Designing Role-Based View for Object-Relational Databases

Ling Wang

B.Sc.

A thesis submitted for the degree of

MASTER OF SCIENCES

to the

DCU

Dublin City University

School of Computing

Supervisor: Mark Roantree

June 2003

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of M.Sc. is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed	[illegible]
Student ID	51186403
Date	June 2003

Acknowledgments

I would like to express my sincere thanks to my supervisor Dr. Mark Roantree, for his help, interest and encouragement over the last sixteen months.

I own a large debt of thanks to the members of Interoperable Systems Group, who was always willing to discuss my problems with technical support and proof-reading. Dalen Kambur, Damir Bećarević and Gerald O'Connor, your help make this thesis possible.

I would like to acknowledge the significant input of Jianming Zou. On many occasions I must have bored him by talking about my research problems I were having with the thesis, but he always listened and made be believe that I could do it. I would like to thank him for his patience, encouragement and faith in me.

Finally, this thesis is dedicated to my parents.

Abstract

In a federated database system, a view mechanism is crucial since it is used to define exportable subsets of data; to perform a virtual restructuring dataset; and to construct the integrated schema. The view service in federated database systems must be capable of retaining as much semantic information as possible. The object-oriented (O-O) model was considered the suitable canonical data model since it meets the original criteria for canonical model selection. However, with the emergence of stronger object-relational (O-R) model, there is a clear argument for using an O-R canonical model in the federation. Hence, research should now focus on the development of semantically powerful view mechanism for the newer model. Meanwhile, the availability of real O-R technologies offers researchers the opportunity to develop different forms of view mechanisms.

The concept of roles has been widely studied in O-O modelling and development. The role model represents some characteristics that the traditional O-O model lacked, such as object migration, multiple occurrences and context-dependent access. While many forms of O-O views were designed for the O-O canonical model, one option was to extend the O-O model to incorporate a role model. In a role model, the real entity is modelled in the form of a role rather than an object. An object represents the permanent properties of an entity is a root object; and an object represents the temporary properties of an entity is a role object.

The contribution of this research is to design a view system that employees the concept of roles for the O-R canonical model in a federated database system. In this thesis, an examination of the current O-R metamodel is provided first in order to provide an environment for recognising the role-view metadata and measuring the view performance; then a Roleview Definition Language (RDL) is introduced, along with the semantics for defining virtual classes and generating virtual extents; finally, a working prototype is provided to prove the role-based view system is implementable and the syntax is semantically correct.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The concept of federated database systems is where heterogeneous databases can communicate with each other through an interface provided by a canonical model. A federated database system is structured as follows: data resides in heterogeneous databases or information systems; each participating database schema is translated to a canonical model named the component schema; view schemas are defined as subsets of the component schema, and shared with other databases or information systems; view schemas are exported to a global or federated server where they are integrated to form multiple global or federated schema. With a highly flexible architecture, the federated database systems not only provide location transparency but contain advantages of conventional view systems such as a virtual restructuring of the physical datasets to meet the requirements of different users.

Many federated database systems development tasks have been addressed, including schema translation, access control, negotiation and schema integration [SL90]. A view mechanism is crucial since it is used to define exportable subsets of data; to perform a virtual restructuring dataset; and to construct the integrated schema. The view service in federated database systems must be capable of retaining as much semantic information as possible, for example it should retain information regarding inheritance and associations between classes.

As the form of object-oriented (O-O) federated database system [BE96] is a specialised form of federated database systems, ODMG model [CB97], the standard model for O-O databases, is used to represent the component schema in many proposals [PBE95, BE96, Rad96, KR01, RKB01]. However, as the popularity of the ODMG model declines, and with the emergence of stronger object-relational (O-R) models (Oracle 9i for example), there are now genuine options for canonical models using the original criteria for canonical model selection [SCGS91]. Hence, research should now focus on the development of semantically powerful view mechanism for the newer model. Meanwhile, the availability of real O-R technologies such as Oracle 9i offers researchers the opportunity to develop different forms of view mechanisms.

1.1 A Federated Database Architecture

Sheth and Larson [SL90] proposed a five-level architecture for federated database systems (FDBS) which is now widely accepted standard architecture for these systems. The five-level architecture is illustrated in *figure 1.1:*

- **Local Layer.** The local schema contains each participating databases or information systems. It describes the logical schema of an autonomous database which may be unaware of

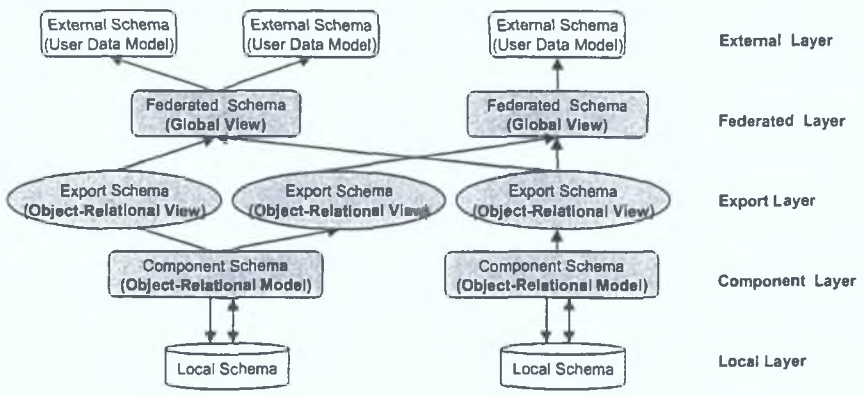

External Schema (User Data Model)
External Schema (User Data Model)
External Schema (User Data Model)
External Layer
Federated Schema (Global View)
Federated Schema (Global View)
Federated Layer
Export Schema (Object-Relational View)
Export Schema (Object-Relational View)
Export Schema (Object-Relational View)
Export Layer
Component Schema (Object-Relational Model)
Component Schema (Object-Relational Model)
Component Layer
Local Schema
Local Schema
Local Layer

Figure 1.1: Five-Level Architecture of FDBS.

the existence of federated database operations [SL90].

- **Component Layer.** The component schema contains the transformed local schema and is presented in terms of the canonical model. The O-R model is considered the canonical model since it meets the original criteria for canonical model selection [SCGS91]. It plays two specific roles: the transformation of data from local schema representation to the O-R representation; and the retrieval and updating of data in the local schema. The O-R model that the component schema presents must supply a powerful view mechanism in order to build the exportable schema. Although the current O-R view mechanism does not fully support the *semantic relativism* metric [SCGS91], it provides an opportunity to develop a richer and powerful view service.

- **Export Layer.** Not all data of a component schema may be available to the federation and its users. The export schema represents a subset of a component schema that is available to the federation [SL90]. An export schema is considered local view schema, which is shared with other databases or information systems. The view mechanism performs virtual restructuring datasets of component schema. A filtering process is used for checking queries and enforcing access restrictions on specific data in order to define views.

- **Federated Layer.** The federated schema is an integration of multiple export schemas. Hence, a federated schema is also considered global view schema. Data from the export schema is merged using a constructing processor to form the federated schema or global schema. The view mechanism plays an important role of constructing the integrated schema. Command decomposing and data merging are the functionalities that the view mechanism must supply. Our research currently deals with the issue of local view schema, defining the federated schema is for further research.

- **External Layer.** The external schema contains a subset of the federated schema by using filtering processor (if required). In this schema, a further data model transformation is required for the purpose of translating the global schema to an end-user preferred data model. Since the Extensible Markup Language (XML) [Gro01] represents a standard for encoding an distributing data portable across various platforms, it is an option that the data is modelled in the XML format in the external schema.

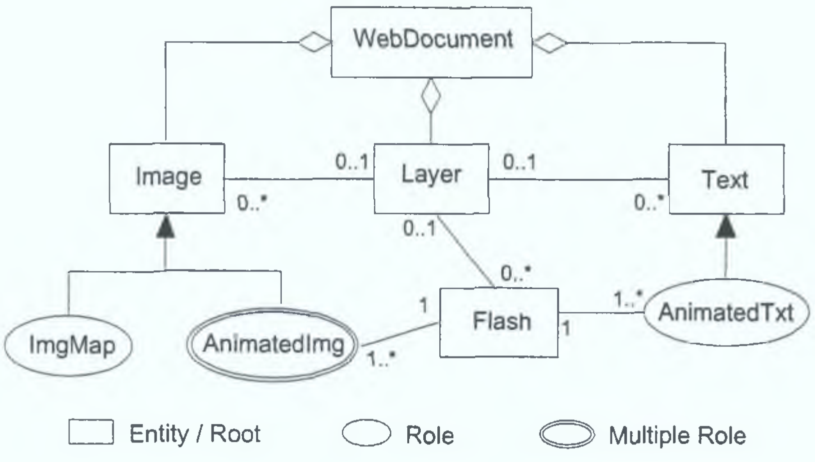


Figure 1.2: Website Schema with Role Extensions.

1.2 Object-Oriented Views

In a FDBS, a view mechanism must contain powerful features because it is used to define exportable schemas; to perform virtual restructuring of datasets; and to integrate global or federated schemas. Different forms of O-O views [SLT91, Run92, SAD94] try to provide the semantic restructuring power for canonical model requirements [SCGS91]. Although there is no standard solution to properly address the metamodel or behaviour issues, the O-O view mechanisms provide the rigidity that satisfies the requirements of canonical model in a FDBS. Those features are examined as the following:

- Base class and virtual class are separated. The issue of placement is a great concern if a defined virtual class is included in the base hierarchy. In a class hierarchy, each class has its own extent and no overlap exists between class extents. However, with the virtual class inside the class hierarchy, it is not possible to ensure that query results are disjoint. A separated hierarchy of base and virtual classes must exist in the O-O model. Thus, any view mechanisms for O-R databases should maintain a separate hierarchy for the O-R base and virtual types.

- Object preserving semantics is used to represent O-O views. These semantics are used to bind the base and virtual classes and perform updates. With object preserving semantics, persistent references are generated for base classes only. It guarantees correct mappings between virtual and base objects, and subsequently provides a reliable update mechanism. Where references to virtual objects are required, the base references are used.

- A view is considered a virtual schema rather than a virtual class. A FDBS requires its view mechanism retain as much semantic information as possible. This requirement emerged from the study on FDBS in [SCGS91]. Hence, most researchers defined an O-O view as a virtual schema rather than a virtual class. It is more powerful if it permits multiple classes with a single view. While defining the O-O view mechanism, a proposed query is used to generate the extents for included virtual classes.

1.3 Object-Oriented Roles

While defining a view mechanism for the O-O data model, one option is to extend the data model to incorporate a role model. The real world entity is modelled in the form of a role rather than an object. In this section, we describe the concept of a role and demonstrate the semantic power that the O-O data model cannot represent. *Figure 1.2* presents an sample schema with role extensions. A web document is represented by `WebDocument` class in the website schema. It is composed of classes `Image`, `Layer` and `Text`. A `Layer` class associates with classes `Image`, `Text` and `Flash`. An `Image` class is the root class of role classes `ImgMap` and `AnimatedImg`. In other words, an `Image` object is allowed to be view as an `ImgMap` object and an `AnimatedImg` object at the same time. The role class `AnimatedImg` is represented as a multiple role class, where the multiple occurrences of a single role object is allowed. It associates with `Flash` class while an `AnimatedImg` object participating in a `Flash` object. The `Text` class is also a root class of role class `AnimatedTxt` because we assume a `Text` object is considered an `AnimatedTxt` role when it appears in a `Flash` object. In this sample schema, there are two root classes exists, which are `Image` class and `Text` class, along with three role (including multiple role) classes, which are `ImgMap`, `AnimatedImg` and `AnimatedTxt`.

The *Role data model*, an extension of the network model, is credited as the first data model that introduced an explicit notion of roles [BD77]. A role is a concept that lacks semantic rigidity. For a concept to be a role, it is always in a relationship with other roles or entities; and represents only the extrinsic properties of the real-world entity. Individuals can enter and leave the extent of the concept without losing their identities [FBCP01]. On the other hand, a natural type (e.g. class or type) is characterised by semantic rigidity, it represents the intrinsic properties of the real-world entity. An individual of a natural type cannot drop its type without losing its identity. A role is considered the temporary aspect of a natural type. For example, in *figure 1.2,* `AnimatedImg` is defined as a role since to be an `AnimatedImg` the appearance in a `Flash` is required, and the disappearance does not lead to a loss of its root identity. In other words, an `Image` object still exists even though it does not play a role of `AnimatedImg`. On the contrary, `Image` is a natural type, because an `Image` object will always remain an `Image` and being an `Image` is independent of the existence of any relationships.

1.3.1 Role Features

Many features of roles have been identified in last decade, some conflicting with others, so that there is no single definition of a role [Ste00]. The following are the most important features of roles that have been widely accepted:

- Roles can be acquired and abandoned dynamically. A role represents the extrinsic features of an object due to its participation in an event, and it is created when the participation begins. If the object stops participating, the role may cease to exist and all its properties and behaviour no longer hold. For example, in *figure 1.2,* an `Image` object is allowed to play an `ImgMap` role originally, and gain a new role `AnimateImg` later, while retaining the first role. It is also allowed that an `Image` object loses its `ImgMap` role and gains an `AnimatedImg` role.

- Each role of an object has its own properties and behaviour. A role is used to represent one specific state of a multi-faceted object. The properties and behaviour that the role holds, presents only the extrinsic aspects of the object. In *figure 1.2,* root class `Image` and role classes `ImgMap` and `AnimatedImg` represent different properties and behaviour.

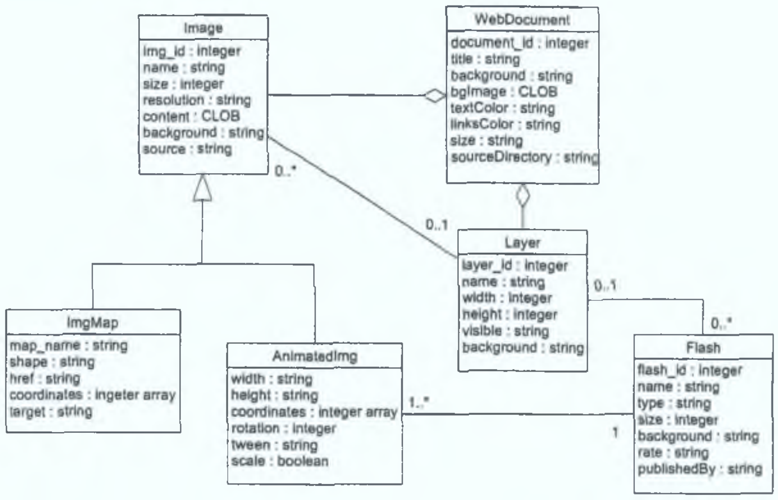


Figure 1.3: Image Schema.

- An object may play different roles simultaneously. This is one of the most broadly accepted properties of the role concept. Since a role is usually regarded as a special 'type', it amounts to the multiple classification of objects. Instead of exclusive and permanent relationships, the relationship between an object and its role is dynamic and temporary. The typical example is that an `Image` object plays role `ImgMap` and role `AnimatedImg` at the same time.

- An object may play the same role several times. This is fundamental concept in the real world with an example being a student registering at several universities. Unlike with different roles however, it does not correspond to multiple classification. This situation is described as multiple occurrences [GSR96]. The main reason for distinguishing multiple occurrences in the same role is that each occurrence of the object in a role is associated with a different state. For example, in *figure 1.2,* an `AnimatedImg` role object appears with different *rotation* in different `Flash` objects.

- A role can be transferred between different objects. It is useful to let a concrete role be dropped by one object, and picked up by other objects, or even to specify the properties of a concrete role without naming a particular role player. For example, the `ImgMap` role can be transferred from one `Image` object to another `Image` object. Note that many role features are transferred without changing, while others must be re-computed in light of the new entity playing the role. For example, in *figure 1.2*, if the *rotation* of an `Image` is '`+90`' for being an `AnimatedImg` role , then the *rotation* property must be recomputed should that role be transferred.

1.3.2 Role Usage

The role concept was proposed as a way to overcome the limitations of classical object models. It captures evolutionary aspects of real-world objects that cannot be modelled by time-dependent property values and that are not well captured by the generalisation relationship [DPZ02]. Following are the examinations of how the notion of roles relax those limitations:

- **Object Migration.** Consider the sample schema presented in *figure 1.3*, where an `Image` object cannot be modelled as an `AnimatedImg` object if it is originally defined as an `ImgMap`

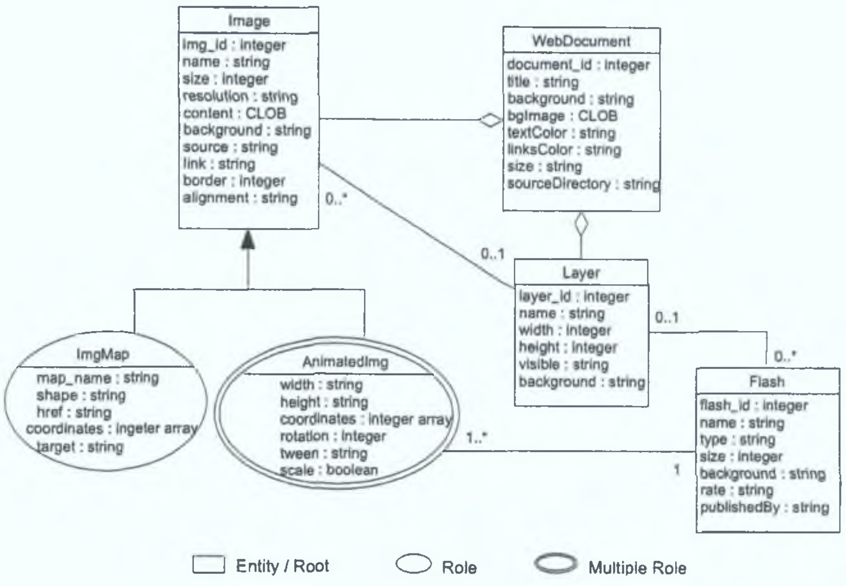


Figure 1.4: Image Schema with Role Extensions.

object. In the O-O model, each object is identified by a unique object identifier. Hence, an **Image** object '*ISG*' is identified by different identifiers when it moves from being an **AnimatedImg** object to an **ImgMap** object. This problem is resolved by using the role concept in object modelling. *Figure 1.4* illustrates that **Image** is specified as an object (root) class, and **ImgMap** and **AnimatedImg** are two role classes of **Image** class. According to the feature of the roles, an object can acquire or abandon the roles dynamically without changing the identity of that object. **Image** '*ISG*' now can be represented by an **Image** object with an **ImgMap** role in the beginning; and with an extra role **AnimatedImg** afterwards.

- **Multiple Occurrences.** In the real world, it is possible that multiple occurrences of the same object exist. Considering the previous example again, **Image** object '*ISG*' is an animated image and represented as an instance of **AnimatedImg** class. However, it later participates in another **Flash** object as an animated image. In the O-O model, an object becomes an instance more than once of the same class is not possible. One of the role features is that an object can play the same role many times simultaneously. Hence, in *figure 1.4*, **Image** '*ISG*' is represented as an **Image** object with one **AnimatedImg** role, where allows multiple occurrences. In other words, representing the same object as more than one instance of the same class (role class) is possible.

- **Context-dependent Access.** The O-O model has no ability to view a multi-faceted object in a particular perspective [DPZ02]. Considering the previous example again, if **Image** object '*ISG*' is specified as an **ImgMap** object, it can never be viewed as an **AnimatedImg** object. This obstacle is released with the concept of roles. In *figure 1.4*, **Image** '*ISG*' is allowed to be observed separately either as an **ImgMap** role or an **AnimatedImg** role.

1.4 Motivation & Contribution

In a FDBS architecture, it is necessary for each participating system to provide a description (view definition) of its shareable data in a semantically rich manner. The O-O model offers some semantic power by defining object views. However, a role mechanism may provide a flexibility while retaining the expressive qualities. The original criteria for canonical model selection [SCGS91] has shown that an object based model is the most suitable data model. The O-R model, another object based data model, has been strongly developed in last few years, and there is a clear argument for using an O-R canonical model because of its increasing similarity to O-O model and its more widespread acceptance (Note: In this research, we regard the latest version of Oracle to be the O-R standard). In the O-R model, similar objects are grouped into a *type*, which defines the structure and behaviour of its instances. Types are organised into a type hierarchy, where the structure and behaviour of several types can be abstracted into a common supertype. Generalisation and specialisation are the common properties of the O-R model. The *collection type* in the O-R model is used to represent the aggregation and composition relationships. While defining a type in the O-R model, the *method* is declared to implement the behaviour that users want objects of that type to perform [Ora01]. Furthermore, the view mechanism supplied by the O-R model provides an opportunity to develop a richer and more powerful view service to meet the canonical model requirements.

The O-O model and the O-R model present similar suitability as canonical models in the federation. In our research, the O-R model is considered the canonical model because it provides new experiments. The motivation of our research is to define a new view system which is based on the concept of roles in the O-R data model. Our contribution is 2-fold: to examine the deployment of the latest O-R model (both as a model and metamodel) and to specify and implement a view mechanism which is role-based. With the examination of the model, we outline its suitability to meet the canonical model requirements, and also clarify the possibility of extending its metamodel to facilitate roles. Specifically, an extended Structured Query Language (SQL:1999) [GP99] is used to provide the specifications and mappings to support restructuring, and to generate extents for the virtual classes of view mechanism.

1.4.1 Issues Regarding Terminology

This research describes the deployment of a role-based view system for O-R databases. It provides a new look at an alternate view mechanism, which is based on the O-R industry standard. A *relation* is the only possible structure in the relational data model. Hence, a base relation is used to refer to a relation containing physical data, and a virtual relation is a relation derived using a query on base or virtual schema. The relational model handles real world entities by modelling them as relations. The flat feature of the relational model makes it impossible to handle complex objects and object hierarchy. However, the object models (O-O and O-R) contains complex objects and a view may involve the construction of a single virtual object of multiple virtual objects. Some researchers regard a view as a single virtual class, and others regard a view as a virtual schema.

In our research, a view is regarded as a virtual schema to meet the original criteria of canonical model in FDBS. The term of *roleview* is used to represent a new O-R view mechanism that contains multiple virtual classes. The term of *root* is used to specify the static component of the view, which presents the intrinsic properties of an object, and *role* is the term that specifies the dynamic component of that view, which presents the extrinsic properties.

1.5 Conclusions & Dissertation Layout

In this chapter a general introduction to FDBS is provided, together with the functionalities that the view mechanism must provide. The existing O-O view mechanisms present three main features, which are base class and virtual class must be separated; object preserving semantics is used to represent the views; a view is a virtual schema style rather than a virtual class style. We also studied the notion of role concept and role usage. The O-R model is regarded as the canonical model since its popularity is growing, and finally the concept of roles is used to build a new view mechanism rather than the traditional O-O views.

In chapter 2, we examine some projects that have implemented the role concept in practice. The contribution and limitations of each project are listed, while the differences to our approach are also clarified.

In chapter 3, we analyse the current O-R metamodel. A full description of the metaclasses are presented. Before extending the metamodel, a study of our view metamodel is introduced. The extended metamodel provides the capability of recognising roles and roleviews. Finally, we briefly describe the extended metamodel.

In chapter 4, we present our view model. The semantic issues are discussed and a complete specification is provided. The extended SQL:1999 is used to define the mechanism, and a view display system uses the methodology for extents to display views.

In chapter 5, we introduce the details of prototype and the experiments are provided. In chapter 6 we conclude the thesis and discuss the future work.

