approach that the user decides when to create new versions. Sometimes it may be undesirable to automate the version creation process.

[CHOU86] proposed another approach to solve the problem of version proliferation. Whenever a component has a new version created, it informs all the objects that are

directly one level up in the configuration hierarchy. The user of the upper level object then decides if any changes need to be made and informs the object at the higher level. The change notification approach prevents version proliferation in complex objects.

This approach requires that the lower level object has to maintain a table that lists all its direct upper level objects. When a component object is being designed, it may not have been designed to be used by the complex object. This means that the component object needs to be informed about its inclusion in the complex object and needs to update the object's table. This complicates the management of object versioning. We argue that it should be the complex object designer's responsibility to check if any of its components have been modified. Sometimes even though the changes at the lower level object may not have any effect at the upper level, the designer of the upper level object still needs to look into the changes made to the object to decide whether to respond to the change. In a dynamic design environment this may mean many distractions to the designer of the upper level object with a lot of irrelevant notifications.

In our version model, instead of making the lower level object responsible for notifying the upper level object of any change, it is the responsibility of the upper level object designer to specify a set of conditions by which to check its components. If any of the changes in a component fails to meet any of these conditions then the complex object designer is alerted about the change. If the changes in the lower component do not break any of these conditions then those changes will go unnoticed by the upper level object designer. Because the upper level object designer knows

which parameters in the component are more important to the design, he may target these by checking on those important aspects of the component object. Imagine a very complex object with a lot of components, if any change in the direct components leads to a need to inform the upper level object designer, then in Chou's system the upper level designer may spend a lot of time checking all these change notifications, whereas, in our model, he works only on meaningful ones.

8.3.2. Design Environment

In this project versioning is viewed from a design environment perspective, that is. how it is being used in a design environment. We consider such a design environment consists of three levels, i.e. private, project and public workspace. Unlike [CHOU86] where versions are confined to each individual workspace, we provide a globally consistent view of versions in a design environment (called unified version management). In CHOU's approach, when an object checks out of one workspace and checks into another, it is the user's responsibility to decide the relationship between the new version and existing versions already in that workspace. This can lead to inappropriate version semantics being defined in that workspace and there is no mechanism to prevent it from happening.

Unified version management allows consistent version numbers to be used for versions of a versioned object in different the workspaces, i.e. an object version maintains its version number no matter in which database it resides. Unified version management in a multi-level database environment provides consistent version

identifiers throughout the whole environment. Consistent version evolution history is maintained in all the participating databases. The use of database views for version graphs in the project database and public database provides up to date information about the changes in these databases and no extra work is needed to maintain separate version graphs in the project and public database.

8.4 Configuration Management

In this project we explored the expressive power of object-oriented views. Views are used to produce the appropriate version graph in various workspaces. We also explored their use in configuration management. The use of database views in configuration management allows the user to specify a list of configuration criteria in the query language instead of using some arbitrary attribute, e.g. version numbers, to identify a configuration component. In this project, we have developed a view model that supports configuration management. It provides a flexible approach for complex object configuration.

The aim of our view model is to support complex object configuration management. The view approach offers greater flexibility than other configuration management systems [SCIORE94]. Traditionally freezing a configuration is achieved by converting all the generic references to an object into specific references, and consequently is an expensive operation. A view definition can be specified such as to achieve the effect of the frozen configuration when needed. Alternatively, view materialization can be used to freeze a configuration.

8.5 Summary

In this chapter, we have evaluated our project against existing approaches by considering the following aspects: version model, view model and configuration management . By considering each aspect separately while bearing in mind the aim of the project, we have evaluated them against existing work carried out by other researchers. By means of these comparisons, we have now demonstrated that our project provides a novel approach for version management and configuration management in a design environment.

Chapter 9 Conclusions and Future Work

In this the final chapter of the thesis we review the work that has been carried out, demonstrate that the hypothesis has been proved, and discuss the areas in which the research may continue in the future.

9.1 Conclusions

The aim of the research project was to investigate how object-oriented database could provide efficient support for cooperative work in an engineering design environment, with particular emphasis on the use of database views. To achieve the aim of the research, we have investigated several areas including the role of versioning, characteristics of design databases and database view mechanism used as part of design database. Through the investigations, we concluded that object-oriented database system is suitable repository for design environment and database views allow configuration to be specified by a set of desired features. This method expresses a higher semantic level than the alternative method of associating a configuration number with each of the components of a configuration.

The hypothesis of the research, as stated in chapter 1, is that object-oriented database systems provide better support for change management in a design environment than second generation database systems, particularly through the use of database views.

Chapter 2 of this thesis presented an analysis of the needs of design environment through examining version models and configuration management scheme. From this analysis, the importance of flexible configuration management tool in design environment was advanced.

A structured repository for recording all design data is at the heart of a design environment. In chapters 2, we examined the advantages of using database systems in a design environment and investigated how object-oriented database system could satisfy the requirements of design environments. We showed that *OODBs provide better support for design environments than second generation databases*

Design activities are intrinsically iterative in nature. In chapter 3 we established the versioning requirements in a design context. A version model was developed to capture the changes during the evolution of a design artefact that follow during those iterations. The provision of workspaces and classification of version states supports composite object evolution in a cooperative design context. We conclude that *OODB can provide flexible version control that suits the needs of an engineering design environment.*

In chapter 4, we continued by analysing the advantage of using object-oriented views in object database in a design environment. A taxonomy of the view model was presented in this chapter which clearly shows that using object-oriented data model the object-oriented views offer more to design environment than its predecessors. We

showed that *object-oriented views provide a flexible and efficient framework for organising design environments.*

View models offers great flexibility in organising schema and managing data in design environments. However, view classes are computed upon access which imposes overhead on view access time. View materialization is seen as an optimization technique which can improve view performance. In chapter 5, we presented a view materialization strategy that can be applied to versioned objects in a design database. We demonstrated that *view materialization in a design environment provides an effective configuration management scheme.*

Modern design projects are complex as they often require more than one person to work on them. To best support team work in a design environment, the supporting database needs to be partitioned into different levels. In chapter 6 we introduced the concept of three-level workspace into our design environment. To achieve a common design objective, it is usually required to put work of different designers together. This requires configuration management which will enable designers to identify different components of an object.

To avoid the loss of information during object evolution, we developed a unified version management scheme in the partitioned design environment. The unified version management combines object-oriented views with our version model to provide the designers with a consistent version management tool throughout a design environment. The unified version management is achieved by using object-oriented

views. We demonstrated in chapter 6 that *object-oriented views provide a powerful and flexible configuration component selection scheme for configuration management.*

To demonstrate that our model is achievable with existing programming tools, we presented a prototype of our model in chapter 7 and in chapter 8 we evaluate the effectiveness of the model. In these two chapters we have proved that our model does not only exist on paper but also it can be implemented in a real world scenario.

Having met these objectives we have proved the hypothesis.

9.2 Future Work

Following on from the work reported in this thesis, there is scope for continued research in a number of directions.

Firstly, we only investigated the use of instance versioning in design environment. Schema versioning is a potential area to support major design changes in a development environment. As with objects, class schema evolves as well. Simply modifying a class schema will invalidate all previous instances of the class as class modification does not support forward compatibility. The combination of instance versioning and schema versioning will greatly complicate version management in design environment but it also offers better support to iterative design activities.

Secondly, the partition of workspaces could be achieved logically by using database views. The logical partition of workspaces would give the database owner greater flexibility in dynamically control the level of information available to each individual and will also enable him to control the level of abstraction to different users.

Thirdly, further investigate how to react to changes in a configuration management. During a design life cycle, not only the top level object changes. Component objects which are referenced by a complex object also changes. The effects of these changes need to be better understood in order to improve change management in a configuration.

On the implementation side of the project, a more comprehensive message regime can be implemented. The message will provide more information on which attribute has been updated. This information would allow the VMM to decide whether the update will affect any of the view classes based on it. Thus this will further reduce unnecessary re-materialization. To fully assess the benefit of view materialization, a comprehensive performance metric needs to be established. The result of the metrics will provide an indication on what kind of change on base classes will have greater impact on the performance of materialized views. This information will enable us to improve the materialization strategy accordingly.