Chapter 2

Related Research

O-O modelling and development have been widely studied in research and developed in industry. However, there are still some problems requiring for solutions. One of these problems, the lack of adequate object evolution mechanisms, also knowing as role modelling, has attracted the attention of many researchers. Different proposals for extending the traditional object model with role mechanisms have been published. As our view system is built on the base of role concept, it is necessary to study the existing role proposals and present a comparison of their features. In this chapter we clarify the differences between those proposals and our solution. Furthermore, we list the functional requirements that a suitable view mechanism of federated systems must represent.

For many years object roles have been studied in O-O literature. Some researchers concentrated on theoretical aspects [RS91, ABGO93, Ste00, DPZ02], while others focused on implementation [GSR96, Won98, AAG00, JHPS02]. In this chapter, we take a close look at some recent projects which represent roles in practice. Each project is studied according to the following criteria: the contributions and the limitations, the context of roles and views (if present) and the beneficial features that we may use. All examples illustrated in this chapter are based on the schema in *figure 1.4*.

2.1 Extended Smalltalk

Gottlob et al. [GSR96] extended the existing O-O programming language *Smalltalk* by adding classes such as `ObjectWithRoles` and `RoleType` to support the role mechanism similar to OBD [KS91]. The root of the role hierarchy is represented as a subclass of `ObjectWithRoles`, and every role type is specified as a subclass of `RoleType`. The object migration and context-dependent access issues are relaxed in this approach. The prime contribution of the work is that multiple occurrences is enabled by defining a new type `QualifiedRoleType`. With that type, it is possible to model a real world entity that has several occurrences of one single role. To be able to distinguish the different occurrences of a role type of a single object, the occurrence has to be uniquely identified via a qualifying attribute. The following examples illustrate the definitions of object with role extensions in Extended Smalltalk.

Example 2.1 *Smalltalk Root Definition.*

```
// Root Type Definition
ObjectWithRoles              // start the root type definition
    subclass: #Image          // root Image is a subclass of ObjectWithRoles
    instanceVariableNames: 'img_id name size resolution content
                            background source'
    classVariableNames: ' '    // variables at the class level are null
    poolDictionaries: ' '      // string of pool names
    category: 'ImageView'.     // category name string


// Root Object Definition
ISG <= Image new.          // create a new object of Image type
    ISG img_id: 1001.      // assign values to instance variables
    ISG name: ' ISG'.
    ISG size: ' 60k'.
    ISG resolution: ' 144p'.
    ISG content: ' logo'.
    ISG background: ' white'.
    ISG source: ' c:\\Graphics\\'.
```

In *example 2.1*, root type and root object are defined. The root type `Image` is defined as a subclass of `ObjectWithRoles`. The root class represents the supertype in a role hierarchy named `ImageView`, and it is also provided as a parameter in addition to the role-specific instance variables. The root object `ISG` is defined by sending the message `Image new`. The definitions of role type and role object are illustrated in *example 2.2*.

Example 2.2 *Smalltalk Role Definition.*

```
// Role Type Definition
RoleType                        // start the role type definition
    defRoleType: #ImgMap      // ImgMap is a subclass of RoleType
    instanceVariableNames: 'map_name shape href coordinates target'
    classVariableNames: ' '    // variables at the class level are null
    poolDictionaries: ' '      // string of pool names
    category: 'ImageView'      // category name string
    roleSuperType: #Image.   // specify its root type


// Role Object Definition
// create new role object, which corresponds to ISG root object
ISGImgMap <- ImgMap newRoleOf: ISG.
    ISGImgMap map_name: 'isg_ban'.   // assign values to instance variables
    ISGImgMap shape: 'oval'.
    ISGImgMap href: 'www.google.com'.
    ISGImgMap coordinates: {125, 50}.
    ISGImgMap target: 'self'.
```

The role type definition starts with the message `defRoleType`, and `roleSuperType` is used to present the supertype in the role hierarchy. In the previous example, class `Image` is specified as the supertype of role type `ImgMap`. The supertype of a role type could be a subclass of `RoleType` or `ObjectOfRoles`. In other words, a role can be played by either a root or another role. While defining a role object, the message `newRoleOf` is used for representing an object as the ancestor of the new role type instance. In this case, role object `ISGImgMap` is played by the root object `ISG`.

Example 2.3 *Smalltalk QualifiedRole Definition.*

```
//QualifiedRole Type Definition
QualifiedRoleType                // start the qualified role type definition
// AnimatedImg is a subclass of QualifiedRoleType
    defQualifiedRoleType: #AnimatedImg
    instanceVariableNames: 'width height coordinates rotation tween scale'
    classVariableNames: ' '    // variables at the class level are null
    poolDictionaries: ' '     // string of pool names
    category: 'ImageView'     // category name string
    roleSuperType: #Image     // specify its root type
    classOfQualifyingObj: #Flash.      // specify the qualifier
// QualifiedRole Object Definition
isgLogo <- Flash new name: 'isg_logo'.       // create a new object of Flash
// create new qualified role object, which is qualified by isgLogo object
Animated_isg <- AnimatedImg newRoleOf: ISG qualifiedBy: isgLogo.
...
trLogo <- Flash new name: 'tr_logo'.      // create a new object of Flash
// create new qualified role object, which is qualified by trLogo object
Animated_tr <- AnimatedImg newRoleOf: ISG qualifiedBy: trLogo.
```

In the final example, the qualified role type and object are defined. The attribute `clssOfQualifyingObj` is provided as a parameter in addition to the role-specific instance variables and instance methods. The method `newRoleOf: anObject qualifiedBy: qualifyingObj` is predefined for creating new qualified role objects.

Extended Smalltalk supports creating new objects based on the role concept rather than implementing this concept with the current O-O model. The authors try to model objects in an object-role format rather than in O-O format. However, our focus is on how to restructure or review the defined objects, which have been stored in O-R databases. Although this proposal resolves many issues that the traditional O-O model presents, the role definition is based on one single individual object. Each root object or role object is generated individually. However, we concentrate on generating a collection of objects with the capability of restructuring in a new view system. The term of `multirole` in our approach is used to address the issue that `QualifiedRoleType` resolved. *Figure 2.1* represents the differences between Extended Smalltalk and our approach. The premier difference is that Extended Smalltalk concentrated on extending the O-O model at the local layer; and we focus on developing a new view service at the view layer by extending the O-R metamodel.

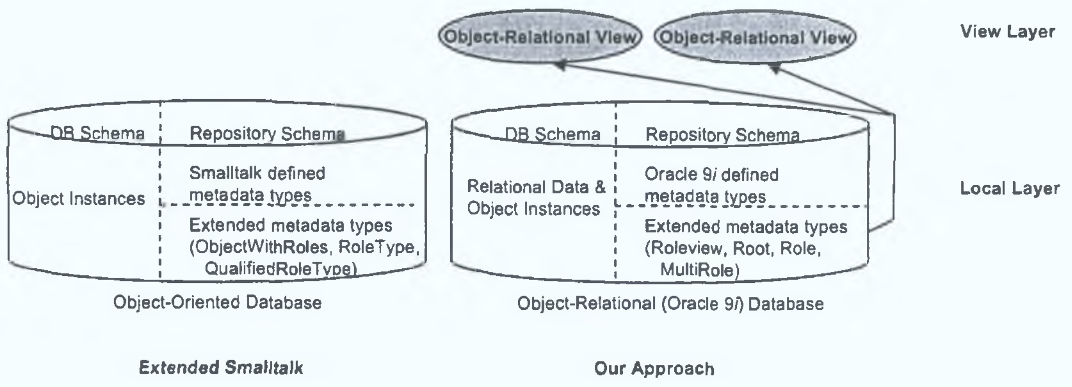

Object-Relational View
Object-Relational View
View Layer
DB Schema
Repository Schema
Object Instances
Smalltalk defined metadata types
Extended metadata types (ObjectWithRoles, RoleType, QualifiedRoleType)
Object-Oriented Database
Extended Smalltalk
DB Schema
Repository Schema
Relational Data & Object Instances
Oracle 9i defined metadata types
Extended metadata types (Roleview, Root, Role, MultiRole)
Object-Relational (Oracle 9i) Database
Our Approach
Local Layer

Figure 2.1: Extended Smalltalk & Our Approach.

2.2 DOOR/MM

Wong et al. proposed a dynamic object-oriented database programming language with role and multimedia extensions [Won98]. The three contributions are: supporting object migration; extending multimedia objects with the notion of roles; and integrating views and roles.

The issue of object migration is resolved by representing roles with the role class names and their values instead of the global unique identifier. Representing roles with a unique identifier offers the following advantage: it distinguishes a role from other roles; it recognises a role as the same role even if its state is changed; it models class migration by adding and deleting roles to an object. However, this unique role identifier scheme still causes problems like dangling references and historical information representation if we allow a reference to a role [WCL97]. For example, if a manager position is replaced by another person, all references to the manager should change to the new manager (who is represented by a different role identifier). Hence, a role object is represented by the combination between role class name and role object value in this approach.

The issue of multiple instantiation is not discussed explicitly. Authors only specify a *boolean* expression in order to distinguish multiple occurrences. In the case of multiple roles satisfying the expression, the system will only return one of them for the sake of simplicity and efficiency.

A newly designed language, the DOOR database programming language (DL), is used to define the roles. Following are the key constructs:

1. (make < `class` > ' subclass-of ...)
2. (make < `virtual-view` > existing-instances ...)
3. (make < `materialised-view` > existing-instances ...)
4. (make an-existing-object-class ' slot 1 ...)
5. (make an-existing-role-class an-existing-instance ' slot 1 ...)

The keyword `make` is used to create a class (either object class or role class), a virtual view, a materialised view, an object, or a role. These constructors are represented by the following examples, which are based on the schema in *figure 1.4*

Example 2.4 *DOOR/MM Root Definition.*

```
// Root Class Definition
(define <Image>     // start the root class definition
    (make <class>
        // specify the attributes
        'slots (list 'img_id 'name 'size 'resolution 'content 'background 'source)
        'label "<Image>")     // specify the label of root class
)
// Root Object Definition
(define ISG     // start the root object definition
    (make <Image>
        'img_id 1001     // assign values to root attributes
        'name "ISG"
        'size "60k"
        'resolution "144p"
        'content "logo"
        'background "white"
        'source "c:\\Graphics\\")
)
```

In *example 2.4*, root class and root object are defined separately. First make is used to create a root class and the second part is used to create a root object.

Example 2.5 *DOOR/MM Role Definition.*

```
// Role Class Definition
(define <ImgMap>     // start the role class definition
    // it is defined as a subclass of Role
    (make <class> 'subclass-of (list <Role>)
        'player-domains (list <Image>)     // specify its root class
        // specify the role class attributes
        'slots (list 'map_name 'shape 'href 'coordinates 'target)
        'label "<ImgMap>")     // specify the label of role class
)
//Role Object Definition
(define ISGImgMap     // start the role object definition
   (make <ImgMap> ISG     // specify its root object
      'map_name "isg_ban"     // assign values to role attributes
```

'shape "oval"

'href "www.google.com"

'coordinates {125, 50}

'target "self")

)

In *example 2.5*, `ImgMap` is defined as a role class which is considered the subclass of `Role`. The message `player_domains` is used to specify the root class. In the role object definition, `ISG` is used to specify the root object which is corresponded to the role object `ISGImgMap`.

In DOOR/MM, views are modelled as multiple representations and abstractions of a multimedia object. Roles are modelled as an object-based specialisation of a multimedia object for dynamic extension, as well as integrating the heterogeneous types of information in the O-O model. In other words, objects and roles (but not views) are regarded as logical entities, and its views are regarded as virtual representations of these logical entities [Won98]. The view definition in this approach is represented by the following example.

Example 2.6 *DOOR/MM View Definition.*

(define ISG-view // start the view definition

(make <virtual-view> (list ISG) // specify the base root object

'img_id 1001 // retrieve the attributes

'name "ISG"

'size "60k"

'resolution "144p"

'content "logo")

)

(define ISGImgMap-view // start the view definition

(make <virtual-view> (list ISGImgMap) // specify the base role object

'map_name "isg_ban" // retrieve the attributes

'href "www.google.com")

)

As *example 2.6* illustrates, a view is specified as a virtual entity of a root or role object. Hence, `ISG-view` represents root object `ISG`; and `ISGImgMap-view` is a virtual representation of role object `ISGImgMap`. While defining views in DOOR/MM, some attributes of base root or role object are allowed to be hid. Each view definition is based on one single object (root or role). However, in our approach, we regard a view as a wrapper of many related logical objects. Although the authors claim that a view may be defined by extracting the abstract and references from other views to represent multifaceted features of a multimedia object, there is no prototype or implementation supported. We differ from this approach by providing a role-based virtual schema rather than just a concept of single root or role. *Figure 2.2* represents the differences between DOOR/MM and our approach. DOOR/MM defines a new object-role model at the local layer and the multimedia-object views are specified based on single root or role object; and the metamodel issue is not properly addressed. Our approach extends the existing O-R metamodel in order to support the view definitions at the view layer.

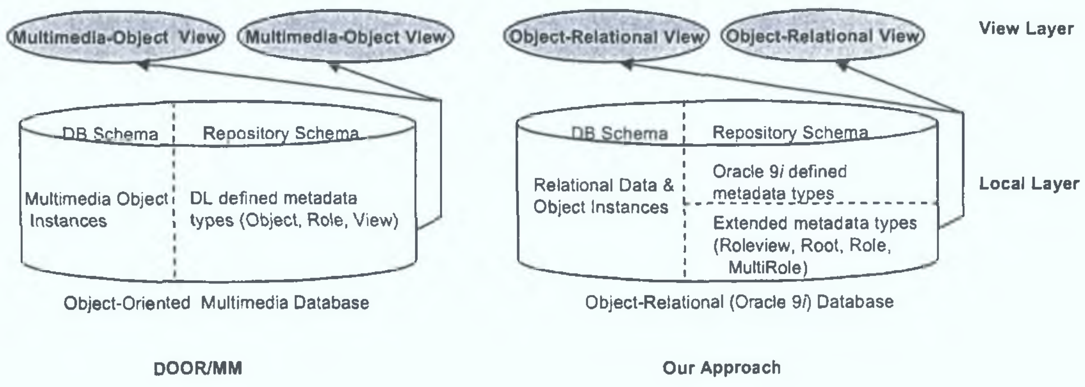

Multimedia-Object View
Multimedia-Object View
DB Schema
Repository Schema
Multimedia Object Instances
DL defined metadata types (Object, Role, View)
Object-Oriented Multimedia Database
DOOR/MM
Object-Relational View
Object-Relational View
View Layer
DB Schema
Repository Schema
Relational Data & Object Instances
Oracle 9i defined metadata types
Extended metadata types (Roleview, Root, Role, MultiRole)
Local Layer
Object-Relational (Oracle 9i) Database
Our Approach

Figure 2.2: DOOR/MM & Our Approach.

2.3 Galileo

Albano et al. focused on developing views for O-O databases with the semantics of viewing operations in the context of Galileo 97, which is a strongly typed database programming language and supports objects with role concept [AAG00]. In Galileo 97, operators are defined in order to create the root and roles : `mkT` is used to construct the object of type `T`; `inS` extends dynamically an object with a new subtype `S` of `T`, without changing its identity, but with the possibility of changing its behaviour; `inS` adds a new role to an object, and returns a reference to this new role of that object. An object expression in Galileo 97 always denotes one specific role of an object. The root and role definitions are illustrated by the following examples.

Example 2.7 *Galileo Root Definition.*

```
// Root Type Definition
let rec type Image <-> [      // start the root type definition
    img_id: int;      // specify the root attributes
    name: string;
    size: string;
    resolution: string;
    content: string;
    background: string;
    source: string ]
;
// Root Object Definition
let ISG := mkImage (      // create a root object ISG
    [ img_id := 1001;      // assign values to root attributes
     name := "ISG";
     size := "60k";
     resolution := "144p";
     content := "logo";
    background := "white";
    source := "c:\\Graphics\\" ] )
;
```

In *example 2.7*, the root type and root object are defined. The root type is defined by passing the message `let rec type` and the root object definition starts with the operator `mkImage`. The following example illustrates the role type and role object definitions. The role type is defined as a subtype of `Image`. The operator `inImgMap` is used to create a role object which is played by `ISG` object.

Example 2.8 *Galileo Role Definition.*

```
//Role Type Definition
let rec type ImgMap <-> is Image and [     // start the role type definition
    map_name: string;     // specify the attributes
    shape: string;
    href: string;
    coordinates: integer array;
    target: string ]
;
//Role Object Definition
    // create role object, specify its root object ISG
let ISGasImgMap := inImgMap (ISG,
    [ map_name := "isg_ban";       // assign values to role attributes
     shape := "oval";
     href := "www.google.com";
     coordinates := {125, 50};
     target := "self" ] )
;
```

In Galileo, authors assert that the role mechanism canot cope with the related problem of giving different views of the same object without affecting its behaviour. The object with role extensions are introduced as *real objects* that have been explicitly constructed using the `mk` or `in` operators, and the views are defined as *virtual objects* that change objects interface. A virtual object has the same identity as the base object; if it is based on a combination of several objects, then its identity is a combination of the identities of base objects. A virtual object can add, remove, and rename fields of its base object, moreover a virtual object can have its own instance variables, which are accessed by its own methods.

The view definition in Galileo starts at the class level. The *view type* specifies the structure of virtual object, and the constructors, such as `project`, `rename`, `extend` and `times`, build virtual objects. *Example 2.9* demonstrates the view definition.

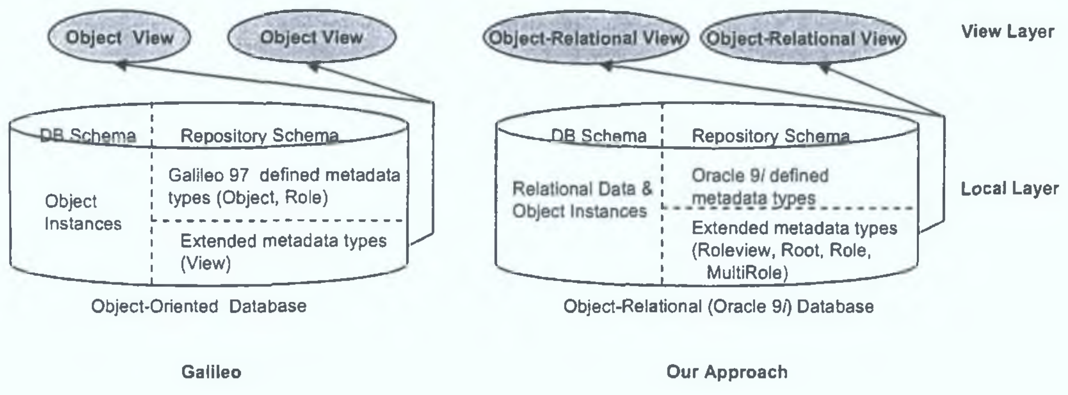

Object View
Object View
DB Schema
Repository Schema
Galileo 97 defined metadata types (Object, Role)
Object Instances
Extended metadata types (View)
Object-Oriented Database
Galileo
Object-Relational View
Object-Relational View
DB Schema
Repository Schema
Oracle 9i defined metadata types
Relational Data & Object Instances
Extended metadata types (Roleview, Root, Role, MultiRole)
Object-Relational (Oracle 9i) Database
Our Approach
View Layer
Local Layer

Figure 2.3: Galileo & Our Approach.

Example 2.9 *Galileo View Definition.*

```
// View Type Definition
let type ImgView :=
    <Image> view [ // specify the base object type, view is type constructor
        name;    // attributes list
        size;
        resolution;
        content]
;
// Vitural Object Definition
// virtual object of ISG, which is defined as an object of Image type.
let ISGImgView :=
    ISG project [    // specify the base object, project is restructure operator
        name;    // selected attributes
        size;
        resolution;
        content ]
;
// virtual object of ISG, which has been extended with the role type ImgMap.
let ISGImgMapView :=
    (ISG as ImgMap) project [    // base object ISG is extended as ImgMap
        map_name;
        href ]
;
```

Authors state that roles and views are common since they both allow an object to be extended. However, roles considered object extensions may modify the behaviour of the original object whereas views do not modify its behaviour. The contribution of this proposal is the clarification of the relationship between roles and views, and the different semantics of method overriding and evaluation in views and roles. The essential differences between roles and views are:

- The set of roles of an object is part of the object itself, and the object can be tested with the predicate `isalso` to find out which role it has; while a view is conceptually external to the object.
- Adding a new role to an object transforms its type into a subtype, while the corresponding view operation `extend` produces an object whose type may not be related to the original one.
- The behaviour of an object changes when it gains a new role, while it is not affected by the creation of a new virtual object.

Although some view operations are specified in this proposal, which present more flexibility than the DOOR/MM proposal, the view is still based on one single object rather than a collection of objects. Furthermore, the premise of defining a view is that the object has to be well specified with role extensions by using Galileo 97. In other words, if the stored object is not modelled in the object-role format, the view of that object cannot be implemented and view operators have no use. *Figure 2.3* represents the differences between this approach and our solution. The object view specified in Galileo is based on the extension of object-role moel, which is defined by Galileo 97; and it is the virtual representation of single object. We differ because we extend the standard O-R model with a standard database programming language SQL:1999 to define the O-R view as virtual schema.

2.4 Summary of Analysis

The role proposals described in this chapter provide different aspects of the role concept, while some of them also discusses the view mechanisms which integrated with roles. In general, there are few broad role features emerged from those definitions:

- A root object plays many role objects at the same time. All the proposals agree that a root object plays many role objects at the same time. The benefits of this feature is that instead of the permanent relationship presented in O-O model, the relationship between a root object and role object is dynamic and flexible. In the real world, there is a possibility that an object does not associate with any roles currently, instead being a potential player in the future. For example, a person object may play a role of student eventually. None of the proposals pay any attention to this possibility, whereas it should be possible in a roleview system.
- Multiple occurrences of roles should be permitted. Both Extended Smalltalk and DOOR/MM discussed this issue, with the latter providing only an expression to distinguish multiple occurrences. Extended Smalltalk represents this issue by defining a new type which is tightly associated with a qualifier type. Multiple occurrence is also a key task in our approach. A role type `multirole` is defined to support multiple instantiation without the associated qualifier type specification. In our appraoch, a `multirole` is considered another form of role rather than a totally different role format, which Extended Smalltalk represented.

- A root object acquires and abandons a role object dynamically. While abandoning a role object, the root object still exists. The deletion of a root object causes the deletion of its role objects. All of the proposals present a loosed coupled relationship between root and roles. A root object is allowed to add or remove its role objects dynamically. This dynamic property comes close to object migration or dynamic re-classification. All of the proposals address that a role cannot be defined unless its root exists in the database schema. In other words, the role existence depends on its root. The root object will not lose its identity when its roles are removed from the schema; conversely, a role object is lost when its root object is deleted from the schema.

2.5 Conclusions

In this chapter, some of the major research projects on roles are examined. According to the emerged key characteristics, together with the analysis of the O-O view mechanisms from chapter one, we provide the functional requirements for a suitable federated view mechanism. These requirements are summarised as the following:

- A roleview is a wrapper of root and roles, which are considered virtual classes. A roleview is defined as a virtual schema rather than a single virtual class. The view mechanism in DOOR/MM is one multimedia-object based. Although the authors assert that the schema-based view is supported also, there is no prototype or implementation provided. The view mechanism in Galileo is also one object based. A view is defined as a virtual root or role. Such representations do not meet the requirements from the study of federated database systems [SCGS91]. In order to retain as much semantic information as possible, our view system wraps multiple related virtual classes (root and role) to represent the underlying schema.

- Object-preserving semantics is an issue. The role projects we studied in this chapter aim to extending the O-O model with role extensions. Hence, object-generating semantics is used to identify the new mechanism. In Extended Smalltalk and Galileo, a unique identifier identifies the role object; in DOOR/MM, the combination of role class name and a pivotal role object value is the role object identifier. On the other hand, object-preserving semantics is an evident in some O-O view mechanisms, such as Cocoon [SLR$^+$94] and Multiview [Run92], where views are defined as the virtual entities of existing objects. In our approach, the roleview is composed of many virtual classes. It is possible that the root object and the role object base on one single entity. In this case, the ambiguities are caused if the identifiers of the virtual objects (root object and role object) are both specified by the object-preserving semantics. A solution to this issue is provided in chapter four, where transient-object-generating semantics is used to identify the role object and object-preserving semantics is used to identify the root object.

- A clear semantics must be provided to generate the view extent. In DOOR/MM, the view mechanism is defined at the object level. Each view definition is considered one single virtual multimedia object. In Galileo, although a view type is defined at the class level, each view object is generated individually. A constructor is provided to generate a single view object rather than collection of view objects in DOOR/MM and Galileo. However, generating the view extent is an issue in some O-O view proposals, where a view is considered virtual schema [SAD94, Run92]. A query is proposed to generate the extents of the virtual classes. However, it is necessary to display a number of classes for which one class determines the extents of all connected classes. In our approach, a roleview is also considered virtual schema. Hence, the questions about generating extents arise, such as how to generate the extent of each virtual class; how to join these extents and generate the extent for the entire virtual schema; and how to avoid the overlap between each virtual class extent. A clear semantics is provided in the roleview definition in order to clarify the extents specifications. A full description of business rule and semantics are introduced in chapter four.

At this point, our roleview system must provide these functional requirements, along with presenting role features listed in the summary of analysis. Before the new view system is introduced in chapter four, it is necessary to study and extend the existing O-R metamodel in order to support our roleview metaclasses.

Chapter 3

Extending the O-R Metamodel

Most of the latest versions of relational databases, such as Oracle, Sybase and Informix extend the relational model with new constructors to support objects. In general, these databases have appeared in the market before the approval of the standard, hence the current version of O-R databases do not fully support the SQL:1999 [GP99] specification. In our research, Oracle9*i* [Ora01], the latest version of the Oracle database, is considered the standard model because it supports most features of the SQL:1999 specification. The purpose of this research is to specify a role-based view system in O-R databases, hence, it is necessary to examine the O-R metamodel to see how it might support roles. If not, it is necessary to extend the metamodel which will support the roleview definition and store new metadata in the schema repository. From now on we will refer to Oracle9*i* as the O-R database and Oracle9*i* metamodel as the standard O-R metamodel.

3.1 The Object-Relational Metamodel

As an amalgamation of relational and O-O data models, the O-R model is complex. Many applications need access to the complex structures through metadata. A metadata model provides interfaces for extracting complete definitions of logical database objects. The O-R database stores metadata in the schema repository as static tables and views [Ora01]. The base tables store information about the database and only the vendor may access these tables. However, views summarise and display the information stored in the base tables and decode the base tables into useful information for metadata queries. Users are allowed to observe metatables by accessing these views.

Figure 3.1 presents an overview of the O-R metamodel. In an O-R metamodel, a schema is a collection of structured data or schema objects. Schema objects (named as `ALL_OBJECTS` in the schema repository) are created and manipulated by SQL and stored as metadata. Schema objects include many structures, such as types, tables, views, triggers, sequences, stored procedures, indexes, synonyms, clusters and database links etc. Our research focuses on the study of the logical structure of databases, hence the schema objects which relate to the physical structures not discussed. A full version of O-R metamodel analysis is presented in [Wan02a]. The relevant schema objects are classified in four sections: types, tables, views and triggers. In *figure 3.1*, `ALL_OBJECTS` plays the role of container, holding all schema objects in the database. `ALL_TYPES`, `ALL_TABLES`, `ALL_VIEWS` and `ALL_TRIGGERS` represent different types of objects.

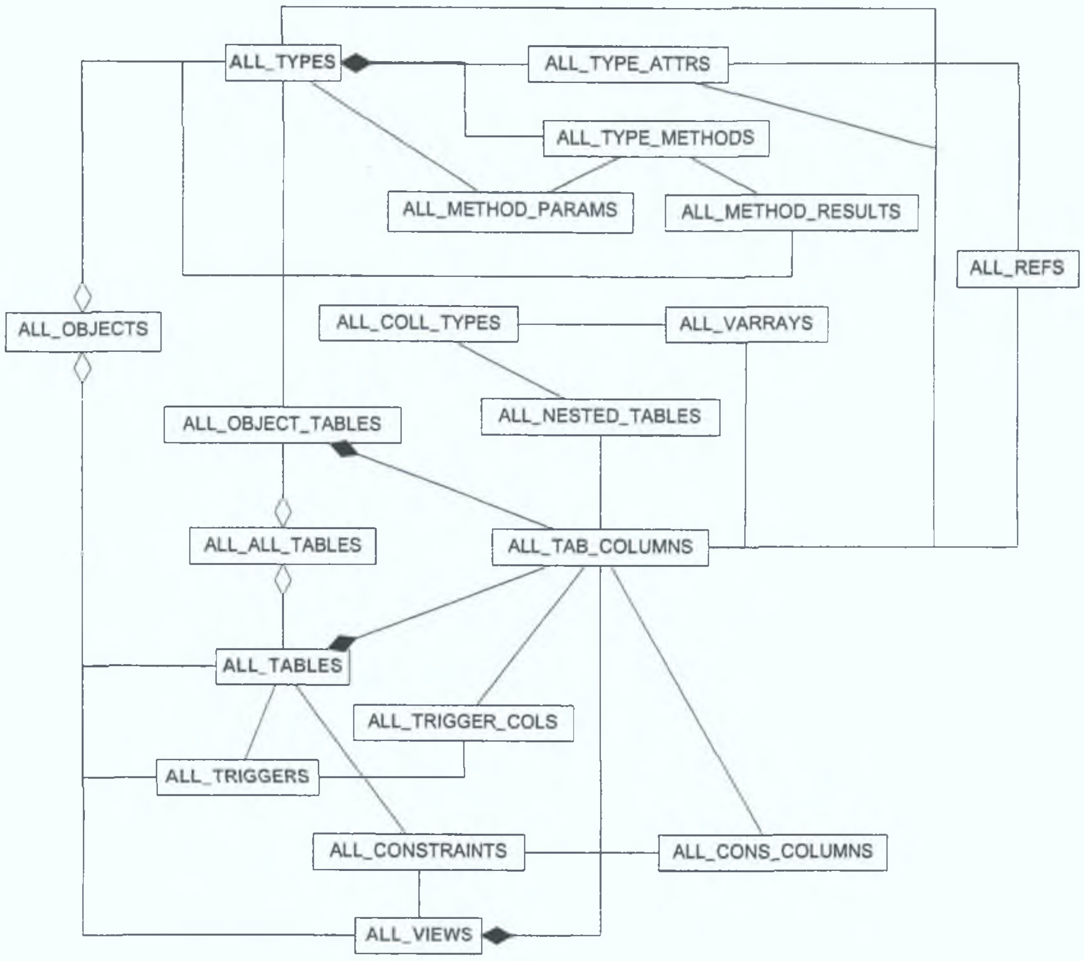


Figure 3.1: Object-Relational Metamodel Overview.

3.1.1 Object-Relational Types

A fully structured O-R model must present some cornerstone characteristics, including base type extension, inheritance and complex objects [SM96]. The O-R metamodel represents those characteristics as object extensions, which is discussed as following [Ora01]:

- In an O-R model, users are allowed to specify an user-defined data type (UDT), according to the required built-in datatypes. This feature makes it easier for developers to work with complex data such as image, audio and video. An UDT stores structured data in its natural form and allow applications to retrieve it in that form. An instance of an UDT is an object, which is identified by a unique object identifier (OID). Objects in O-R model are not isolated, they link each other through association, inheritance and aggregation/composition.

- UDTs are organised into a type hierarchy, where the structure and behaviour of several UDTs can be abstracted into a common supertype. A single inheritance model is supported: the subtype can be derived from only one parent type. It inherits all the attributes and methods of its direct supertype. A subtype can add new attributes and methods, and may override any of the inherited methods. Furthermore, a subtype can itself be refined by defining another subtype which derives from it, thus building up type hierarchies.

- A rich collection of complex objects are supported using collection types: *varray* and *nested table*. A varray is an ordered collection of elements and stored as opaque object like RAW or

`CLOB`. A nested table is an unordered set of data elements, all of the same datatype. It is a natural way to implement aggregation or composition, which is not specified in the SQL:1999 specification. Collection types whose elements are themselves directly or indirectly another collection type, build up multi-level collection types. Both single-level collection types and multi-level collection types can be used with columns in a table or with object attributes in object tables.

Figure 3.2 provides an overview of major types-metadata contained in the O-R metamodel. In this section, we also discuss how the object extensions of O-R model are represented in the metamodel. For a full description of how to access the O-R metadata, please refer to [O'C02].

- **ALL_TYPES.** `ALL_TYPES` is used to represent all the UDTs defined at the database schema. An UDT is an abstraction of a real-world entity and has three components: name, attributes (viewed from `ALL_TYPE_ATTRS`) and methods (viewed from `ALL_TYPE_METHODS`). It is a *template*, whereas an instantiated type is called an *object*. An UDT provides only the structure, and the extents are stored in object tables (viewed from `ALL_OBJECT_TABLES`) for the purpose of manipulation. An object can be retrieved into an object view (viewed from `ALL_VIEWS`) according to users' requirements. An object view (or typed view) is regarded as a virtual object table, where each row in the view is an object. A column object (viewed from `ALL_TAB_COLUMNS`) is used to describe an UDT occupied table column.
- **ALL_TYPE_ATTRS.** The attributes of an UDT model represent the structure and state of the real-world entity. Attributes are either built-in types such as `varchar2`, `integer`, `BLOB` or other UDTs (viewed from `ALL_TYPES`). Hence, there are dual relationships between `ALL_TYPE_ATTRS` and `ALL_UDTS`: an UDT is a composition of attributes and methods; an attribute also associates with an UDT because of its data type; an attribute can be a REF type (viewed from `ALL_REFS`), which represents the association between two UDTs.
- **ALL_TYPE_METHODS.** The methods of an UDT are functions or procedures written in PL/SQL or Java and stored in the database, or written in a language and stored externally. Methods implement operations that the application performs on the real-world entity. A method is allowed to take some arguments as parameters (viewed from `ALL_METHOD_PARAMS`) and may return results (viewed from `ALL_METHOD_RESULTS`) if it is defined as a function. They fall into three categories: member method, static method and comparison method.
- **ALL_METHOD_PARAMS.** Each method of an UDT is allowed to have zero or many parameters. The datatype of a parameter is either a built-in type or more complex, an UDT (viewed from `ALL_TYPES`).
- **ALL_METHOD_RESULTS.** The difference between a function and a procedure is that function returns values but procedure does not. If users specify the method as a function, the valuable results will be returned to the system. The result of a method may associate with `ALL_TYPES` also because its datatype is allowed to be either a built-in type or a complex UDT.
- **ALL_COLL_TYPES.** A collection type is another form of UDT, as it represents a collection of complex objects. A collection type describes a data unit made up of an indefinite number of elements, all of the same data type. The collection types include array types (viewed from `ALL_VARRAYS`) and table types (viewed from `ALL_NESTED_TABLES`). `ALL_COLL_TYPES` provides the abstract structure of collection types.

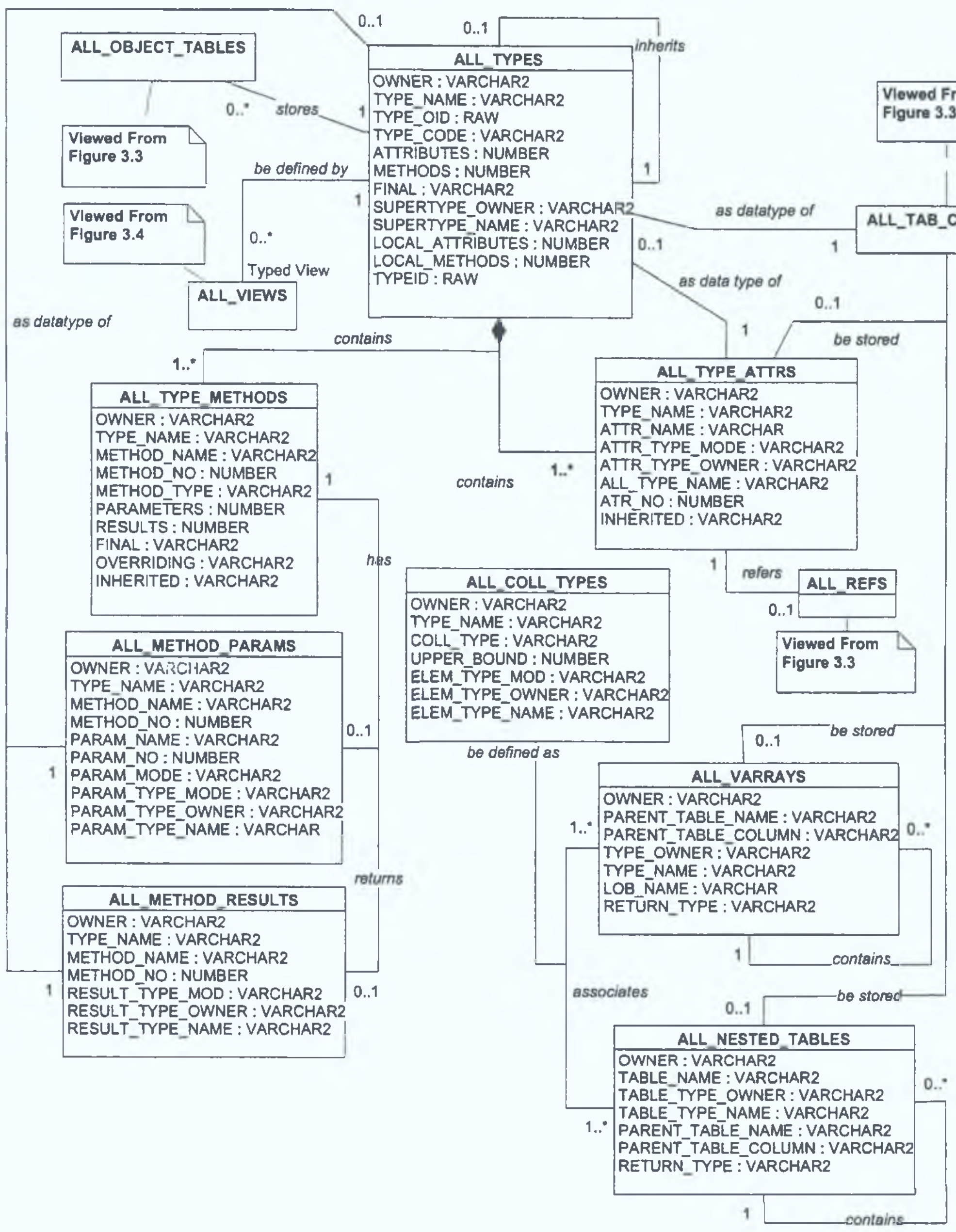


Figure 3.2: Object-Relational Types Metadata.

- **ALL_VARRAYS.** A varray is an ordered set of data elements. Users must specify the maximum number of elements while defining a varray. If it is sufficiently large, the built-in type BLOB can be used to store such varray. A varray can be used as the datatype of a column of table (viewed from ALL_TAB_COLUMNS). If a varray is involved in a multi-level collection type, it is possible that the varray associates with other varrays or nested tables (viewed from ALL_NESTED_TABLES).

- **ALL_NESTED_TABLES.** A nested table is an unordered set of elements, all of the same data type. It has a single column, and the type of that column is either a built-in type or an UDT. If the column in a nested table is an UDT, the table can also be viewed as a multi-column table, with a column for each attribute of the UDT. A nested table may associates with other nested tables or varrays (viewed from ALL_VARRAYS) and build up a multi-level collection type.

3.1.2 Object-Relational Tables

As the basic unit of data storage in the O-R model, tables hold all of the user-accessible data. A definition of a table includes: table name, column name, column datatype, column width or scale and precision (if datatype is NUMBER). Integrity constraints and triggers can also be defined for a table. Object table is a special kind of table that holds objects and provides a relational view of the attributes of those objects. Objects that appear in object tables are called *row objects*; and objects appear in table columns or as attributes of other objects are called *column objects*. In this section, an overview of metadata for O-R tables (*figure* 3.3) is represented. We also discuss how the object extensions are embedded the relational base.

- **ALL_ALL_TABLES.** ALL_ALL_TABLES contains all relational tables that store the relational format of data, and all object tables that store user-defined objects in the schema database. It is an aggregation between ALL_TABLES and ALL_OBJECT_TABLES.

- **ALL_TABLES.** All relational tables can be viewed at ALL_TABLES. A table is composed of at least one table column (viewed from ALL_TAB_COLUMNS), and specified by some constraints (viewed from ALL_CONSTRAINTS). A trigger (viewed from ALL_TRIGGERS) usually associates with tables. As an important mechanism in the O-R model, relational views (viewed from ALL_VIEWS) are created on the base of tables.

- **ALL_OBJECT_TABLES.** As stated previously, an UDT only defines the structure of entity. As the O-R database uses object table to hold objects, it provides a tabular view of an UDT (viewed from ALL_TYPES). An object table is also composed of table columns (viewed from ALL_TAB_COLUMNS). The pre-requisite to object table definition is that the base UDT must be defined and stored in the schema. An object table can be viewed in two ways: a single column table in which each row is an object, perform O-O operations or a multi-column table in which each attribute of the UDT occupies a column, perform relational operations [Ora01]. The first representation allows objects to be accessed through an O-O application; and the latter representation allows objects to meet the relational format, which are accessed as relational data. In an object table, objects that occupy complete rows are considered as row objects; object that occupy table columns in a larger row, or are attributes of other objects, are viewed as column objects. Object tables are the overlap between the O-O concept and relational data model. The real-world entity is defined as UDT which is similar to a class in the O-O model, and stored in a tabular format which the relational model recognises.

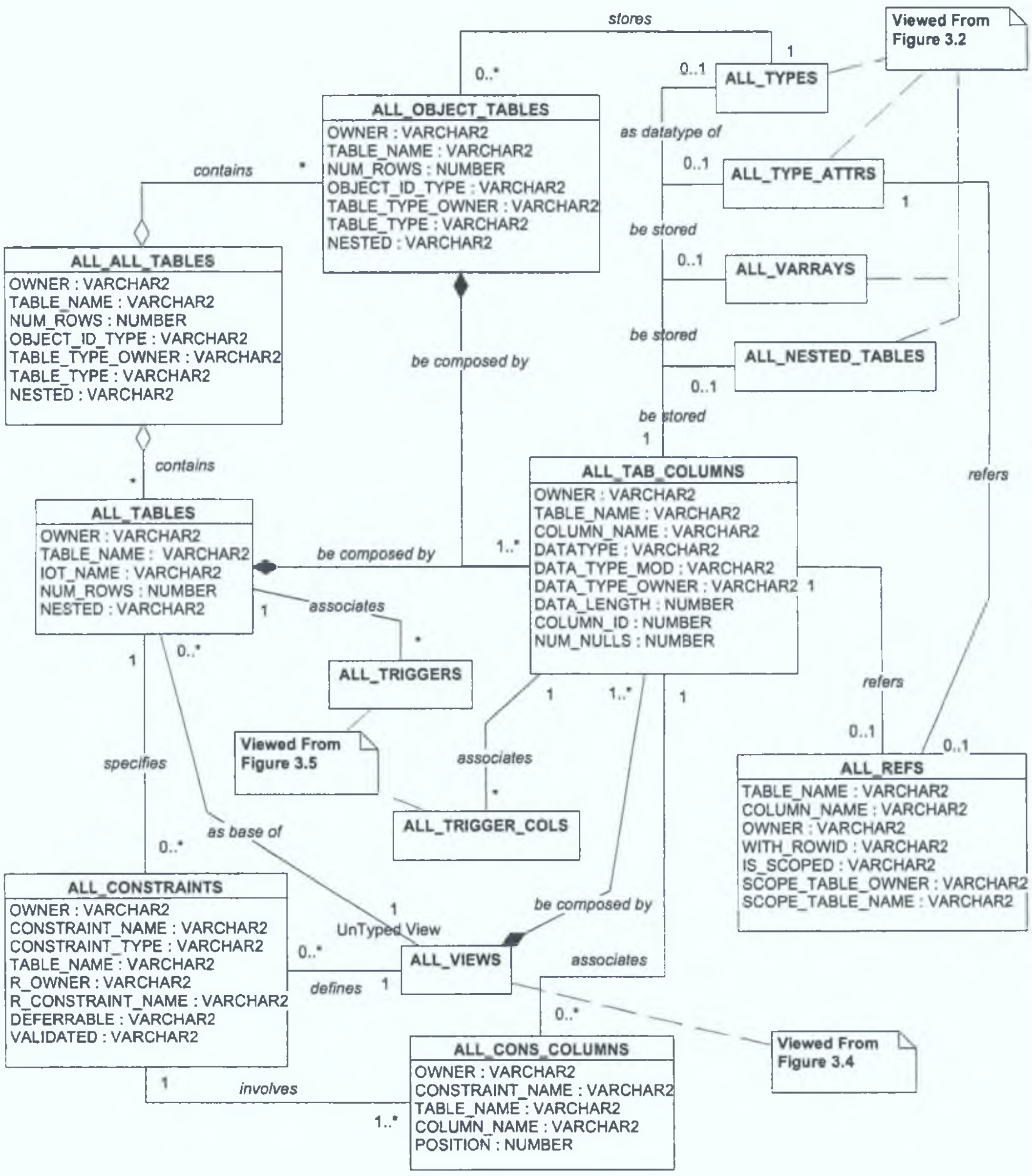


Figure 3.3: Object-Relational Tables Metadata.

- **ALL_TAB_COLUMNS.** Both relational tables and object tables are composed of columns. Each table requires at least one column. The datatype of each column is allowed to be a built-in type or an UDT (viewed from `ALL_TYPES`) or a collection type (viewed from `ALL_VARRAYS` or `ALL_NESTED_TABLES`). If an object table is viewed as a multi-column table, then each table column stores the corresponding attribute of the UDT (viewed from `ALL_TYPE_ATTRS`). It is possible that a table column is a `REF` datatype (viewed from `ALL_REFS`) for the purpose of referencing another object. A column is allowed to associate with constraints (viewed from `ALL_CONS_COLUMNS`) and triggers (viewed from `ALL_TRIGGER_COLS`).

- **ALL_REFS.** A `REF` is a logical pointer to a row object. It is specified while defining an UDT. The reason for discussing `REF` in O-R tables section, rather than O-R types section is that it is the link between two objects, not object structures. In other words, a `REF` is represented as an object table column (viewed from `ALL_CONS_COLUMNS`). `REF`s model the associations between objects, especially many-to-one relationships in order to reduce the need for foreign keys. An easy navigation between objects is provided by this mechanism. `ALL_REFS` stores the information of object table columns that references to other UDTs.