Finally, the view mechanism facilities are made available through a set of C++ libraries, which the user must link to use. Clearly there is a need to be able to define views using declarative query languages, e.g., OQL, instead of programming

language. This will require the extension of OQL in order to support definition of database views.

Appendix Bibliography

[ABITEBOUL91] Abiteboul, Serge and Bonner, Anthony
Objects and Views.
SIGMOD Record 20, pp 238 - 247.
1991.

[AGRAWAL89] Agrawal, Rakesh and Jagadish, H.V.
On Correctly Configuring Versioned Objects
Proceedings of the 5th International Conference on VLDB
1989

[AGRAWAL94] Agrawal, Rakesh and DeMichiel, Linda G
Type Derivation Using the Projection Operation
Proceedings of International Conference on Extending Database Technology, pp 7 - 14
1994

[AHMED91a] Ahmed, Shamim; Wong, Albert; Sriram, Duvvuru; Logcher, Robert
A Comparison of Object-Oriented Database Management Systems for Engineering Applications
Massachusetts Institute of Technology Research Report R91-12
1991

[AHMED91b] Ahmed, Rafi and Navathe, Shamkant B
Version Management of Composite Objects in CAD Databases
Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 218 - 227
1991

[ATKINSON89] Atkinson, Malcolm; Bancilhon, Francois DeWitt, David; Dittrich, Klaus; Maier, David and Zdonik, Stanley
The Object-Oriented Database Manifesto
Proceedings of DOOD
1989

[BEECH88] Beech, David and Mahbod, Brom
Generalized Version Control in an Object-Oriented Database
IEEE Transactions on Knowledge and Data Engineering
pp 14 - 22
1988

[BERTINO92] Bertino,Elisa
A View Mechanism for Object-Oriented Databases.
EDBT 3rd, pp 136 - 151.
1992

[BILIRIS89] Biliris, Alexandros
Database Support for Evolving Design Objects
Proceedings of 26th ACM/IEEE Design Automation Conference, pp 258-263
1989

[BOTZER96] Botzer, David and Etzion, Opher
Optimization of Materialization Strategies for Derived Data Elements
IEEE Transactions on Knowledge and Data Engineering
Vol. 8; pp. 260-272,
1996

[BRATSBERG92] Bratsberg, Svein Erik
Unified Class Evolution by Object-Oriented Views.
ER 11th, pp 423-439.
1992

[BROWN88] Brown, AW
A View Mechanism for An Integrated Project Support Environment
Ph.D. Thesis
Computing Laboratory
University of Newcastle Upon Type
1988

[BROWN89] Brown, AW
Object-Oriented Databases and their use within an Integrated Project Support Environment.
University of York Technical Report YCS124
1989

[BTSDYRTH92] Btsdyrth, Svein Erik
Unified Class Evolution by Object-Oriented Views
Proceedings of International Conference on the Entity Relationship Approach, pp 421-439
1992

[CARNDUFF93] Carnduff, T W and Gray, W A
Function Materialization Through Object-Versioning in Object-Oriented Database

BNCOD 11, Springer_Verlag, pp 111-128.
1993

[CARNDUFF94] Carnduff, T W
Supporting Engineering Design with Object-Oriented Databases
Ph.D. Thesis
University of Wales College, Cardiff
1994

[CATTELL97] Cattell, R.G.G.
The Object Database Standard: ODMG - 93 Release 2.0
Morgan Kaufmann Publishers, Inc
1997

[CERI91] Ceri, Stefano and Widom, Jennifer
Deriving Production Rules for Incremental View Maintenance
Proceedings of 17th VLDB, pp 577-589
1991

[CHEVAL90] Cheval, Jean Louis
A Version Model for Object-Oriented Databases
Proceedings of the 8th British National Conference on Databases
1990

[CHOU86] Chou, Hong-Tai and Kim, Won
A Unifying Framework for Version Control in a CAD Environment
Proceedings of VLDB
1986