- **ALL_CONSTRAINTS.** The O-R database uses integrity constraints to prevent invalid data entry into tables. Users are allowed to define integrity constraints to enforce the business rules which must be associated with the information in the database. An integrity constraint is defined for tables (viewed from `ALL_TABLES`) and stored in the schema repository. It can also be applied to a view (viewed from `ALL_VIEWS`). Each integrity constraints of a table or a view includes many constraint columns (viewed from `ALL_CONS_COLUMNS`), which clarify the individual constraint.

- **ALL_CONS_COLUMNS.** While specifying a table column, an integrity constraint is associated according to users requirements. For tables, an integrity constraint (viewed from `ALL_CONSTRAINTS`) imposes rules only on the column (viewed from `ALL_TAB_COLUMNS`) in which it is defined.

3.1.3 Object-Relational Views

A view mechanism is an important feature of the O-R model. Like a table, a view is composed of columns (viewed from `ALL_TAB_COLUMNS`); unlike a table, a view does not allocate any storage space, nor does a view actually contain data or objects. Rather, a view is defined by query extracts and derives data or objects from tables that the view references. In this section we focus on discussing object views, which is the conjunction between relational data and object based applications. *Figure 3.4* provides an overview of metadata included in this section. The relational view is defined as *untyped view* and the object view is named as *typed view* in the O-R model. We discuss them separately although they are represented by one class `ALL_VIEWS`.

- **Untyped View.** An untyped view presents the relational view and can be thought of as a virtual table. Users are allowed to use it in most places where base tables (viewed from `ALL_TABLES`) can be used, also may query it with some restrictions (viewed from `ALL_CONSTRAINTS`) as the subsets of table constraints. While specifying the restrictions on a table, users cannot define a constraint on a column whose datatype is an UDT. Views also associate with triggers with some restriction (viewed from `ALL_TRIGGERS`).

- **Typed View.** A typed view is an extension of the relational view mechanism. It is useful in prototyping or transitioning to O-O applications because the data in the view can be

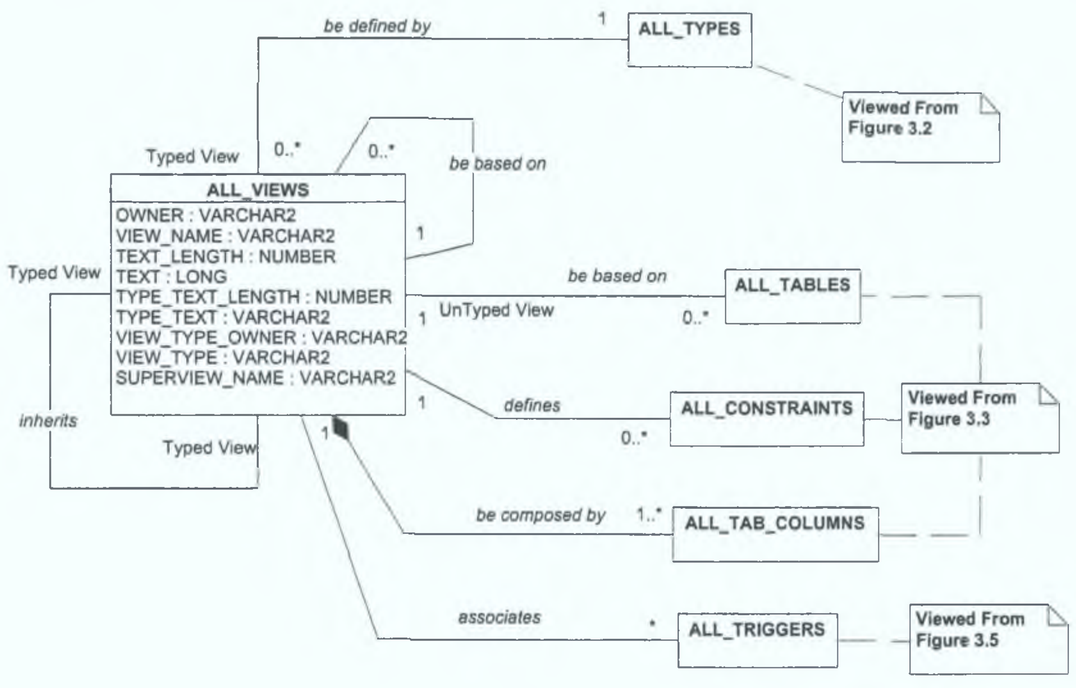


Figure 3.4: Object-Relational Views Metadata.

taken from relational tables and accessed as if the table were defined as an object table [Ora01]. A typed view is also a virtual representation of UDT (viewed from `ALL_TYPES`). It is thought of as a virtual object table. The extent of typed view is generated with selecting objects from the base UDT extent stored at object tables. While specifying a typed view, it is the responsibility of users to define OID as a combination of columns in the defining select(typically this will be a combination of primary keys used in the query).

3.1.4 Object-Relational Triggers

Triggers (*figure* 3.5) are user-defined procedures that execute implicitly when an `INSERT`, `UPDATE`, or `DELETE` statement is issued against the associated table (viewed from `ALL_TABLES`), against a view (viewed from `ALL_VIEWS`), or when database system actions occur [Ora01]. Triggers are similar to stored procedures. However, procedures and triggers differ in the way that they are invoked. A procedure is explicitly executed by a user, application, or trigger. Triggers are implicitly fired by the system when a triggering event occurs, no matter which user is connected or which application is being used.

Although, users cannot explicitly define triggers on views, they can be defined for the underlying based tables referenced by the view. The O-R database supports the definition of logical constraints on views. User can write normal `INSERT`, `UPDATE`, and `DELETE` statements against the views. `INSTEAD OF` triggers are activated for each row of the view that gets modified.

- **ALL_TRIGGERS.** `ALL_TRIGGERS` is used to describe all the triggers have been defined at the database schema. Its basic parts, such as event or statement, restriction and action are represented by the attributes.
- **ALL_TRIGGER_COLS.** `ALL_TRIGGER_COLS` is used to describe the use of columns in triggers or in triggers on tables, which is associated with table columns.

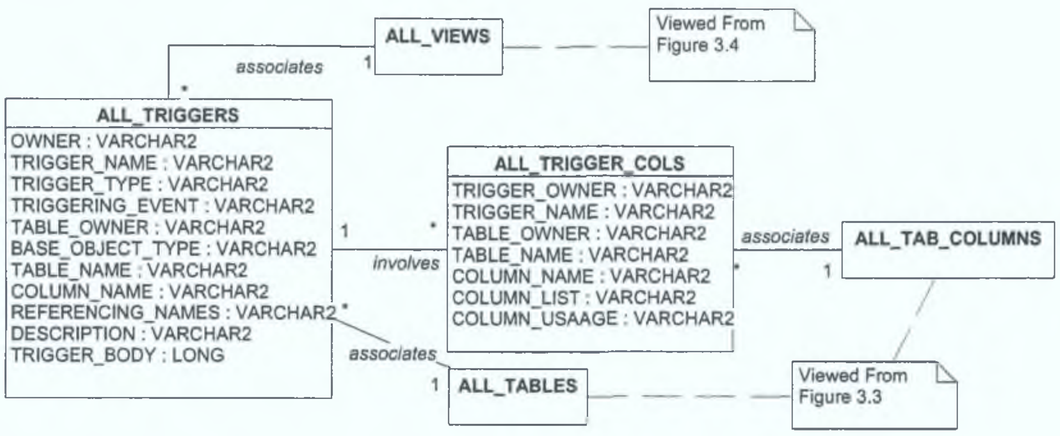


Figure 3.5: Object-Relational Triggers Metadata.

3.1.5 Summary

The O-R metamodel provides a better understanding of O-R data model. It specifies interdependencies among concepts used to build an O-R schema, some inherent constraints, and abstract syntax of corresponding data description statements. As stated previously, the metamodel supports the object extensions of the O-R model, such as object types, collection types and type hierarchy. Furthermore, object tables and object views, the intersection between O-O model and relational model are supported also. Although the metadata of a role-based view system is not provided in the metamodel, it offers us the opportunity to develop different forms of view mechanism. The next step is to extend the metamodel with the metadata constructs needed to specify a role-based view mechanism.

3.2 Extending the O-R Metamodel

In this section, we provide extensions to the O-R metamodel. A metamodel of our role-based view system is presented first; then we present an extended O-R metamodel; and finally, the implementation is briefly introduced. Note that only the major issues about the metamodel extensions are covered due to the thesis length limit. For a full description of how to extend the O-R metamodel, please refers to [Wan02b].

3.2.1 Role-Based View Metamodel

The purpose of this section is to present the metamodel of the role-based view system and provide a better understanding of our view model. An UML class diagram tool is used to present the metamodel in an O-O perspective (*figure 3.6*). There are five metaclasses existing in our role-based view metamodel: `Roleview` class is composed of `Root` and `Role` classes; each `Root` class or `Role` class is composed of `Attribute` and `Method` classes. The details of each metaclass and the relationships between them are now presented:

- **Roleview Class.** This is the container for the root and role classes. An aggregation relationship exists between the `Roleview` and `Role` class. A `Role` is part of `Roleview` but it is possible that a `Roleview` exists without the existence of `Role` class. However, a `Roleview` does not exist if the `Root` class is not included. Both the `Root` class and `Role` class are

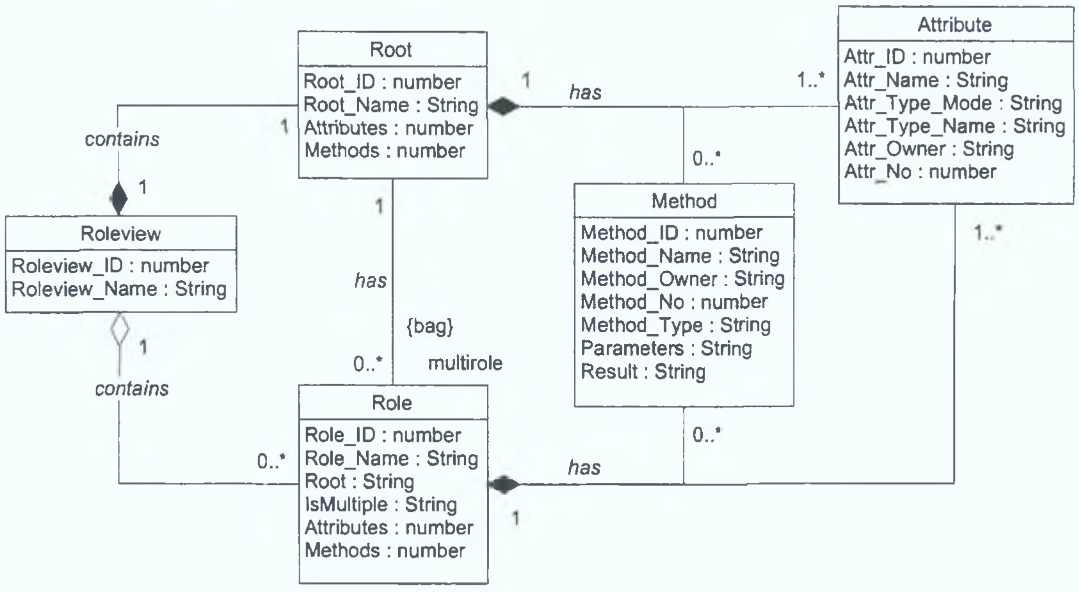


Figure 3.6: Roleview Metamodel.

composed of `Attribute` and `Method` classes, which are discussed later. A `Roleview` instance is identified by a system-generated OID represented by attribute `Roleview_ID`.

- **Root Class.** A `Root` class is composed of `Attribute` and `Method` class. There is a one-to-many relationship between `Root` class and `Role` class. A `Root` instance may associate with many `Role` instances; but a `Role` instance only associates with one single `Root` instance. In this association, if the instance of `Role` class is a `multirole`, it allows multiple occurrences of the same `Role` instance to occur. The keyword `bag` represents this situation. A `Root` instance is identified by a system-generated `Root_ID`.

- **Role Class.** This is the class that represents the structure of all defined *role* objects. A `Role` class is composed of both `Attribute` and `Method` class. The attribute `IsMultiple` represents the type of `Role` instance, which is either a `multirole` or a normal (single) role. Each `Role` instance is identified by an unique identifier `Role_ID`.

- **Attribute Class.** This class is used to represent the properties of `Root` or `Role` classes. The attribute `Attr_Owner` specifies that it associates with a `Root` or `Role` class. The `Attribute` class presents the state of root or role class as the `ALL_ATTRIBUTES` metaclass presents the state of UDT in the O-R metamodel. An `ATTRIBUTE` instance is identified by a system-generated `Attr_ID`.

- **Method Class.** This class is used to represent the functions or procedures in which a `Root` or `Role` instance performs behaviour. The attribute `Method_Owner` is used to specify that it associates with a `Root` or `Role` class. The `Method` class presents the behaviour of root or role class as the `ALL_METHODS` metaclass presents the UDTs behaviour in the O-R metamodel. A `Method` instance is allowed to take some parameters and may return results as well. It is also identified by a system-generated `Method_ID`.

3.2.2 Metamodel Extensions

While extending the current O-R metamodel with the additional metaclasses, it was important to specify the relationships between the new classes and the original metaclasses. *Figure 3.7* presents

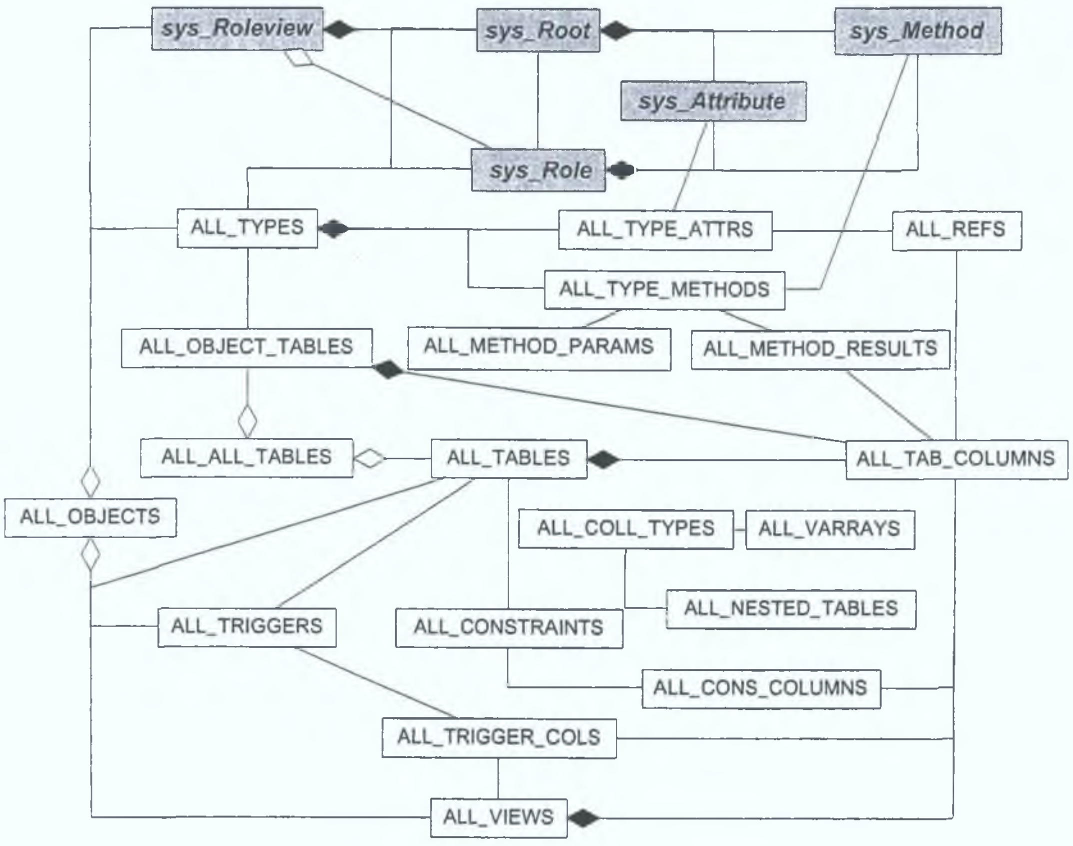


Figure 3.7: Object-Relational Metamodel & Extensions.

an overview of the extended metamodel. The extension metaclasses have a prefix '`sys_`' for clarity. The relationships between the role-based view metaclasses and the O-R metaclasses are clarified.

Figure 3.8 presents the role-based views metadata section from an implementational perspective. The datatype of each attributes is now converted to SQL:1999 datatypes and the role-based view metaclasses are linked to the O-R metaclasses represented at other sections. Each metaclass of the role-based view metamodel is discussed separately.

- **Sys_Roleview.** The `sys_Roleview` class is a logical structure that refers all objects in a role-based view. Since `ALL_OBJECTS` is defined as the logical structure of schema objects, `sys_Roleview` is viewed as the extra content of `ALL_OBJECTS`. Each `sys_Roleview` is identified by a system generated `Roleview_ID`, as with any other schema object in the metamodel. As *figure 3.7* shows `ALL_OBJECTS` is now the aggregation of `sys_Roleview`, `ALL_TYPES`, `ALL_TABLES`, `ALL_VIEWS` and `ALL_TRIGGERS`. A root is the static part of a role-based view, hence it is necessary to specify a pointer to that root. The attribute `Root` represents the reference.

- **Sys_Root.** A `sys_Root` can be viewed as a virtual representation of an existing UDT with the intrinsic properties. In the O-R schema repository, UDTs can be seen in the `ALL_TYPES` view. Hence, a specified `sys_Root` must be based on an underlying UDT. However, it is possible that an existing UDT is not associated with any `sys_Root` class. It is supported that one single UDT is specified in several `sys_Root` classes, which are associated with different `sys_Roleview` classes, according to different requirements. For example, if an UDT `Person` is specified in a root class in one roleview `Professional_View`; it can also appear in the root class of roleview `Academic_View`. In `sys_Root` metaclass, the attribute `Type_Ref`

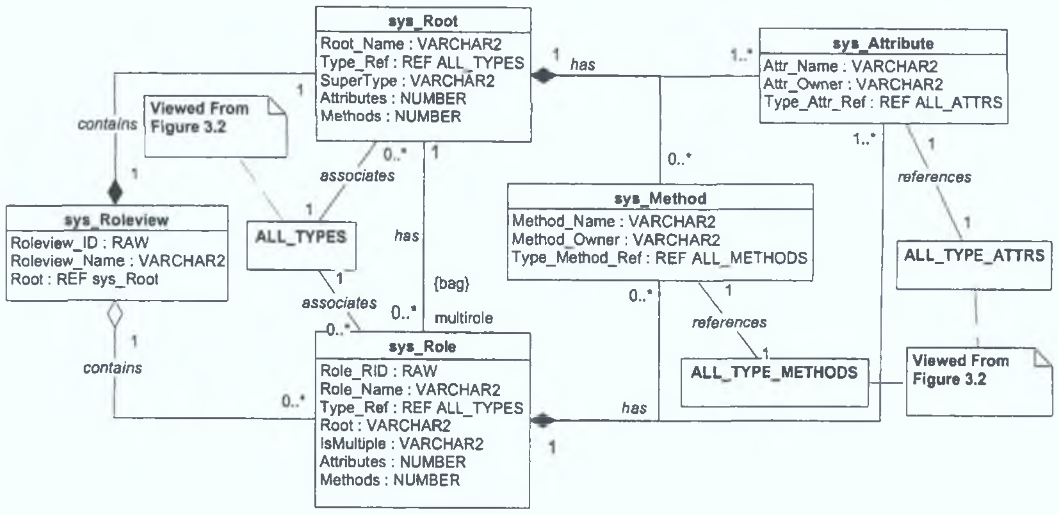


Figure 3.8: Object-Relational Roleviews Metadata.

is a REF datatype, which references the base UDT. Because each type is organised into type hierarchy in the O-R model, there is a possibility that a `sys_Root` based UDT is the subtype in a type hierarchy. Hence, the attribute `SuperType` is used to store the name of supertype or *null* value if the base type has no supertype. Compared with the `Root` class represented in *figure 3.6,* `sys_Root` metaclass loses the attribute `Root_ID`. The reason we remove this attribute is that a `sys_Root` is the virtual representation of an existing UDT and does not contain any objects. The object preserving semantics is used to maintain the same unique identifier.

- **Sys_Role.** The metaclass `sys_Role` is a virtual representation of an existing UDT with the extrinsic properties. Like `sys_Root`, a `sys_Role` is also based on an existing UDT and tightly depends on its existence. If the UDT that `sys_Role` based does not exist, the specification of `sys_Role` has no use. On the other hand, it is possible that an existing UDT is not associated with any `sys_Role`. One single UDT is allowed to be specified in several `sys_Role` classes of different `sys_Roleview` classes. For example, UDT `Student` may be specified in a role class of roleview `Professional_View` and `Academic_View`. The attribute `Type_Ref` is used to reference the base UDT. A `sys_Role` object is identified by the object preserving semantics along with a system-generated transient OID `Role_RID`.

- **Sys_Attribute.** A `sys_Attribute` class represents the property of the `sys_Root` or `sys_Role` class. Classes `sys_Root` and `sys_Role` are virtual representations of existing UDTs, hence, there is no physical data existing in `sys_Attribute` class. We specify a mappings between `sys_Attribute` and `ALL_TYPE_ATTRS`, where the physical data can be found.

- **Sys_Method.** A `sys_Method` represents the function or procedure the `sys_Root` or `sys_Role` class contains. Since the methods of an UDT have been stored in the schema repository, the `sys_Method` needs only to reference the stored method rather than define a new method. There is also a mappings specified between `sys_Method` and `ALL_TYPE_METHODS`.

3.2.3 Implementing the Metamodel Extensions

In reality, the role-based view metaclasses must be defined as new meta-UDTs and stored in the schema repository along with other existing meta-UDTs. Object references are used to express the

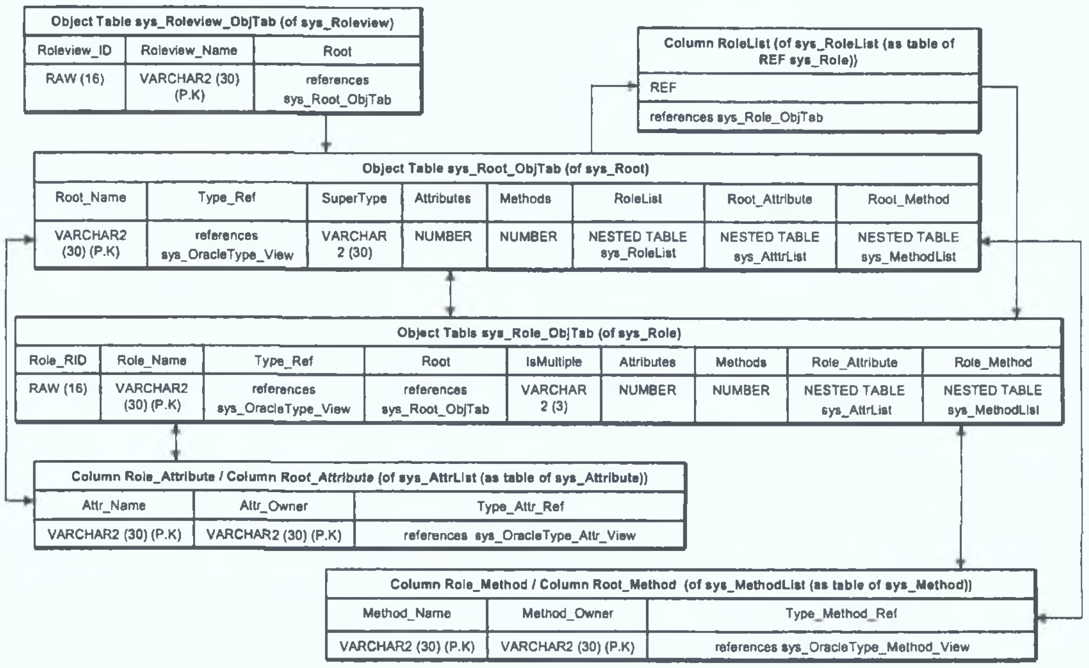

Object Table sys_Roleview_ObjTab (of sys_Roleview)
Roleview_ID
Roleview_Name
Root
RAW (16)
VARCHAR2 (30) (P.K)
references sys_Root_ObjTab
Column RoleList (of sys_RoleList (as table of REF sys_Role))
REF
references sys_Role_ObjTab
Object Table sys_Root_ObjTab (of sys_Root)
Root_Name
Type_Ref
SuperType
Attributes
Methods
RoleList
Root_Attribute
Root_Method
VARCHAR2 (30) (P.K)
references sys_OracleType_View
VARCHAR 2 (30)
NUMBER
NUMBER
NESTED TABLE sys_RoleList
NESTED TABLE sys_AttrList
NESTED TABLE sys_MethodList
Object Table sys_Role_ObjTab (of sys_Role)
Role_RID
Role_Name
Type_Ref
Root
IsMultiple
Attributes
Methods
Role_Attribute
Role_Method
RAW (16)
VARCHAR2 (30) (P.K)
references sys_OracleType_View
references sys_Root_ObjTab
VARCHAR 2 (3)
NUMBER
NUMBER
NESTED TABLE sys_AttrList
NESTED TABLE sys_MethodList
Column Role_Attribute / Column Root_Attribute (of sys_AttrList (as table of sys_Attribute))
Attr_Name
Attr_Owner
Type_Attr_Ref
VARCHAR2 (30) (P.K)
VARCHAR2 (30) (P.K)
references sys_OracleType_Attr_View
Column Role_Method / Column Root_Method (of sys_MethodList (as table of sys_Method))
Method_Name
Method_Owner
Type_Method_Ref
VARCHAR2 (30) (P.K)
VARCHAR2 (30) (P.K)
references sys_OracleType_Method_View

Figure 3.9: Object-Relational Presentation of Roleview Metamodel.

relationships between them; and collection types model the multi-value attributes. Furthermore, an O-R perspective is presented in order to store and manipulate the meta-objects. The standard data definition language is used to define the meta-UDTs. There are two steps to complete the implementation of metamodel extensions, with a fully described prototype provided in *Appendix A*.

1. The O-R Meta-UDTs Definitions. The O-R model stores metadata in the schema repository as static tables and views. The base tables store information about the associated database and users are not allowed to access these tables. It is necessary for our approach to place a number of 'virtual' UDTs that represent the structure of the O-R metamodel. Hence, `sys_OracleType` is specified to represent the existing UDT structure, `sys_OracleType_Attr` presents the structure of UDT attributes and `sys_OracleType_Method` presents the UDT methods structure. Furthermore, we must specify the corresponding meta-object tables based on the existing O-R meta-views in order to query and manipulate the meta-objects. As the result, the metadata presented in the O-R meta-views can be accessed and referenced through the meta-tables we defined.

2. The Role-Based View Meta-UDTs Definitions. We now define the role-based view meta-UDTs which represent the metamodel extensions. There are five new meta-UDTs: `sys_Roleview`, `sys_Root`, `sys_Role`, `sys_Attribute` and `sys_Method`. It is very important to specify a clear mappings between these new meta-UDTs and the virtual UDTs defined previously. For example, the attribute `Type-Ref` of `sys_Root` references `sys_OracleType` and attribute `Type_Attr_Ref` of `sys_Attribute` references `sys_OracleType_Attr`. Once the new meta-UDTs are defined completely, it is necessary to specify the corresponding meta-object tables which store and manipulate the roleview meta-objects. *Figure 3.9* illustrates the role-based view meta-UDTs in an O-R perspective. A full description of the metamodel implementation is discussed in chapter 5.

3.3 Conclusions

One of the functions of this thesis is to examine the latest version of O-R metamodel. The O-R model is complex and the examination of its metamodel provides a better understanding of how it works. An O-R metamodel contains the meta objects stored in the schema repository. It is divided into four sections: O-R types, O-R tables, O-R views and O-R triggers. Extending the relational model with object extensions is one of the characteristics of the O-R model. It is represented at the O-R types section by providing some new features, such as allowing user to define new object types; using collection types to present rich object collection; and group object types into type hierarchy. In the O-R metamodel, the structure of UDTs are specified at the O-R types section, while object storage is represented at the O-R tables section. Object table, a new object type, is defined to store the O-O concept based UDTs in a tabular format which the relational model recognises. It is the overlap between the O-O concept and the relational model. Object tables provide a relational perspective for the O-O based UDTs, and on the other hand, O-R views provide an object perspective for the relational data. In the O-R views section, object views present the functionality that abstracts the relational data in an O-O format. The O-R triggers section present the object extensions of the relational triggers.

The major contribution of this thesis is to define a role-based view mechanism in the O-R model. However, the current O-R metamodel does not support role-based view metadata. Hence, it is necessary to extend the metamodel with extra metaclasses in order to model the new object types. The extended O-R metamodel is also introduced in this chapter. There are two steps to complete the extensions: first is to discuss the role-based view metamodel; second is to specify the relationships between the existing O-R metaclasses and the new metaclasses. While introducing the role-based view metamodel, an UML design tool is used to represent the structure and each individual metaclass is discussed. An extension is provided by clearly specify the associations between existing metaclasses and new defined metaclasses. Once the concept and semantics are discussed, we offer a brief introduction of implementing the extended metamodel.

Chapter 4

Designing Role-Based Views for O-R Databases

In this chapter a role-based view mechanism is described. Throughout the rest of this chapter and in subsequent chapters, it is assumed that 'virtual schema', 'view schema', 'roleview' and 'subschema' have the same meaning, i.e. they refer to a role-based view, and each term may be used while describing how a role-based view is created and stored in O-R databases.

In §4.1 a general introduction of a roleview is provided. A roleview is a subschema as it includes multiple virtual classes. The specification of virtual classes are also introduced. In §4.2 the semantics and syntax of roleview are presented, along with the generation of roleview extents. In §4.3, we introduce a number of restructuring operators which are used to derive new virtual classes or manipulate the virtual class objects. Finally, some conclusions are offered in §4.4.

4.1 Introduction

The view mechanism in a FDBS must retain as much semantic information as possible in order to meet the original criteria for canonical model selection [SCGS91]. The perception of a view is that of a subschema, possibly containing multiple virtual classes. In our role-based view system, the root and role classes are virtual classes, composed of existing UDTs. A virtual class is automatically constructed and stored in the schema repository when roleview is processed. An existing UDT may be used by multiple virtual classes in multiple roleviews. In an O-R database, a single schema comprises multiple UDTs, while the extended model includes multiple roleviews and virtual classes. The relationships between the database schema, base UDT, view schema, root class and role class are illustrated in *table 4.1*.

While specifying virtual classes of a roleview, there are two allowable restructuring options: *projection* and *selection* [Wan03]. The projection option is used to select certain attributes from the base UDT as the virtual attributes while discarding other UDT attributes. It is expressed by the `select` clause. In the O-R model, an UDT is used to present the structure of an entity and an extent is a collection objects of the same UDT. These objects are stored in an object table for the purpose of query or manipulation. The selection is used to select objects from the base UDT extent to form the extent of virtual classes. This operation consists of selecting rows of object tables which satisfies certain conditions, and it is expressed by the `where` clause.

Entity A	Entity B	Relationship	Description
DB Schema	Base UDT	1 – n	A database schema comprises many UDTs.
DB Schema	Roleview	1 – n	A roleview is a subset of the base schema, many are defined using one database schema.
DB Schema	Virtual Class (Root Class & Role Class)	1 – n	A database schema comprises multiple virtual classes although never it directly references them.
UDT	Virtual Class (Root Class & Role Class)	m – n	An UDT may be redefined in multiple virtual classes; and more than one UDTs are used to form a single virtual class.
Roleview	Base UDT	m – n	An UDT may participate in multiple roleviews, and any roleview may associate with more than one UDT.
Roleview	Root Class	1 – 1	Each roleview only contain one single root class. A root class only belongs to one roleview.
Roleview	Role Class	1 – n	A roleview may contains multiple role classes. Each role class only belong to one roleview.

Table 4.1: Scope Issues.

4.2 Defining Role-Based Views

Traditionally, a view definition is a data definition statement which incorporates a data manipulation command. For example, a view structure is defined, a query is used to specify the structure and to generate the extents of the virtual classes, and the entire view definition is stored in the database. This standard is maintained by providing our role-based view declaration, but is extended to allow multiple class redefinitions, in order to construct a subschema of multiple virtual classes. A form of extended SQL:1999 is needed where the new object type `roleview` is added to the list of existing object types such as `type` and `table`. The keyword `create roleview` indicates that virtual schema and its components must be constructed and stored in the schema repository. Before defining the role-based view in an O-R database, it should be ensured that the O-R metamodel has been extended with role-based view metaclasses. The proposed roleview mechanism is based on clear semantics. The O-R model uses UDTs to describe the real-world entities, and this concept is extended by using virtual classes to describe the virtual entities. There are three steps to design a roleview specification language: understanding the semantics; the syntax to define the roleviews; and the generation of the extents.

4.2.1 Role-Based View Semantics

A roleview is a named collection of virtual classes based on the defined UDTs in the database schema. Those UDTs may be connected either through inheritance or associations. The name of the roleview, the names of all virtual UDTs (root and role classes) and base UDTs must be explicit in the roleview definition. Furthermore, the structure of each virtual UDT must be clearly specified in the roleview definition. The semantics of roleviews are as follows:

- Each virtual class is based on an existing UDT, with virtual meta-types and meta-objects stored in the schema repository. It is users' responsibility to ensure the existence of base UDTs. The system will throw an error exception if the base UDTs cannot be found in the schema repository. In the O-R model, UDTs are organised into a type hierarchy, where the structure and behaviour of several UDTs can be abstracted into a common supertype. Hence, it is possible that a virtual class is based on an UDT which is a subtype in a type hierarchy. If that UDT in a root class, we assume that the root class also inherits the properties and behaviour from the supertype of its base UDT. If that UDT in a role class, the role class does not inherits the properties and behaviour from the supertype of its base UDT.

- A roleview contains multiple virtual classes. The root and role classes are virtual representations of existing UDTs. An object defines the permanent properties in a root object, while each role object defines some of its transient properties. In other words, a root and role class present different characteristics of a real-world entity.

- A roleview contains only one root class, while the number of role classes is infinite. The root class is a static component of the roleview while a role class is dynamic. The roleview is deleted if its root class is dropped from the database while it does not lose its identity if one of its role class is dropped.

- A root object may play multiple role objects or the same role object multiple times. A role object cannot exist without its root object. Deleting a root object implies deleting all of its role objects.

- An object preserving semantics is used to identify the root object. No new identifiers are constructed as a result of a new root object. Where a root class is derived from a base UDT, the base UDT identifier is used, and the onus is on the roleview mechanism to provide access to the root class. Every attribute and relationship property in the roleview schema is connected to the base UDT equivalent.

- The role object is identified by a compound identifier which comprises the object identifier of the base UDT and a system generated transient role object identifier. While defining root and role classes, it is possible that the base UDTs are either link to each other or not. In the first case, it is not necessary to use the object preserving semantics in a role object because its root object has the same object identifier as its base UDT, and this UDT links to the role base UDT through either inheritance or association relationship. In the second case, we must use the object preserving semantics to identify the role object, otherwise, there is no any references to specify the link between a role object and its base UDT object. For the purpose of simplicity and non-ambiguity, we use a compound identifier to identify the role objects. The object preserving semantics is used to mapping the role object to its base UDT object; and the object generating semantics is used to represent the dynamic feature of roles. The system only generate the transient identifier at the run time rather than compile time.

- Derived attributes and relationships must connect to existing attributes and relationships. The root and role classes are the virtual classes of the base UDT. There are no new attributes or relationships generated in the roleview definition.

4.2.2 Role-Based View Definition Syntax

We now provide a syntax for defining roleviews. There are two elements involved in defining a virtual schema: first it is necessary to define the structure of the newly defined schema; and second, it is necessary to specify queries which are used to select objects from the base schema, which generate an extent for each virtual class. In the first case, it is necessary to provide the names of base UDTs and the properties contained within those virtual classes. In the latter case, it is necessary to identify the queries and ensure there is no overlap between each extent. The `create roleview` declaration is used to define virtual schema, and in this chapter a full BNF grammar is provided for this extension to SQL:1999, and the practice of numbering production rules has been adopted to aid later explanations. The top level of format is illustrated in *definition 1*, and in *definition 2* and *definition 3* the expansions are provided. Note that a form of BNF used by the ANTLR [Ant03] parser library is employed, as this was used to construct the parser developer for this research. It is described in full in chapter 5 together with details of prototype and experiments. An earlier version of view definition syntax is presented in [WR03].

Definition 1 *Roleview Statement.*

"create" "roleview" roleview_name: Identifier "as"

root_dcl

*(role_dcl)**

;

In the statement shown in *definition 1*, there are two portions, and a strict ordering exists so that they must be specified according to their ordering in the definition. The root class must be specified first in the definition, and the role classes are specified afterwards. According to the definition, only one root class exists in the statement, while zero or more role declarations are allowed in the statement. The *'(rule)*'* format indicates this option. In *definition 2*, more details regarding `root` and `role` declarations are illustrated.

Definition 2 *Root & Role Statement.*

root_dcl:

"root"

qualifier_dcl

;

role_dcl:

"role | multirole"

qualifier_dcl

;

In this definition, `root_dcl` and `role_dcl` are illustrated, along with a sub-declaration `qualifier_dcl` which is used by both declarations. The root declaration uses the `root` keyword followed by a `qualified_dcl` declaration. The role declaration starts with the optional keyword `role` or `multirole`, followed by a `qualified_dcl` declaration. The keyword `multirole` indicates multiple occurrences, while the keyword `role` indicates the single occurrence.

Definition 3 *Production Rules.*

qualifier_dcl:

class_name: Identifier "of" type_name: Identifier "is"

sql_dcl

;

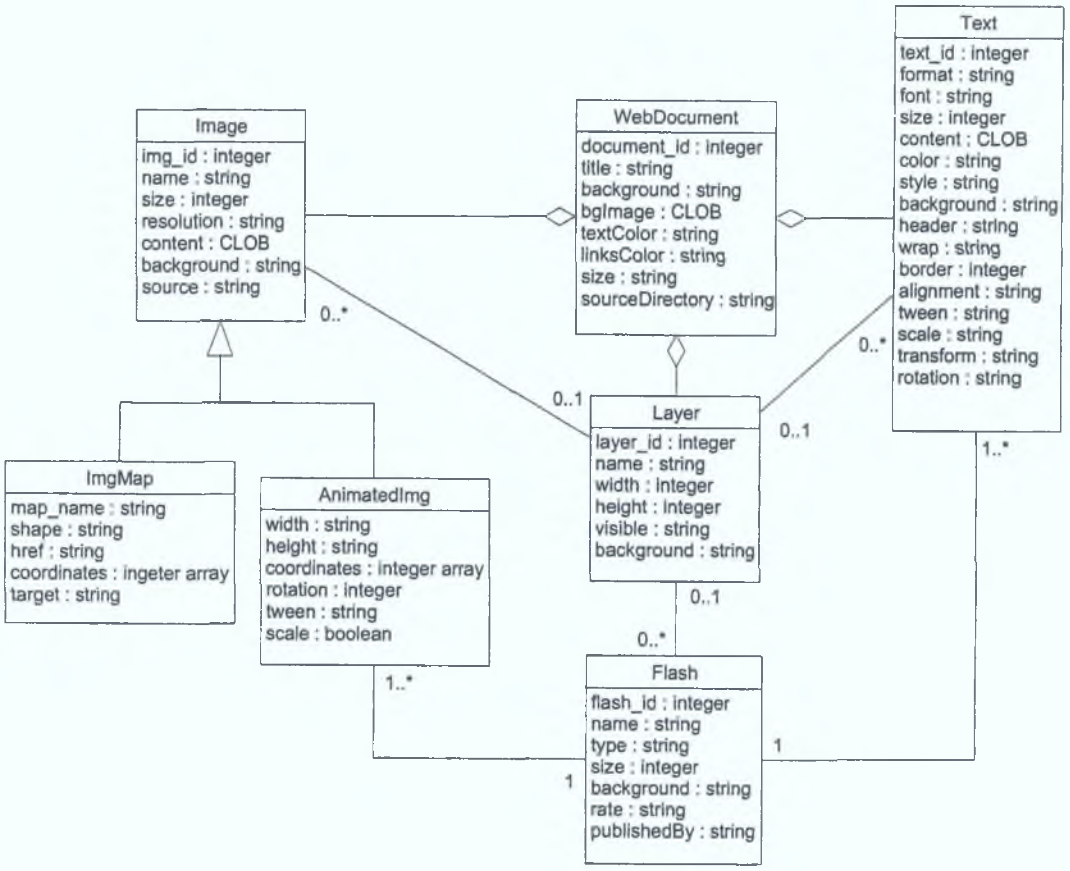


Figure 4.1: Web Design Schema.

In the statement of `qualifier_dcl`, an identifier is used to specify the name of root class or role class, followed by the keyword `of` and another identifier which indicates the name of existing UDT. This declaration is finished by the substatement `sql_dcl`. The `sql_dcl` expression performs the `select...from...where...` statement in SQL:1999. The `select...` substatement is used to restructuring the properties by projection option; the `from...` substatement is used to specify the underlying UDTs; the `where...` substatement deals with restricting the extent of UDTs by the selection condition.

4.2.3 Role-Based View Definition Examples

Throughout the remainder of the thesis, the pragmatics of roleviews are demonstrated through a number of examples. In this section, we provide the initial examples of how to define a roleview. In *figure 4.1*, a web design schema is presented and in *figure 4.2* two roleview schema are illustrated which are based on the web design schema. As *figure 4.1* shown, a `WebDocument` is composed of `Image`, `Text` and `Layer` UDTs. A `Layer` UDT associates with `Image` and `Text` UDTs. An `Image` UDT is the supertype of `ImgMap` and `AnimatedImg`. A `Layer` UDT also associates with UDT `Flash` and `AnimatedImg`. UDT `Image`, `ImgMap` and `AnimatedImg` are grouped into a type hierarchy and the subtypes inherits the properties and behaviour from its supertype. There are two subschema defined on the base of sample schema: `ImageView` (*figure 4.2(a)*) and `TextView` (*figure 4.2(b)*). The roleview `ImageView` represents the entity `Image` with all of its different aspects. The permanent properties are structured into root class `ImgRoot`, which is based on UDT `Image`; and the transient properties are structured into role class `ImagMapRole` based on UDT `ImgMap`

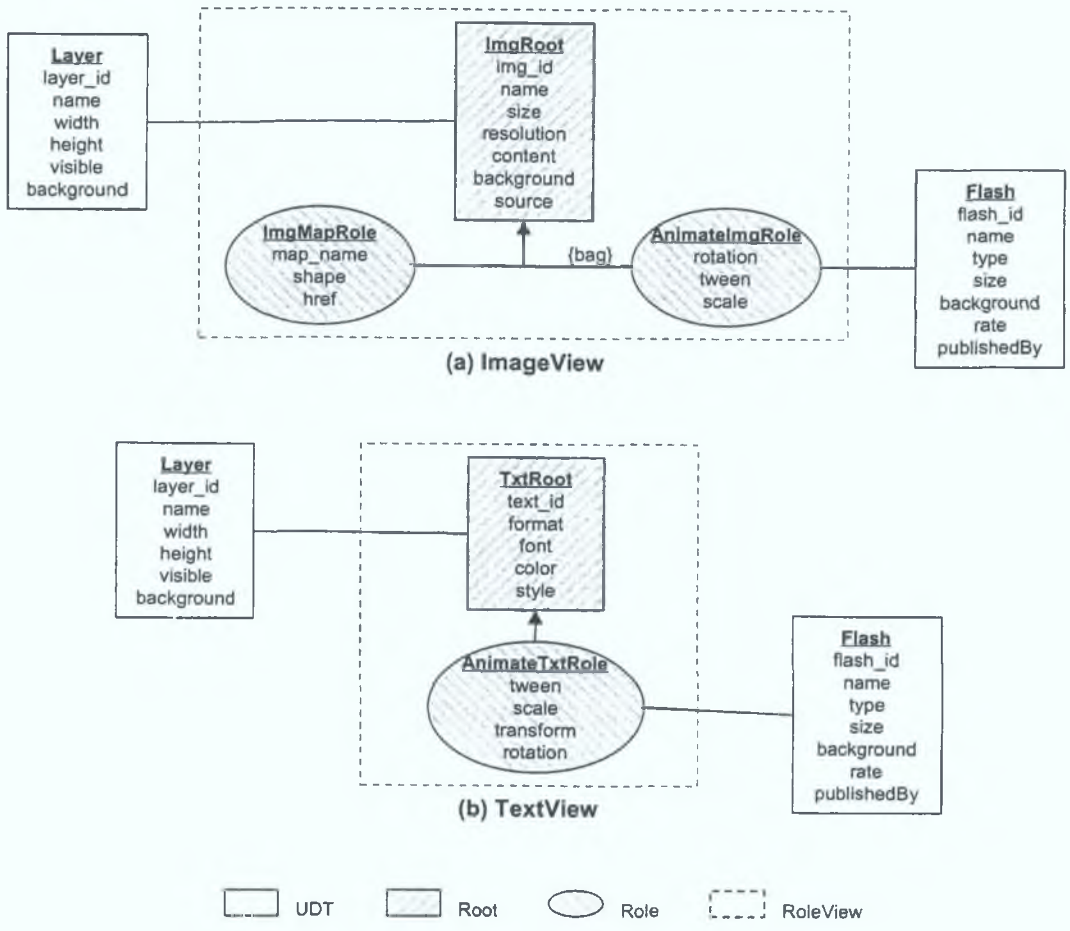


Figure 4.2: Roleview Examples.

and `AnimatedImgRole` based on UDT `AnimatedImg`. In this roleview, the associations between base UDTs are also derived, root `ImgRoot` links to UDT `Layer` and role `AnimatedImgRole` links to UDT `Flash`. The roleview `TextView` represents the `Text` entity and its role `AnimatedTxtRole`. We assume that a `Text` object plays a role of animated text when it appears in a `Flash` object. Hence, the `Text` UDT is the base for both root and role classes. The derived association is divided into the association between root class with UDT `Layer` and role class with UDT `Flash`.

Note that there are two options for generating extents for root and role classes: either retrieve the entire extent of the underlying UDT; or select only the subset of extent through the projection option. Since we concentrate on defining the roleview structure in this section, the root and role extents are simply generated by retrieving the entire extents of based UDTs. A full discussion on extents is provided in *section 4.2.4*.