[COLBY96] Colby, Latha S; Griffin, Timothy and Libkin, Leonid
Algorithms for Deferred View Maintenance
Proceedings of ACM SIGMOD, pp 469 - 480
1996

[CONRADI96] Conradi, Reidar and Bernhard Westfechtel
Version Models for Software Configuration Management
Technical Report AIB96-10
Norwegian University of Science and Technology
1996

[DATE95] Date, CJ
An Introduction to Database Systems (6th Edition)
Addison-Wesley
1995

[DAYAL89] Dayal, Umeshwar
Queries and Views in an Object-Oriented Data Model
Proceedings of the 2nd International Workshop on Database Programming Languages
pp 80-102
1989

[DITTRICH88] Dittrich, Klaus R and Lorie, Ramond A
Version Support for Engineering Database Systems
IEEE Transactions on Software Engineering
Vol. 14, No. 4, pp 429 - 437
1988

[GUPTA93] Gupta, Ashish; Mumick, Inderpal S and Subrahmanian, V.S.
Maintaining Views Incrementally,
SIGMOD Record; vol. 22, pp. 157-166;
1993

[GUPTA95] Gupta, Ashish; Mumick, Inderpal S and Ross, Kenneth A
Adapting Materialized Views After Redefinitions
SIGMOD Record, vol. 24; pp. 211-222;
1995

[HANSON87] Hanson, E N
A Performance Analysis of View Materialization Strategies
SIGMOD, pp 440-453
1987

[HEILER90] Heiler, Sandra and Zdonik, Stanley
Object Views: Extending the Vision.
IEEE International Conference on Data Engineering. pp 86-93
1990

[HEILER91] Heiler, Peter H
Configuration Management Models in Commercial Environments
Software Engineering Institute Technical Report
CMU/SEI-91-TR-7
Carnegie Mellon University
1991

[KATZ87] Katz, RH
Managing Change in a Computer-Aided Design Database
Proceedings of VLDB, pp 455 - 462
1987

[KATZ90] Katz, R. H.
Towards a Unified Framework for Version Modeling in Engineering Databases
ACM Computing Surveys, Vol. 22, No. 4, pp 375 - 408
1990

[KATZ97] Katz, R. H. and Chang, E
Managing Change in a Computer-Aided Design Database
Proceedings of the 13th VLDB Conference, pp 455-462
1987

[KEMPER91] Kempter, Alfon; Kilger, Christoph and Moerkotte, Guido
Function Materialization in Object Bases
ACM SIGMOD Record
Vol. 20, Issue 2, pp 258- 267
1991

[KEMPER94] Kempter, Alfon; Kilger, Christoph and Moerkotte, Guido
Function Materialization in Object Bases: Design Realization and Evaluation
IEEE Transaction on Knowledge and Data Engineering
vol. 6 ; no. 4; 587 - 608
1994

[KENT79] Kent, W
Limitations of Record-Based Information Models
ACM Transactions on Database Systems
Vol. 4 No. 1
March 1979

[KIM88] Kim, Hyoung Joo
Issues in Object-Oriented Database Schemas
Ph.D. Thesis
University of Texas at Austin
1988

[KIM90] Kim, Won
Introduction to Object-Oriented Databases
The MIT Press
1990

[KIM95] Kim, Won and Kelley, William
On View Support in Object-Oriented Database Systems
Modern Database Systems - The Object Model, Interoperability, and Beyond
Chapter 6, pp 108-129
ACM Press, 1995

[KUNO95a] Kuno, H.A. and Rundensteiner, E.A.
Materialized Object-Oriented View in MultiView;
RIDE-DOM, pp. 78-85;
1995

[KUNO95b] Kuno, H.A.; Ra, Young-Gook and Rundensteiner, E.A.
The Object Slicing Technique: A Flexible Representation and its Evaluation;
University of Michigan Technical Report,
1995

[LIU94] Liu, Chien-Tsai; Chrysanthis, Panos K and Chang, Shi-Kuo
Database Schema Evolution through the Specification and Maintenance of Changes on Entities and Relationships
Proceedings of the 13th International Conference on Entity-Relationship Approach,, pp 132-149
1994