Example 4.1 *ImageView Definition.*

create roleview ImageView as

root ImgRoot of Image is (

*select **

from Image_ObjTab)

role ImgMapRole of ImageMap is (

select map_name, shape, href

from ImgMap_ObjTab)

multirole AnimatedImgRole of AnimatedImg is (

select rotation, tween, scale, fla_ref

from AnimatedImg_ObjTab);

In *example 4.1* a roleview is comprised by three base UDTs: `Image`, `ImgMap` and `AnimatedImg`. Although the processing of a roleview definition is not covered until later in this chapter, it is possible to explain what happens with this sample definition. A new roleview `ImageView` definition will be stored in the schema repository. The `ImageView` defines one root `ImgRoot` and the two roles `ImgMapRole` and `AnimatedImgRole`. The latter is a `multirole` where multiple occurrences are allowed. The appropriate properties are easily retrieved from the metadata repository, with new role metadata generated and stored in the extended O-R schema repository. `ImgRoot`, `ImgMapRole` and `AnimatedImgRole` are based on previously defined UDTs, and `ImgMapRole` and `AnimatedImgRole` are restructured using the projection option. The derived relationship is represented by the attribute `fla_ref`.

The previous example illustrates that the root and role classes may be based on different UDTs. However, it is also possible for the root and role classes to be based on a single UDT provided that attribute sets are disjoint. *Figure 4.2(b)* illustrates this option, where a `Text` object links to a `Flash` object, it actually plays a temporary role as animated text.

Example 4.2 *TextView Definition.*

create roleview TextView as

root TxtRoot of Text is (

select text_id,format, font, color, style, layer_ref

from Text_ObjTab)

role AnimatedTxtRole of Text is (

select tween, scale, transform, rotation, flash_ref

from Text_ObjTab);

In *example 4.2*, root `TxtRoot` and role `AnimatedTxtRole` are based on a single UDT `Text`. The associations between UDT `Layer`, `Text` and `Flash` are derived by root class and role classes: attribute `layer_ref` specified in the root declaration; attribute `flash_ref` specified in the role declaration.

In a real world scenario, it is possible that an object exists only with its intrinsic properties and acquires a transient role afterward. A typical example is where a person becomes an employee at some point in time. Thus, it is possible to define a roleview with a root but no role specifications and later restructure the roleview with adding a role specification.

4.2.4 Generating Extents for Virtual Classes

As stated previously, a roleview definition includes defining the structure and generating the extents for virtual classes. In the last section, we provide clear semantics for defining the structure of roleview and the virtual classes. It is a complex issue while generating the extents for each individual virtual class and ensuring no-duplication or ambiguities between each extent. The semantics of generating extents are as follows:

- The extent of each virtual class is generated separately. In a roleview, a query is used to specify the structure and generate the extent for the virtual class. Multiple queries exist because multiple virtual classes are included in a roleview definition. The extent of each virtual class is generated individually after the structure is defined. The projection option presented by the `where` clause is used to filter the base UDT objects and generate the extents as result. If no `where` clause is specified, the entire extents for the underlying UDTs are used to create the extents of the virtual classes.

- To generate the extent for a role class, it is necessary to take the extent of the base UDT, and apply some predicate to generate the virtual extent. Where a virtual class is a role class, an extent is generated for the role class and its root class. In our roleview schema, each root class may have multiple role classes. Hence, the root extent contains the extents of all its role classes by default. For example, the extent of role class `Student` contains 2 objects and the extent of role class `Employee` contains 1 object, then the extent of their root class `Person` contains those 3 objects automatically.

- A root class is also a virtual class of base UDT, hence, the extent of the root class is generated by applying the project option to its base UDT extent. In addition, it is expressed by the union of the extents of all of role classes. In order to eliminate the overlap in the union, a filter function *distinct()* must be applied to the root class. It is similar to that of distinct query in SQL:1999. The expression of root extent is: E_{root} = distinct $(E(P_{root}) \cup E(P_{role1}) \cup E(P_{role2}) \ldots \cup E(P_{rolei}))$[1]

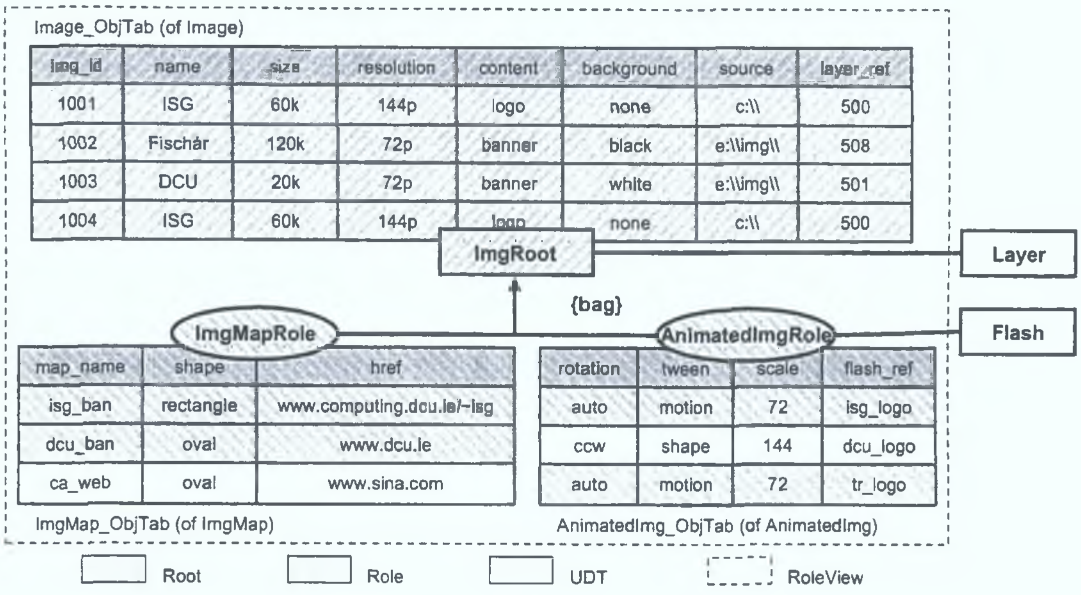
The next step is to generate extents for virtual classes using the semantics. *Example 4.3* generates the specific extent of the roleview from *example 4.1*.

Example 4.3 *ImageView Definition with Extent.*

```
create roleview ImageView as
root ImgRoot of Image is (
        select *
        from Image_ObjTab
        where name = 'Fischár')
role ImgMapRole of ImageMap is (
        select map_name, shape, href
        from ImgMap_ObjTab
        where map_name = 'dcu_ban' or shape = 'rectangle')
multirole AnimatedImgRole of AnimatedImg is (
        select rotation, tween, scale, fla_ref
        from AnimatedImg_ObjTab
        where tween != 'shape');
```

Figure 4.3 illustrates how the extents of root and role classes are generated for *example 4.3*. The non-shaded columns and rows are not part of the view specification but are part of base UDTs.

[1]The projection option is expressed by P(), which is applied to the virtual class specification.



img_id	name	size	resolution	content	background	source	layer_ref
1001	ISG	60k	144p	logo	none	c:\\	500
1002	Fischár	120k	72p	banner	black	e:\\img\\	508
1003	DCU	20k	72p	banner	white	e:\\img\\	501
1004	ISG	60k	144p	logo	none	c:\\	500

map_name	shape	href
isg_ban	rectangle	www.computing.dcu.ie/~isg
dcu_ban	oval	www.dcu.ie
ca_web	oval	www.sina.com

rotation	tween	scale	flash_ref
auto	motion	72	isg_logo
ccw	shape	144	dcu_logo
auto	motion	72	tr_logo

Figure 4.3: ImageView with Extents.

The `multirole AnimatedImgRole` {auto,motion} is represented using two rows. There are two role objects in the `ImgMapRole` class (which gets its extent from the `ImgMap_ObjTab` object table) and one object in the `AnimateImgRole` class. These three objects are part of the root extent by default. In addition, the `Image` named as '`Fischár`' is selected as root object according to the root specification. Hence, the root extent includes 4 objects. The extent for root class `ImgRoot` is generated as follows: $E_{\text{ImgRoot}} = E(P_{\text{ImgRoot}}) \cup E(P_{\text{ImgMapRole}}) \cup E(P_{\text{AnimatedImgRole}})$

In the illustrated roleview definition, it is allowed that a root class and its role class may be based on a single UDT. *Example 4.4* expands the definition shown in *example 4.2* with a `where` clause, and *figure 4.4* illustrates the result.

Example 4.4 *TextView Definition with Extent.*

create roleview TextView as

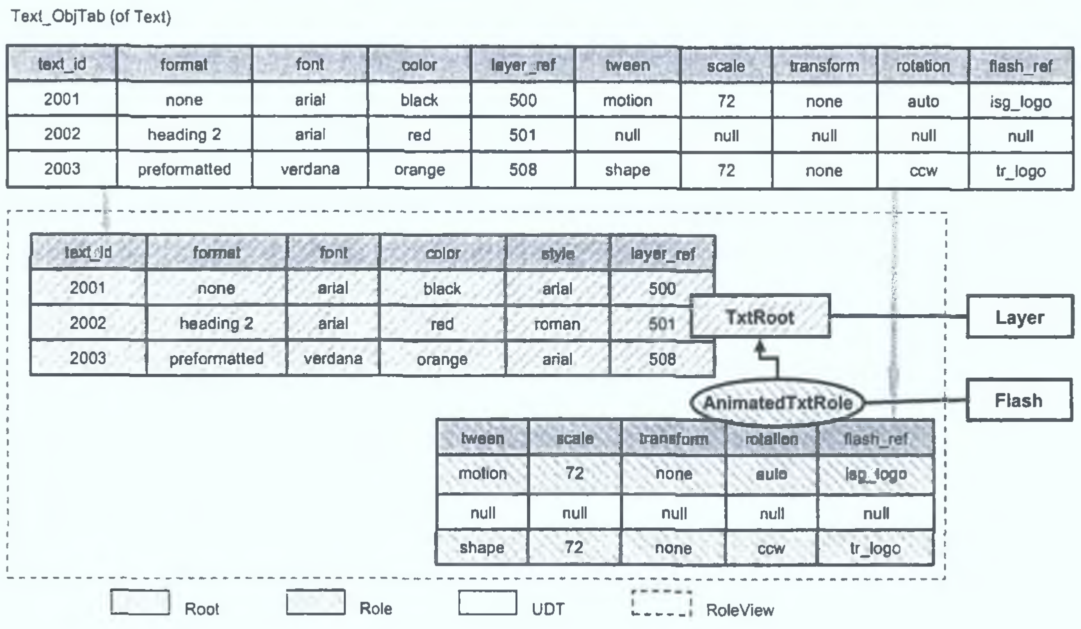
root TxtRoot of Text is (

> *select text_id, format, font, color, style, layer_ref*
>
> *from Text_ObjTab*
>
> `where text_id = '2002'`*)*

role AnimatedTxtRole of Text is (

> *select tween, scale, transform, rotation, fla_ref*
>
> *from Text_ObjTab*
>
> `where tween = 'motion' or tween = 'shape'`*);*

The extent of UDT `Text` is divided into `TxtRoot` extent and `AnimatedTxtRole` extent. According to the root specification, there is one `Text` object is selected as the root object. According to the role specification, there are two `Text` objects are selected into the role extent. As stated previously, these two objects are part of the root extent by default. The root object `Text '2002'` does not play role `AnimatedTxtRole` currently; hence there are three root objects in the root extent. The expression of root extent is: $E_{\text{TxtRoot}} = E(P_{\text{TxtRoot}}) \cup E(P_{\text{AnimatedTxtRole}})$



Text_ObjTab (of Text)

text_id	format	font	color	layer_ref	tween	scale	transform	rotation	flash_ref
2001	none	arial	black	500	motion	72	none	auto	isg_logo
2002	heading 2	arial	red	501	null	null	null	null	null
2003	preformatted	verdana	orange	508	shape	72	none	ccw	tr_logo

text_id	format	font	color	style	layer_ref
2001	none	arial	black	arial	500
2002	heading 2	arial	red	roman	501
2003	preformatted	verdana	orange	arial	508

tween	scale	transform	rotation	flash_ref
motion	72	none	auto	isg_logo
null	null	null	null	null
shape	72	none	ccw	tr_logo

Figure 4.4: TextView with Extents.

4.3 Restructuring Operations

In this section, restructuring operations that modify the structure of a roleview are described. The operations fall into two catalogues: *class* level and *object* level. There are three operations of restructuring roleviews, which are `rename`, `add` and `drop` operators. They are originally used to restructure the defined UDT in SQL:1999. We modify these operators and allow them to restructure the virtual classes of roleview. There are also three operations that manipulate role objects, which are `acquire`, `abandon` and `migrate` operators. Each operator is expressed in BNF format and followed by an example.

4.3.1 Class Level Operations

The formal expression of class level restructuring operations are represented in *definition 4*.

Definition 4 *Class Level Restructuring Operations.*

"alter" "roleview" roleview_name: Identifier

rename_ope |

add_ope |

drop_ope

;

rename_ope:

"rename" "root | role" old_class_name: Identifier "as" new_class_name: Identifier

;

add_ope:

*"add" (role_add_dcl)**

;

role_add_dcl:

> *"role | multirole" class_name: Identifier "of " type_name: Identifier "is"*
>
> *(sql_dcl)*[2]

;

drop_ope:

> *"drop" (role_drop_dcl)**

;

role_drop_dcl:

> *"role | multirole" class_name: Identifier*

;

Note that the keyword `alter` is used to emphasise the alteration of a roleview. The three possible expressions are all optional.

- `rename`

The `rename` expression is used to rename the virtual classes. There are two arguments: the first is the existing virtual class identifier and the second is a new virtual class identifier. If the new virtual class identifier already exists in the extended schema repository, an error message will be generated. A simple example is given in *example 4.5.* While a virtual class is renamed, the metadata stored in the schema repository is automatically updated.

Example 4.5 *Rename Operation.*

alter roleview ImageView

> *rename role AnimatedImgRole as Frame*

;

- `add`

The `add` expression is used to allow the root class to require role classes. The keyword `role` or `multirole` clarify the type of role class. Note that a `sql_dcl` substatement is used to construct the class structure and generate the extents, which is similar with the specification of role class in a roleview. For example, root `ImgRoot` requires a new role class `LogoRole`, which is based on existing UDT `Image`. *Example 4.6* illustrates this operation. Each statement contains multiple operations. However, if one operation fails, the system will rollback and unstore the updates.

Example 4.6 *Add Operation.*

alter roleview ImageView

> *add role LogoRole of Image is (*
>
> > *select type, size from Image_ObjTab)*

;

[2]The `sql_dcl` expression has been discussed at the section §4.2.2.

- `drop`

The drop expression allows the root class to remove its role classes. The keyword `role` or `multirole` is used to clarify role type. Like `add` operation, each statement allows multiple `drop` operations to be executed. One failure causes the rollback of the whole statement. This operation is only employed to role class, because dropping the root class results the roleview lost its identity. The `drop` operation is illustrated in *example 4.7.*

Example 4.7 *Drop Operation.*

alter roleview ImageView

drop role LogoRole

;

4.3.2 Object Level Operations

There are three object level operations are employed to the roleview definitions. They offer a flexibility that a root object may acquire or abandon its role object; and a role object may be transferred between different root object. It is not an option that a root or role object can be transferred between different roleviews.

- `acquire`

Definition 5 *Acquire Operation.*

"acquire" "role | multirole" class_name: Identifier

"from" "roleview" roleview_name: Identifier

"where" "root" sql_condition

;

The `acquire` operator is used to update a role extent by placing a new object into a role extent (and root extent by default). Note that substatement `sql_condition` is imported from SQL:1999 `where` clause [GP99]. It is also possible to acquire a new multirole by using the `acquire multirole` command. An informal expression is shown in *example 4.8.*

Example 4.8 Root Object Acquire Role Object.

```
acquire role ImgMapRole
from roleview ImageView
where root ImgRoot.img_id = '1003';
```

- abandon

Definition 6 *Abandon Operation.*

"abandon" "role | multirole" class_name: Identifier

"from" "roleview" roleview_name: Identifier

"where" "root" sql_condition

;

The abandon operator is executed when a root object does not play a role object anymore. A root object cannot be dropped unless its base UDT is removed from database schema. The substatement `sql_condition` is also reused from SQL:1999. *Example 4.9* shows the syntax.

Example 4.9 Abandon Role Object from Root Object.

```
abandon role ImgMapRole
from roleview ImageView
where root ImgRoot.img_id = '1003';
```

Once again, there are two important conditions that guarantee this operation succeeds: the root "1003" exists in the root extent; and it definitely has an `ImgMap` role. Otherwise, an error message will be generated.

- `migrate`

Definition 7 *Migrate Operation.*

"migrate" "role " class_name: Identifier

"from" "roleview" roleview_name: Identifier

"where" "root" sql_condition

"to" sql_condition

;

The `Migrate` operator permits the changing of a root object while retaining all role information. Under normal circumstances this requires a number of operations to delete all root and role data, and then add new root and role data [GSR96]. This is not necessary where a new object 'replaces' an existing one. For the purpose of simplicity and efficiency, only single role object can be transferred. It is not supported that the movement of multirole object. Again, the sql_condition is imported from SQL:1999, it provides the same functions that `where` condition presents. Following is an example of `migrate` operation.

Example 4.10 Migrate Role Object.

```
migrate role ImgMapRole
from roleview ImageView
where root ImgRoot.name = 'ISG'
to ImgRoot.name = 'DCU';
```

4.4 Conclusions

In this chapter the deployment of a role-based view system for O-R databases was presented, it is the major contribution of this thesis. In an O-R model, the real world entity is represented by an UDT, each schema is composed by many UDTs. Correspondingly, the virtual entity is represented by a role or root class, which is the virtual class of existing UDT; and each roleview is composed by many virtual classes. The relationships between base schema, UDT, roleview and virtual classes were illustrated in this chapter.

The semantics of defining a roleview were discussed before providing the syntax. In general, it falls into two catalogues: the rules for specifying individual virtual class and the rules for wrapping multiple virtual classes together. A virtual class cannot be specified unless the base UDT has been defined and stored in the database. The roleview is considered virtual schema, only one root class exists whereas the number of roles class is infinite. A root object is identified by object preserving semantics, and the identifier of a role object combines the base UDT object identifier and a system generated transient identifier. The definition of a roleview is completed by defining the structure and generating the extents for the included virtual classes. The syntax is expressed in a formal BNF format and followed by the real examples. The generation for virtual extents is represented by providing a clear semantics first, and extending the previous examples with the extent specifications.

In the last section, we provided number of restructuring operations that allow a roleview to be modified and manipulated. The operations fall into two catalogues: class level and object level. At the class level, a roleview is restructured by renaming the virtual class, adding or deleting virtual classes. At the object level, a root object can acquire or abandon its role objects, a defined role object is allowed to be transferred between different root object in a roleview.

At this point, we have offered the roleview concept and semantics of defining roleview, along with the O-R metamodel extensions introduced in last chapter. The implementation of the roleview mechanism is discussed in chapter 5.

Chapter 5

Implementation

One of the goals of this research is to deploy a role-based view system for O-R databases. Since the current O-R metamodel does not support the roleview, the first step of the implementation is to extend the metamodel with extra metaclasses. These metaclasses must be stored in the schema repository, along with existing metaclasses. The next step is to define the Roleview Definition Language (RDL) in BNF format and use ANTLR to parse the grammar; and generate the semantic actions for each of the production rules. The third step is to specify the roleview examples using RDL and store the metadata in the database. All of the roleview examples provided in this thesis are defined and stored in the O-R database and are thus all syntactically correct.

The prototype system is composed of a server side and client side prototype, it is illustrated in *figure 5.1*. The server side prototype extends the O-R metamodel; and the client side prototype validates RDL and stores the metadata in the schema repository. The details of building a server side prototype is provided in §5.1. In §5.2 a discussion on client side prototype is presented. In §5.3 details of experiments are described, and in §5.4, some conclusions are drawn.

5.1 Server Implementation

The server side prototype aims to extend the O-R metamodel with roleview metadata. The extra metaclasses and meta-objects must be stored in the schema repository along with the existing metaclasses and meta-objects. In reality, the server side prototype is divided into two sections: virtual O-R meta-UDT definitions and roleview meta-UDT definitions.

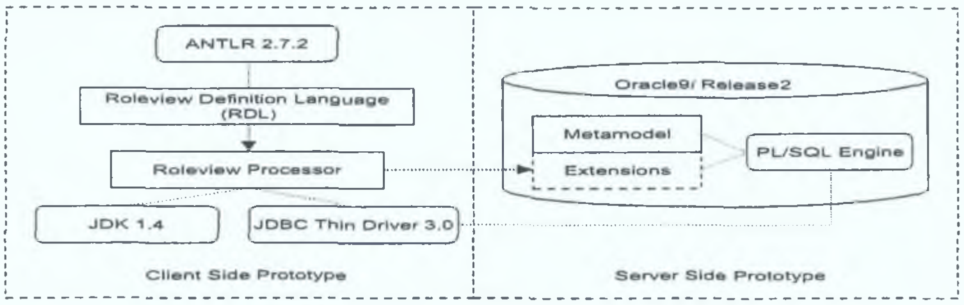


Figure 5.1: Prototype Overview.

5.1.1 Defining O-R Meta-UDTs

As stated previously, the O-R metadata is stored in the schema repository as static tables and views. The base meta-tables store information about the associated database and only the vendor may access these meta-tables. Hence, it is necessary for our approach to place a number of virtual meta-UDTs that represent the metamodel. These virtual meta-UDTs are defined by using SQL:1999 data definition language.