[LU95] Lu, James J; Moerkotte, Guido and Schue, Joachim and Subrahmanian, V.S.
Efficient Maintenance of Materialized Mediated Views;
SIGMOD Record, vol. 24; pp. 340 351;
1995

[MARIANI93] Mariani, J. A.
Realizing relational style operators and views in the Oggetto object-oriented database system.
Information and Software Technology 35 April, pp 207-216.
1993

[MONK92] Monk, Simon and Sommerville Ian
A Model for Versioning of Classes in Object-Oriented Databases
Proceedings of 10th BNCOD, pp 41 - 58
1992

[MONK93] Monk, Simon and Sommerville, Ian
Schema Evolution in OODBs Using Class Versioning
SIGMOD Record, Vol. 22, No. 3, pp 16-22
1993

[MOTSCHNIG96] Motschnig-Pitrik, Renate
Requirements and Comparison of View Mechanisms for Object-Oriented Databases.
Information Systems, Vol. 21
1996

[ODBERG95] Odberg, Erik
MultiPerspectives: Object Evolution and Schema Modification Management for Object-Oriented Databases
Ph.D. Thesis
Norwegian Institute of Technology
1995

[OMG97] Object Management Group
A Discussion of the Object Management Architecture
Available from http://www.omg.org/ [Accessed August 1997]
1997

[OUSSALAH93] Oussalah, C; Talens G and Colinas, MF
Concepts and Methods for Version Modeling
IEEE Transactions on Knowledge and Data Engineering
pp 332 - 337
1993

[PARK95] Park, Hyun-Ju and Suk, I
Implementation of a Version Manager on an Object-Oriented Database Management System
Proceedings of International Conference on Object-Oriented Information Systems
1995

[PIROTTE94] Pirotte, Alain; Zimányi, Esteban; Massart, David and Yakusheva, Tatiana
Materialization: A Powerful and Ubiquitous Abstraction Pattern.
Proceedings of VLDB, pp. 630-641;
1994

[PRESSMAN94] Pressman, Roger S.
Software Engineering - A Practitioner's Approach
Third Edition (European Edition)
McDraw-Hill Book Company Europe
1994

[RA95] Ra, Young-Gook and Rundensteiner, Elke A
A Transparent Object-Oriented Schema Change Approach Using View Evolution
IEEE Transactions on Knowledge and Data Engineering
Vol. 11; pp 165 - 172
1995

[RODDICK96] Roddick, John F
A Survey of Schema Versioning Issues for Database Systems
Information and Software Technology

Vol. 37, pp 383-393
1996

[SCHOLL91] Scholl, M.H.; Laasch, C and Tresch
Updatable Views in Object-Oriented Databases.
Proceedings International Conference on Extending Database Technology
1992

[SCIORE91] Sciore, Edward
Multidimensional Versioning for Object-Oriented Databases
Proceedings of DOOD'91, pp 355 - 370
1991

[SCIORE94] Sciore, Edward
Versioning and Configuration Management in an Object-Oriented Data Model
VLDB Journal, Vol. 3
1994

[STAUDT96] Staudt, Martin and Jarke, Matthia
Incremental Maintenance of External Materialized Views;
VLDB; pp. 227-238;
1996

[STONEBRAKER90] The Committee for Advanced DBMS Function
The Third-Generation Database System Manifesto
SIGMOD RECORD, VOL. 19, pp 31-44
1990

[TALENS93] Talens, G; Oussalah, G and Colinas, M. F.
Versions of Simple and Composite Objects
Proceedings of the 19th VLDB Conference
1993

[ZDONIK86] Zdonik, Stanley B
Version Management in an Object-Oriented Database
Proceedings International Worship on Advanced Programming Environments, pp 405-422
1986

[ZELLER95] Zeller, Andreas
A Unified Configuration Management Model
Informatik-Bericht Nr. 95-03
Institut für Programmiersprachen und Informations Systeme
Technische Universität Nraiscjweog
1995