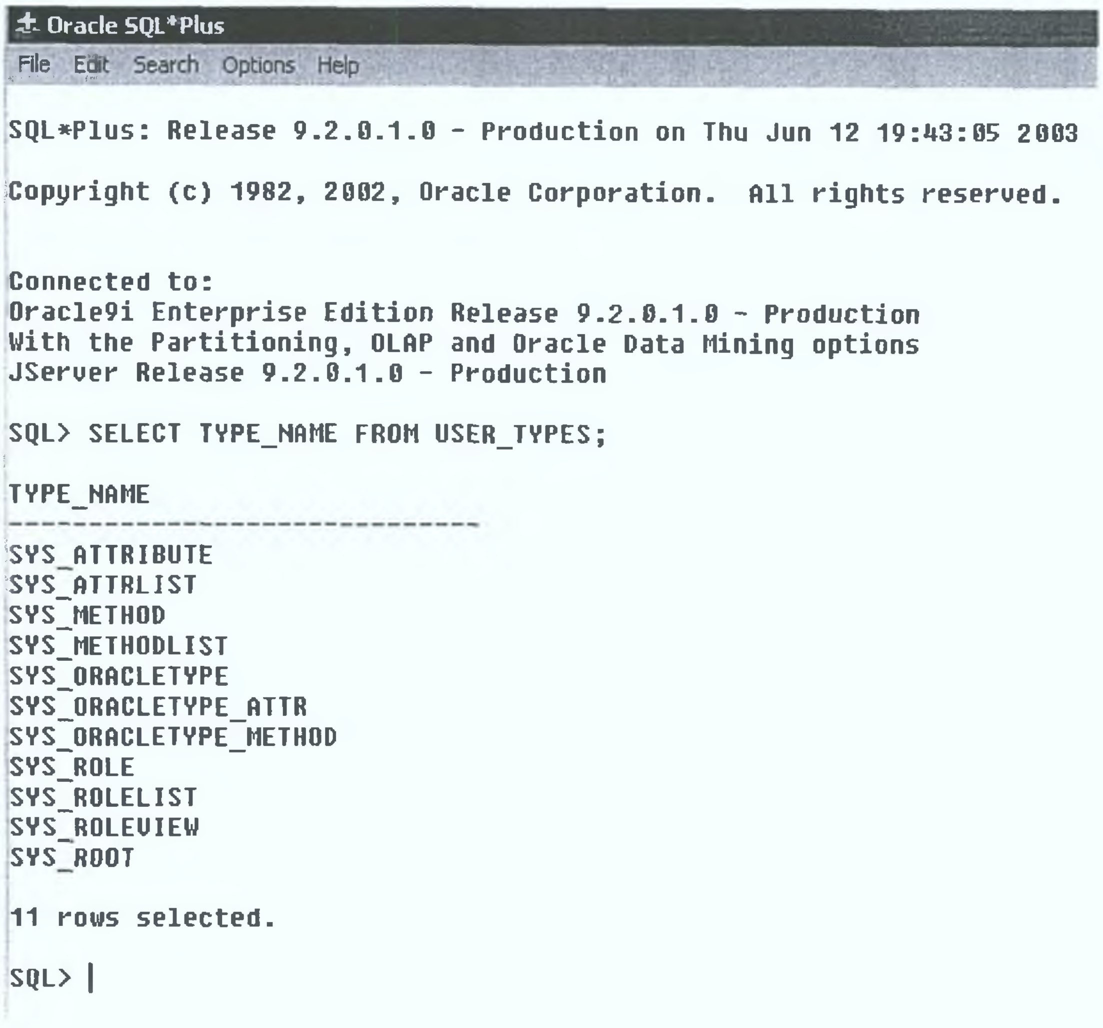
According to the extended metamodel presented in chapter 3, the roleview metamodel associates with `Type`, `Type_Attribute` and `Type_Method`. They can be observed in meta-views `ALL_TYPES`, `ALL_TYPE_ATTRS` and `ALL_TYPE_METHODS`. Hence, the first step is to define the UDTs which copy the structure of O-R `Type`, `Type_Attribute` and `Type_Method`. The following script presents how to construct a virtual UDT for O-R `Type`:

```
create type sys_OracleType as object (
     Type_Name varchar2(30),
     Type_OID raw(16),
     Typecode varchar2(30),
     Attributes number,
     Methods number,
     Final varchar2(3),
     SuperType_Owner varchar2(30),
     SuperType_Name varchar2(30),
     Local_Attributes number,
     Local_Methods number);
```

The next step is to create an object view for the virtual meta-UDT. As stated previously, the metadata can only be seen in the relational meta-views. Hence, it is necessary to abstract the relational view into an O-O format, where the tabular data is represented by an OID like any other objects. In an O-R model, the object view provides this functionality. The script of creating object view for `sys_OracleType` is presented as the follows:

```
create view sys_OracleType_View of sys_OracleType
  with object identifier (Type_OID) as
     select Type_Name, Type_OID, Typecode, Attributes,
     Methods, Final, SuperType_Owner, SuperType_Name,
     Local_Attributes, Local_Methods
     from ALL_TYPES;
```

The last step is to create the object table for the virtual meta- UDT, and update object table by retrieving the meta-objects represented in the object view. The object view generated in last step is only a virtual O-O representation of metadata, it is not possible that other meta-objects can access or link those meta-objects. Hence, we must create the meta-object table which physically store the meta-objects. As the result, the 'unaccessible' relational metadata is represented in an O-O format and can be accessed by other meta-objects. The script of creating object table for `sys_OracleType` is presented as the follows:



```
SQL*Plus: Release 9.2.0.1.0 - Production on Thu Jun 12 19:43:05 2003

Copyright (c) 1982, 2002, Oracle Corporation.  All rights reserved.


Connected to:
Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production

SQL> SELECT TYPE_NAME FROM USER_TYPES;

TYPE_NAME
------------------------------
SYS_ATTRIBUTE
SYS_ATTRLIST
SYS_METHOD
SYS_METHODLIST
SYS_ORACLETYPE
SYS_ORACLETYPE_ATTR
SYS_ORACLETYPE_METHOD
SYS_ROLE
SYS_ROLELIST
SYS_ROLEVIEW
SYS_ROOT

11 rows selected.

SQL> |
```

Figure 5.2: Virtual O-R & Roleview Meta-UDTs.

```
create table sys_OracleType_ObjTab of sys_OracleType (

     Type_Name primary key);

insert into sys_OracleType_ObjTab

     select * from sys_OracleType_View;
```

Other virtual O-R meta-UDTs, `sys_OracleType_Attribute` and `sys_OracleType_Method` are defined following the same procedure. The full script is presented in *Appendix A*.

5.1.2 Defining Roleview Meta-UDTs

As the extended metamodel presented in chapter 3, there are 5 meta-UDTs acquired: `sys_Roleview`, `sys_Root`, `sys_Role`, `sys_Attribute` and `sys_Method`. While defining roleview meta-UDTs, it is important to specify the relationships between the new meta-UDTs and the virtual O-R meta-UDTs. In the extended O-R metamodel, `sys_Root` and `sys_Role` associate with `sys_OracleType`; `sys_Attribute` links to `sys_OracleType_Attr`, and `sys_Method` links to `sys_OracleType_Method`. The association between the meta-UDTs is described by a built-in datatype `REF`. The script of defining roleview meta-UDTs is presented as the follows:

```
create type sys_Attribute as object (
    Attr_Name varchar2(30),
    Attr_Owner varchar2(30),
    Type_Attr_Ref REF sys_OracleType_Attr);
create type sys_Method as object (
    Method_Name varchar2(30),
    Method_Owner varchar2(30),
    Type_Method_Ref REF sys_OracleType_Method);
create type sys_AttrList as table of sys_Attribute;
create type sys_MethodList as table of sys_Method;
create type sys_Role as object (
    Role_RID RAW(16),
    Role_Name varchar2(30),
    Type_Ref REF sys_OracleType,
    Root REF sys_Root,
    IsMultiple varchar2(30),
    Attributes number,
    Methods number,
    Role_Attribute sys_AttrList,
    Role_Method sys_MethodList);
create or replace type sys_Root as object (
    Root_Name varchar2(30),
    Type_Ref REF sys_OracleType,
    SuperType varchar2(30),
    Attributes number,
    Methods number,
    RoleList sys_RoleList,
    Root_Attribute sys_AttrList,
    Root_Method sys_MethodList);
create type sys_RoleList as table of REF sys_Role;
create type sys_Roleview as object (
    Roleview_Name varchar2(30),
    Roleview_ID raw(16),
    Root REF sys_Root);
```

Note that there are three collection types are defined in the script, they are used to represent the collection of meta-objects. As *figure 5.2* illustrated, 5 roleview meta-UDTs are defined and stored in the schema repository, along with 3 virtual O-R meta-UDTs. In addition, 3 collection UDTs are defined to complete the extensions. As the result, there are 11 meta-types are defined in the server side prototype.

Since the roleview meta-UDTs model the structure of roleview meta-objects, it is necessary to define the roleview meta-object tables that store the roleview meta-objects. The script of defining object tables for the roleview meta-types is presented as the follows:

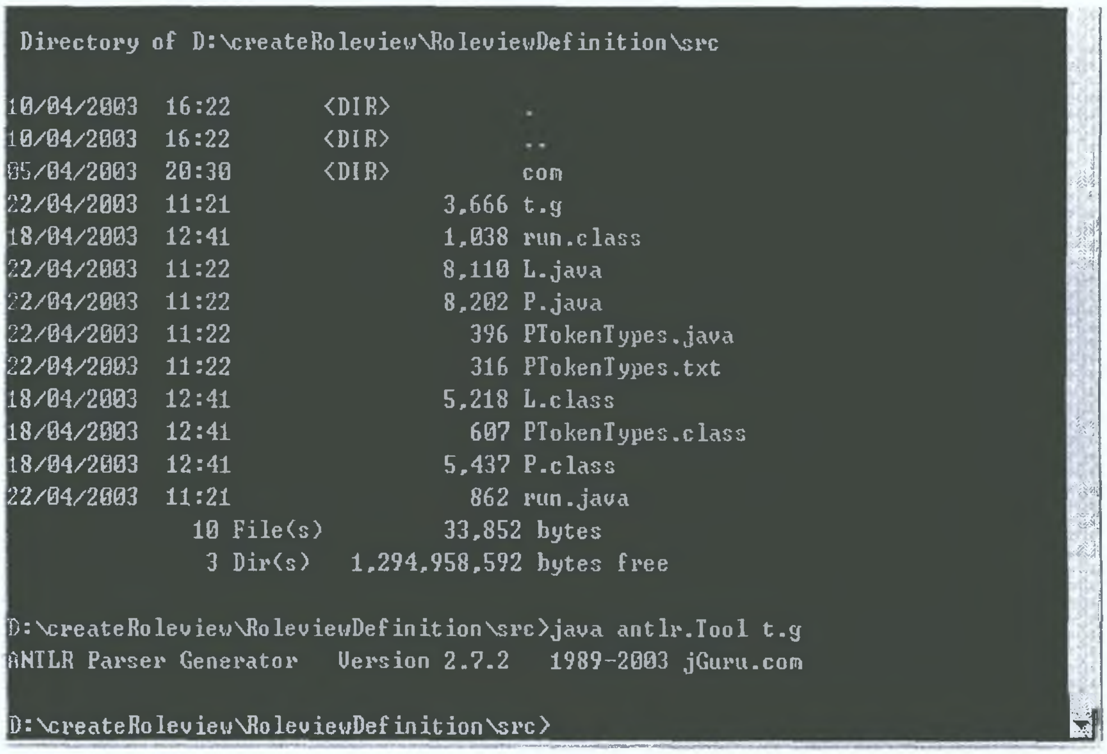
```
create table sys_Root_ObjTab of sys_Root (
    primary key (Root_Name),
    Type_Ref references sys_OracleType_ObjTab)
    nested table RoleList store as sys_Roles,
    nested table Root_Attribute store as sys_Root_Attributes,
    nested table Root_Method store as sys_Root_Methods;
create table sys_Role_ObjTab of sys_Role (
    primary key (Role_RID, Role_Name),
    Type_Ref references sys_OracleType_ObjTab,
    Root references sys_Root_ObjTab)
    nested table Role_Attribute store as sys_Role_Attributes,
    nested table Role_Method store as sys_Role_Methods;
create table sys_Attribute_ObjTab of sys_Attribute (
    primary key (Attr_Name, Attr_Owner),
    Type_Attr_Ref references sys_OracleType_Attr_ObjTab);
create table sys_Method_ObjTab of sys_Method (
    primary key (Method_Name, Method_Owner),
    Type_Method_Ref references sys_OracleType_Method_ObjTab);
create table sys_Roleview_ObjTab of sys_Roleview (
    primary key (Roleview_ID),
    Root references sys_Root_ObjTab);
```

5.2 Defining Roles

The client side prototype is also divided into two sections: parsing RDL and building roleview processor. Some background technologies such as ANTLR, Oracle JDBC Thin Driver and JDK are required to complete the prototype. These technologies are introduced briefly along with the prototype details.

5.2.1 Parsing RDL

RDL is specified in BNF format and parsed by ANTLR 2.7.2 in the client side prototype. The latest version of ANTLR can be obtained at `www.antlr.org`. ANTLR is a parser and translator tools that lets one define language grammar in either ANTLR or AST syntax. It is more than just



```
 Directory of D:\createRoleview\RoleviewDefinition\src

10/04/2003  16:22       <DIR>          .
10/04/2003  16:22       <DIR>          ..
05/04/2003  20:30       <DIR>          com
22/04/2003  11:21                3,666 t.g
18/04/2003  12:41                1,038 run.class
22/04/2003  11:22                8,110 L.java
22/04/2003  11:22                8,202 P.java
22/04/2003  11:22                  396 PTokenTypes.java
22/04/2003  11:22                  316 PTokenTypes.txt
18/04/2003  12:41                5,218 L.class
18/04/2003  12:41                  607 PTokenTypes.class
18/04/2003  12:41                5,437 P.class
22/04/2003  11:21                  862 run.java
              10 File(s)         33,852 bytes
               3 Dir(s)   1,294,958,592 bytes free

D:\createRoleview\RoleviewDefinition\src>java antlr.Tool t.g
ANTLR Parser Generator   Version 2.7.2   1989-2003 jGuru.com

D:\createRoleview\RoleviewDefinition\src>
```

Figure 5.3: Parse RDL Using ANTLR 2.7.2.

a grammar definition language, however, the tools provided allow one to implement the ANTLR defined grammar by automatically generating lexers and parser in either Java or Sather [Ant03]. In our prototype, ANTLR is used to parse RDL and generated the Java files, which represent the lexers and parsers. Note that RDL must be saved as a .g file and execute with command line:

```
java antlr.Tool Filename
```

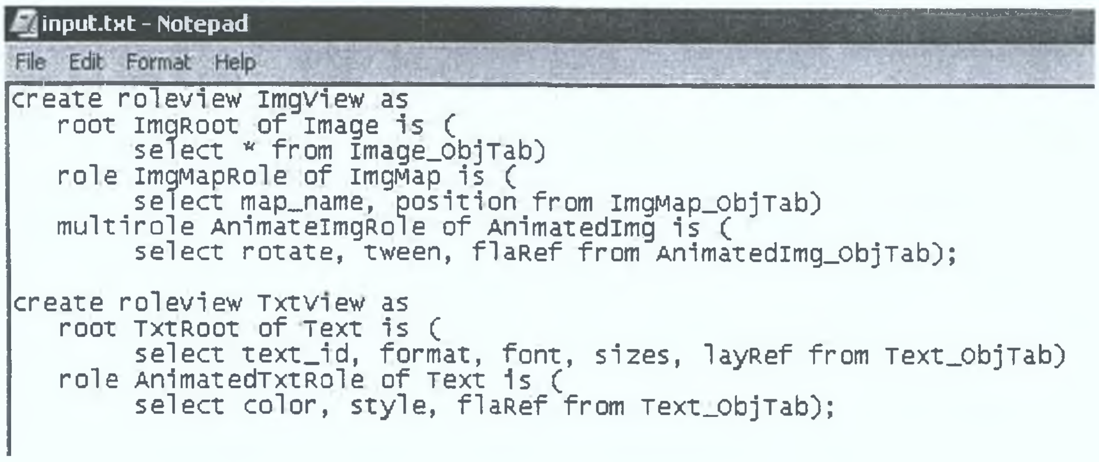
If an error occurs while validating RDL production rules, the programme is terminated by throwing an exception. If the validation is successful, the parsers and lexers are generated. Since RDL contains 1 parser and 1 lexer, there are 3 Java files and 1 Text file generated. *Figure 5.3* represents the results of parsing RDL. The detailed RDL is provided in *Appendix B.*

5.2.2 Building Roleview Processor

This task is completed by specifying Java programme which takes the RDL variables and stores them to the pre-defined meta-tables. There are 2 class libraries required:

- Oracle JDBC Thin Driver 3.0

Oracle JDBC is a standard Java interface for connecting from Java to databases. JDBC Thin driver provides the power and flexibility to use dynamic SQL statements in Java programmes [Ora01]. Using JDBC, a calling programme can construct SQL statements at runtime. The JDBC programme is compiled and run like any other Java programme. No analysis or checking of the SQL statements is performed. Any errors that are made in SQL code raise runtime errors. The latest version JDBC 3.0 provides two types of drivers, which are Thin driver and OCI. The reason we chose the Thin driver is that it is a 100% pure Java driver, and targeted for Oracle JDBC applets but can be used for applications as well. Because it is written entirely in Java, this driver



```
create roleview ImgView as
   root ImgRoot of Image is (
        select * from Image_ObjTab)
   role ImgMapRole of ImgMap is (
        select map_name, position from ImgMap_ObjTab)
   multirole AnimateImgRole of AnimatedImg is (
        select rotate, tween, flaRef from AnimatedImg_ObjTab);

create roleview TxtView as
   root TxtRoot of Text is (
        select text_id, format, font, sizes, layRef from Text_ObjTab)
   role AnimatedTxtRole of Text is (
        select color, style, flaRef from Text_ObjTab);
```

Figure 5.4: Roleview Definitions in RDL.

is platform-independent. It does not require any additional Oracle software on the client side. Oracle JDBC Thin driver can be obtained at `www.oracle.com`. In our prototype, Oracle JDBC Thin driver is used as the class library for building the roleview processor.

- JDK 1.4

JDK 1.4 is downloaded from `www.java.sun.com`. It is considered a class library that provides environments for creating and editing Java source code, and compiling and debugging programmes. In our prototype, `jdk1.4\jre\lib` directory is set in the PATH variables because it contains the Java Runtime facilities that are used when users execute a Java programme. The Java Runtime takes care of retrieving what it needs from the archive when the program executes.

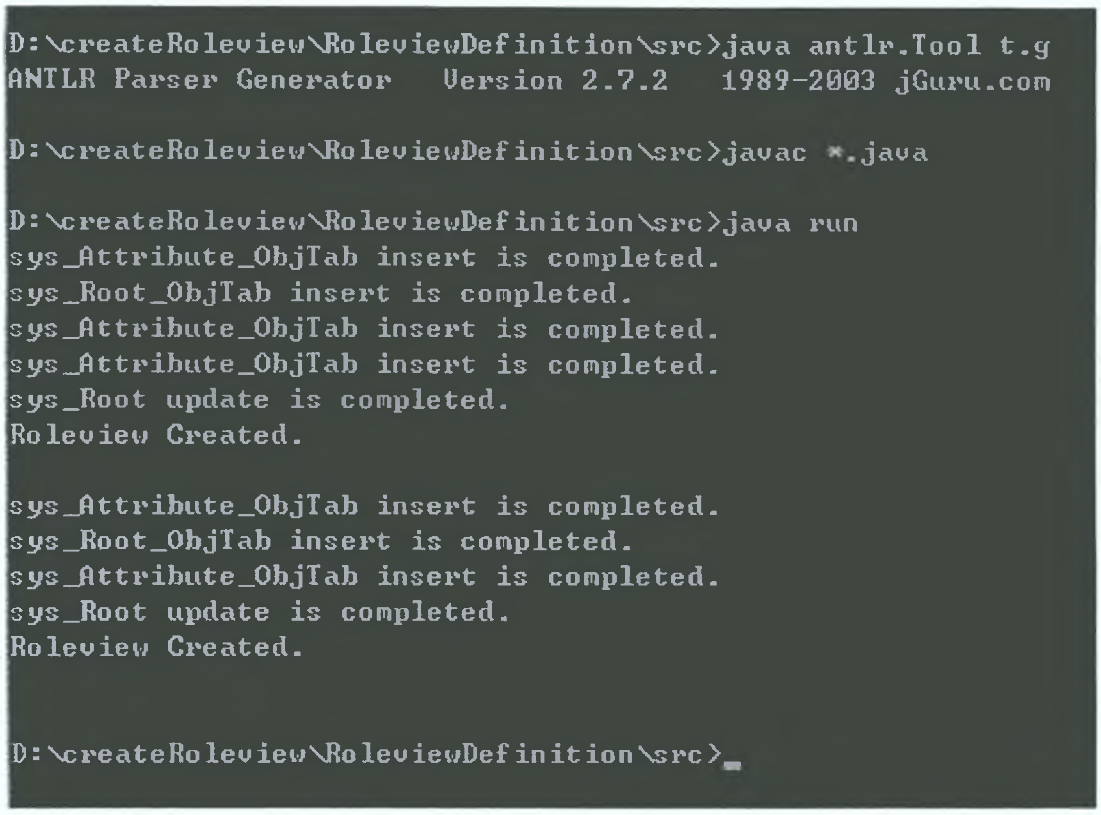
To store the roleivew metadata in the schema repository, a roleview processor is programmed that takes the roleview definitions as input, connect to the database, pass the roleview metadata to PL/SQL Engine and finally store the metadata to the meta-tables. In our prototype, there are 3 Java classes specified in the processor. The detailed code are provided in *Appendix C*, *D* and *E*.

5.3 Experiments

All of the roleview examples provided in this thesis are defined and stored in the schema repository. The premise of designing these roleview examples is that a Web Design Schema has been defined in the database. The base schema is defined by using SQL:1999 in Oracle9*i* Release 9.2.0.1.0 database. The script of defining the schema is represented as the follows:

```
create type Layer as object (
     layer_id number,
     name varchar2(30),
     width integer,
     height integer,
     visible varchar2(30),
     background varchar2(30));
```

```
create type Flash as object (
     flash_id number,
     name varchar2(30),
     type varchar2(30),
     size varchar2(30),
     background varchar2(30),
     rate varchar2(30),
     publishedBy varchar2(30));
create type Image as object (
     img_id number,
     name varchar2(30),
     size integer,
     resolution varchar2(30),
     content CLOB,
     background varchar2(30),
     source varchar2(30),
     layRef REF Layer) not final;
create type ImgMap under Image (
     map_name varchar2(30),
     shape varchar2(30),
     href varchar2(30),
     coordinates integer,
     target varchar2(30));
create type AnimatedImg under Image (
     width varchar2(30),
     height varchar2(30),
     coordinates integer,
     rotation integer,
     tween varchar2(30),
     scale boolean,
     flaRef REF Flash);
create type Text as object (
     text_id number,
     format varchar2(30),
     size integer,
```

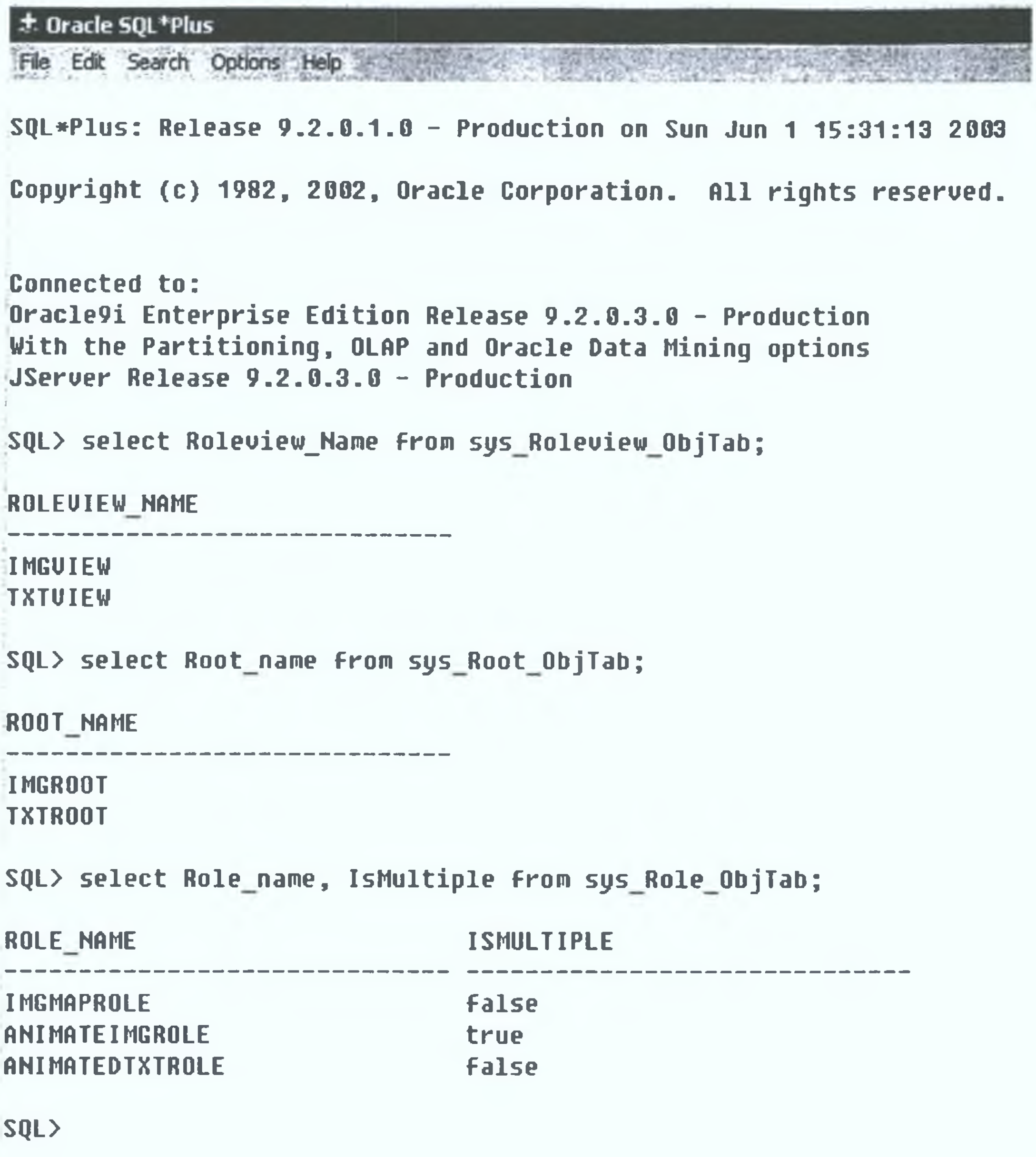


```
D:\createRoleview\RoleviewDefinition\src>java antlr.Tool t.g
ANTLR Parser Generator   Version 2.7.2   1989-2003 jGuru.com

D:\createRoleview\RoleviewDefinition\src>javac *.java

D:\createRoleview\RoleviewDefinition\src>java run
sys_Attribute_ObjTab insert is completed.
sys_Root_ObjTab insert is completed.
sys_Attribute_ObjTab insert is completed.
sys_Attribute_ObjTab insert is completed.
sys_Root update is completed.
Roleview Created.

sys_Attribute_ObjTab insert is completed.
sys_Root_ObjTab insert is completed.
sys_Attribute_ObjTab insert is completed.
sys_Root update is completed.
Roleview Created.


D:\createRoleview\RoleviewDefinition\src>_
```

Figure 5.5: Roleview Processor Execution.

```
     content CLOB,

     color varchar2(30),

     style varchar2(30),

     tween varchar2(30),

     scale varchar2(30),

     transform varhcar2(30),

     rotation varchar2(30),

     layRef REF Layer,

     flaRef REF Flash);

create table Layer_ObjTab of Layer;

create table Flash_ObjTab of Flash;

create table Image_ObjTab of Image;

create table ImgMap_ObjTab of ImgMap;

create table AnimatedImg_ObjTab of AnimatedImg;

create table Text_ObjTab of Text;
```

The roleview schema `ImageView` and `TextView` are defined as the virtual representations of base schema. Although the experiments were conducted within a laboratory environment, they demonstrate that the deployment of role-based views for O-R databases is syntactically correct. *Figure 5.4* illustrates the roleview definitions. In `ImgView` definition, the base UDTs, `Image`, `ImgMap` and



```
SQL*Plus: Release 9.2.0.1.0 - Production on Sun Jun 1 15:31:13 2003

Copyright (c) 1982, 2002, Oracle Corporation.  All rights reserved.


Connected to:
Oracle9i Enterprise Edition Release 9.2.0.3.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.3.0 - Production

SQL> select Roleview_Name from sys_Roleview_ObjTab;

ROLEVIEW_NAME
------------------------------
IMGVIEW
TXTVIEW

SQL> select Root_name from sys_Root_ObjTab;

ROOT_NAME
------------------------------
IMGROOT
TXTROOT

SQL> select Role_name, IsMultiple from sys_Role_ObjTab;

ROLE_NAME                      ISMULTIPLE
------------------------------ ------------------------------
IMGMAPROLE                     false
ANIMATEIMGROLE                 true
ANIMATEDTXTROLE                false

SQL>
```

Figure 5.6: Display Roleviews.

AnimatedImg have been stored in the schema repository and the object tables, Image_ObjTab, ImgMap_ObjTab and AnimatedImg_ObjTab contain the objects. Note that there are no any new attributes specified in root or role specifications, every attributes of virtual class must be retrieved from the base UDTs. In TxtView definition, both root class TxtRoot and role class AnimatedTxtRole are based on one UDT Text.

A roleview definition may be complex and long because it includes multiple virtual class specifications. RDL allows multiple roleview definitions to be saved at one single file as input for the purpose of easy maintaining. In *figure 5.4*, ImageView and TextView are specified in one single file, however, the roleview processor only process one definition at a time.

Once a roleview is defined, the roleview processor takes the roleview definition as input and store the roleview metadata in the meta-tables. *Figure 5.5* illustrates that the roleview examples have been successfully executed. Note that the roleview definition is processed in order. While executing a roleview definition, if one of the insertion or update is failed, the processor will rollback the SQL queries and throws an error message.

Figure 5.6 illustrates the stored roleview metadata, which displays the result of roleview processor. In meta-table `sys_Roleview_Objtab`, `ImgView` and `TxtView` are stored; `ImgRoot` and `TxtRoot` are stored in meta-table `sys_Root_ObjTab`; and `sys_Role_ObjTab` stores meta-objects `ImgMapRole`, `AnimatedImgRole` and `AnimatedTxtRole`. In this thesis, we only provide some simple SQL queries to display the roleview definitions, which are represented as the follows. A more sophisticated roleview query system is presented in [O'C03].

Example 5.1 *Query Root of a roleview.*

select Root.Root_Name

from sys_Roleview_ObjTab r

where r.Roleview_Name = 'ImgView';

Example 5.2 *Query Roles of a root.*

*select role.**

from sys_Root_ObjTab root, TABLE (root.RoleList) role

where root.Root_Name = 'TxtView';

Example 5.3 *Query Roles of a roleview.*

select re.Role_Name

from sys_Roleview_ObjTab rw, sys_Root_ObjTab rt, TABLE(rt.RoleList) re

where rw.Roleview_Name = 'ImgView'

and rw.Root.Root_Name = rt.Root_Name;

5.4 Conclusions

The deployment of the roleview system is represented by a working prototype. It includes the server side and client side implementations. The server side prototype is used to extend the O-R metamodel with roleview metaclasses; and the client side prototype is composed of RDL an a roleview processor.

The server side prototype is built by defining the virtual O-R meta-UDTs and roleview meta-UDTs in the schema repository. The scripts of meta-UDTs were represented and key issues were highlighted. At the client side, RDL grammar was parsed by using ANTLR parser, a roleview processor was built to store the roleview metadata to the pre-defined meta-tables. The technologies were introduced in order to provide a better understanding of the prototype. Finally, we offered the experiments which demonstrate the concept of roleview are implementable, and the semantics are syntactically correct.

Chapter 6

Conclusions

The aim of this research was to demonstrate that a standard such as the O-R model could be used as a basis for defining roleviews. Unlike other federated database research, one of the objectives of this research was to reuse the existing data model and concepts, and thereby, eliminate the need to define a new proprietary model. A second objective was to deploy a roleview system for O-R databases, rather than a traditional O-O view mechanism. In this chapter a review of the thesis is presented in §6.1; and options for further research are discussed in §6.2.

6.1 Thesis Summary

In chapter one, an introduction to federated database systems was presented. The Sheth and Larson architecture adopted by many researchers was described. In federated database systems, a view mechanism is crucial as it is used, to define exportable subsets of data; to perform a virtual restructuring of data; and to construct the integrated schema. Hence, a federated view must be formed in a semantic rich manner. In the last decade many researchers defined different forms of O-O views since the O-O model was considered the suitable canonical model before the emergence of the O-R model. The rigidity of existing O-O view mechanisms was discussed in this chapter. While defining O-O views for the object model, one option was to extend the data model to incorporate a role model. The concept of a role was introduced, along with the role features and role usage. The current O-R model has no facility to provide a rich view mechanism, but it provides an opportunity to develop the role-based view mechanism. The aim of this research was to employ the role concept, enhance its capabilities to construct a view schema, and demonstrate the usability of this idea through a working prototype and series of experiments using the O-R model.

One of the problems in O-O modelling was the lack of adequate object evolution mechanisms, also knowing as role modelling. The notion of roles presented many features that the traditional O-O model lacked, such as object migration, multiple occurrences and contest-dependent access. In chapter two, an examination and comparison of some of the existing O-O role projects were presented. The output from this critical analysis provided the requirements for the design of a role-based view mechanism for this research.

One objective was to implement the view language and display services in order to prove that the roleview schema could be constructed, and to provide an environment for testing views. Hence, the extensions to the O-R metamodel was described in chapter three. Before extending the O-R metamodel, it was necessary to examine the existing metamodel and clarify the possibility for role extensions. The examination of O-R metamodel was presented first, along with the analysis of the role-based view metamodel. In the process of extending the O-R metamodel, the discussions on associating metaclasses were emphasised. The output from this work was the capability of representing roles as an extended metamodel.

The major contribution of this thesis was to define a view definition language. RDL was presented in chapter four. One objective in designing RDL was the need to define the export schema (localised view). These schemas may be built by wrapping multiple virtual classes. Hence, it was necessary to clarify the semantics. Another objective was to query and display the view schema, hence, the demonstration of generating virtual class extents was also necessary. The method for defining virtual extents was presented, along with the semantics of how view processing takes place. An exportable schema in the federated database system may require restructuring operations, and thus, some class and object level restructuring operations were introduced.

The concept of role-based view system, and the syntax of RDL were proved by implementing a working prototype. In chapter five, the details of prototype implementation was provided. All examples illustrated in this thesis were constructed and queried to test the performance of roleview schema, and to provide data for future research.

6.2 Further Research

The O-R model is a new model, which is based on the relational model, while demonstrating the complex capabilities of the O-O model. It has been strongly developed in last few years, and there is a clear argument for using an O-R canonical model because of its increasing similarity to O-O model and its more widespread acceptance. Moreover, using the O-R model in the federated database systems provides new experiments that the traditional O-O model lacked. Since it is a new experiment, many further research are waiting for the exploration. Outlined below are primary areas for future work.

- Behaviour

One of the strengths of O-R model is the ability to incorporate behaviour into an object's UDT definition. The semantics of objects and their operations can be encapsulated within the UDT rather than buried in application programme code. The behaviour of an object is represented by *method* in the O-R model. It can be written in either PL/SQL or any other programming languages. Methods written in PL/SQL or Java are stored in the database; methods written in other languages such as C are stored with the application programme. In the latter case, a view mechanism cannot incorporate an object's operations since it cannot access the behaviour. In the first case, although the methods can be accessed, they are not fully implemented in the current version of O-R model. From a federated perspective, incorporating behaviours with views is a great challenge. Currently, one member of Interoperable Systems Group (ISG) focus on studying the behaviour of federated views [KR01]. The performance of roles will be improved if the behaviour is added to the roleview model.

- Integration

The objective of this research is to develop a semantic rich view system for the O-R model, which is considered the suitable canonical model in the federated database systems. As stated previously, a federated view must provide the functionalities for define export schema (localised view) and federated schema (global view). In this thesis, a powerful localised view is defined by using RDL. However, the issue of schema integration was not covered. In comparison to building exportable schema, the federated schema requires complex restructuring and *integration* operations. Hence, further research can be carried on at the area of extending the roleview system to facilitate various integration operations, which allow the federated system to combine roleviews from separate databases or information systems.

- Delegation

Delegation is an important concept to enrich the O-O model on the concept and implementation levels. One essential motivation to introduce delegation is to be seen in the shortcomings of inheritance to model certain aspects of the real world entity. There is a remarkable amount of work on O-O model which represents delegation [BD96, Mal95, Fra00]. However, the current O-R model does not represent this extensional inheritance feature. In this thesis, a role-based view system is designed from an implementation point of view, which allows the O-R model to support the concept of delegation. Hence, it is necessary to carry on the further research at the area of studying delegation from a concept point of view.

Bibliography

[AAG00] Albanoand, A., Antognoni, A. and Ghelli, G., View Operations on Objects with Roles for a Statically Typed Database Language, in *Knowledge and Data Engineering*, vol. 12(4), pp. 548–567, 2000.

[ABGO93] Albano, A. et al., An Object Data Model with Roles, in *The 19th Conference on Very Large Databases*, pp. 39–51, Dublin, Ireland, 1993.

[Ant03] ANTLR, Complete Language Translation Solutions, jGuru, 2003, URL `www.antlr.org`.

[BD77] Bachman, C. and Daya, M., The Role Concept in Data Models, in *The Third International Conference on Very Large DataBases*, pp. 464–476, Tokyo, Japan, October 6-8, 1977.

[BD96] Bardou, D. and Dony, C., Split Objects: A Disciplined Use of Delegation within Objects, in *Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA 96*, pp. 122–137, New York, United States, 1996.

[BE96] Bukhres, O. and Elmagarmid, A., *Object-Oriented Multidatabas Systems*, Prentice Hall, 1996.

[CB97] Cattell, R. and Barry, D., *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, 1997.

[DPZ02] Dahchour, M., Pirotte, A. and Zimányi, E., A Generic Role Model for Dynamic Objects, in *The 14th Advanced Information Systems Engineering nternational Conference, CAiSE'02*, Toronto, Canada, May 27-31, 2002.

[FBCP01] Fan, J. et al., Representing Roles and Purpose, in *First International Conference on Knowledge Capture, K-Cap'01*, Victoria, B.C., Canada, October 21-23, 2001.

[Fra00] Frank, U., Delegation: An Important Concept for the Appropriate Design of Object Models, in *Object-Oriented Programming*, vol. 13(3), pp. 13–18, 2000.

[GP99] Gulutzan, P. and Pelzer, T., *SQL-99 Complete, Really An Example-Based Reference Manual of the New Standard*, R&D Books Miller Freeman, Inc., 1999.

[Gro01] Group, W. S. W., XML Schema, in , 2001, URL `www.w3c.org`.

[GSR96] Gottlob, G., Schrefl, M. and Röck, B., Extending Object-Oriented Systems with Roles, in *ACM Transactions on Information Systems*, vol. 14(3), pp. 268–296, 1996.

[JHPS02] Jodlowski, A. et al., Objects and Roles in the Stack-Based Approach, in *The 13th International Workshop on Database and Expert Systems Applications, DEXA'02*, pp. 514–523, Aix En Provence, France, 2002.

[KR01] Kambur, D. and Roantree, M., Using Stored Behaviour in Object-Oriented Databases, in *The 4th International Workshop Engineering of Federated Information Systems, EFIS2001*, pp. 61–69, Berlin, Germany, October 9-10, 2001.

[KS91] Kappel, G. and Schrefl, M., Object/Behaviour Diagrams, in *Proceedings of the 7th International Conference on Data Engineering, IEEE Computer Society Press*, Kobe, Japan, 1991.

[Mal95] Malenfant, J., On the Semantic Diversity of Delegation-Based Programming Languages, in *Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA 95*, pp. 215–230, New York, United States, 1995.

[O'C02] O'Connor, G., A Metamodel Interface of Object-Relational Databases, *Tech. Rep. ISG-02-07*, Dublin City University, Glasnevin, Dublin 9, Ireland, 2002, URL `www.computing.dcu.ie/~isg`.

[O'C03] O'Connor, G., A Metadata Interface to Access Extended O-R Meta-Information, *Tech. Rep. ISG-02-13*, Dublin City University, Glasnevin, Dublin 9, Ireland, 2003, URL `www.computing.dcu.ie/~isg/`.

[Ora01] Oracle, *Oracle9i Database Concepts Release1(9.0.1)*, A88856-02, 2001.

[PBE95] Pitoura, E., Bukhres, O. and Elmagarmid, A., Object Orientation in Multidatabase Systems, in *ACM Computing Surveys*, vol. 27(2), pp. 141–195, 1995.

[Rad96] Radeke, E., Extending ODMG for Federated Database Systems, in *The 7th International Workshop on Databases and Export Systems Applications, DEXA'96*, pp. 304–319, Zurich, Switzerland, 1996.

[RKB01] Roantree, M., Kennedy, J. and Barclay, P., Integrating View Schemata Using an Extended Object Definition Language, in *The 9th International IFCIS Conference on Cooperative Information Systems, CoopIS 2001*, pp. 150–162, Trento, Italy, September 5-7, 2001.

[RS91] Richardson, J. and Schwarz, P., Aspects: Extending Objects to Support Multiple, Independent Roles, in Clifford, J. and King, R., eds., *The ACM SIGMOD International Conference on Management of Data, SIGMOD'91*, pp. 298–307, Denver, Colorado, May 29-31, 1991.

[Run92] Rundensteiner, E., Multiview: A Methodology for Supporting Multiple Views in Object-Oriented Databases, in *The 18th International Conference on Very Larger Databases, VLDB'92*, pp. 187–198, Vancouver, Canada, 1992.

[SAD94] Santos, C. D., Abiteboul, S. and Delobel, C., Virtual Schema and Bases, in *Proceedings of the International Conference on Extensive Data Base Technology, EDBT'94*, pp. 81–94, Springer-Verlag, Cambridge, U.K., 1994.

[SCGS91] Saltor, F., Castellanos, M. and García-Solaco, M., Suitability of Data Model as Canonical Models for Federated Databases, in *ACM SIGMOD Record*, 1991.

[SL90] Sheth, A. and Larson, J., Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases, in *ACM Computing Surveys*, vol. 22(3), pp. 183–236, 1990.

[SLR$^+$94] Scholl, M. H. et al., The COCOON Object Model, *Tech. Rep. 211*, ETH Zürich, Department Informatik, 1994.

[SLT91] Scholl, M., Laasch, C. and Tresch, M., Updatable Views in Object-Oriened Databases, in *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Daabases*, pp. 189–207, Springer Verlag, 1991.

[SM96] Stonebraker, M. and Moore, D., *Object-Relaional DBMSs The Next Great Wave*, Morgan Kaufmann Publishers, Inc., 1996.

[Ste00] Steimann, F., On the representation of roles in object-oriented and conceptual modelling, in *Data Knowledge Engineering*, vol. 35(1), pp. 83–106, 2000.

[Wan02a] Wang, L., An Analysis of Object-Relaional Model, *Tech. Rep. ISG-02-06*, Dublin City University, Glasnevin, Dublin 9, Ireland, 2002, URL `www.computing.dcu.ie/~isg/`.

[Wan02b] Wang, L., Extending the Object-Relational Metamodel to Facilitate the Definition of Roles, *Tech. Rep. ISG-02-11*, Dublin City University, Glasnevin, Dublin 9, Ireland, 2002.

[Wan03] Wang, L., Designing Roles for Object-Relational Databases, *Tech. Rep. ISG-03-02*, Dublin City University, Glasnevin Dublin 9, 2003, URL `www.computing.duc.ie/ ~isg/`.

[WCL97] Wong, R., Chau, H. and Lochovsky, F., A Data Model and Semantics of Objects with Dynamic Roles, in *The 13th International Conference on Data Engineering*, pp. 402–411, 1997.

[Won98] Wong, R., Heterogeous and Multifaceted Multimedia Objects in DOOR/MM: A Roles-Based Approach with Views, in *Parallel and Distributed Computing*, vol. 56, pp. 235–250, 1998.

[WR03] Wang, L. and Roantree, M., Desigining Role-Based View for Object-Relational Databases, in *The 5th International Workshop Engineering Federated Information Systems, EFIS'03*, Coventry, UK, 2003.

Appendix A

Roleview Metamodel DDL

********** Oracle Meta-UDTs Definitions **********

```
create type sys_OracleType as object (
Type_Name varchar2(30),
Type_OID raw(16),
Typecode varchar2(30),
Attributes number,
Methods number,
SuperType_Owner varchar2(30),
SuperType_Name varchar2(30),
Local_Attributes number,
Local_Methods number);
/
create type sys_OracleType_Attr as object (
Type_Name varchar2(30),
Attr_Name varchar2(30),
Attr_Type_Mod varchar2(30),
Attr_Type_Owner varchar2(30),
Attr_Type_Name varchar2(30),
Length number,
Precision number,
Scale number,
Character_Set_Name varchar2(30),
Attr_No number,
Inherited varchar2(3));
/
```

```
create type sys_OracleType_Method as object (

Type_Name varchar2(30),

Method_Name varchar2(30),

Method_No number,

Method_Type varchar2(30),

Parameters number,

Results number,

Inherited varchar2(3));

/

create view sys_OracleType_View of sys_OracleType

with object identifier (Type_Name) as

select Type_Name, Type_OID, Typecode, Attributes,

Methods, SuperType_Owner, SuperType_Name, Local_Attributes,

Local_Methods

from USER_TYPES;

/

create view sys_OracleType_Attr_View of

sys_OracleType_Attr with object identifier (Type_Name,

Attr_Name) as

select Type_Name, Attr_Name, Attr_Type_Mod,

Attr_Type_Owner, Attr_Type_Name, Length,

Precision, Scale, Character_Set_Name, Attr_No, Inherited

from USER_TYPE_ATTRS;

/

create view sys_OracleType_Method_View of

sys_OracleType_Method with object identifier (Type_Name,

Method_Name) as

select Type_Name, Method_Name, Method_No,

Method_Type, Parameters, Results, Inherited

from USER_TYPE_METHODS;

/

create table sys_OracleType_ObjTab of sys_OracleType (

Type_Name primary key);

/

create table sys_OracleType_Attr_ObjTab of sys_OracleType_Attr (
primary key (Type_Name, Attr_Name));
/
create table sys_OracleType_Method_ObjTab of sys_OracleType_Method
(primary key (Type_Name, Method_Name));
/
********** Insert metadata to Oracle Meta-UDTs **********
insert into sys_OracleType_ObjTab
select * from sys_OracleType_View;
/
insert into sys_OracleType_Attr_ObjTab
select * from sys_OracleType_Attr_View;
/
insert into sys_OracleType_Method_ObjTab
select * from sys_OracleType_Method_View;
/
********** Roleview Meta-UDTs Definitions **********
create type sys_Attribute as object (
Attr_Name varchar2(30),
Attr_Owner varchar2(30),
Type_Attr_Ref REF sys_OracleType_Attr);
/
create type sys_Method as object (
Method_Name varchar2(30),
Method_Owner varchar2(30),
Type_Method_Ref REF sys_OracleType_Method);
/
create type sys_AttrList as table of sys_Attribute;
/
create type sys_MethodList as table of sys_Method;
/
create type sys_Root;
/
create type sys_Role as object (
Role_RID RAW(16),
Role_Name varchar2(30),
Type_Ref REF sys_OracleType,
Root REF sys_Root,
IsMultiple varchar2(30),
Attributes number,
Methods number,
Role_Attribute sys_AttrList,
Role_Method sys_MethodList);
/
create type sys_RoleList as table of REF sys_Role;
/
create or replace type sys_Root as object (
Root_Name varchar2(30),
Type_Ref REF sys_OracleType,
SuperType varchar2(30),
Attributes number,
Methods number,
RoleList sys_RoleList,
Root_Attribute sys_AttrList,
Root_Method sys_MethodList);
/
create type sys_Roleview as object (
Roleview_Name varchar2(30),
Roleview_ID raw(16),
Root REF sys_Root);
/
create table sys_Root_ObjTab of sys_Root (
primary key (Root_Name),
Type_Ref references sys_OracleType_ObjTab)
nested table RoleList store as sys_Roles,
nested table Root_Attribute store as sys_Root_Attributes,
nested table Root_Method store as sys_Root_Methods;
/
create table sys_Role_ObjTab of sys_Role (
primary key (Role_RID, Role_Name),
Type_Ref references sys_OracleType_ObjTab,
Root references sys_Root_ObjTab)

nested table Role_Attribute store as sys_Role_Attributes,

nested table Role_Method store as sys_Role_Methods;

/

create table sys_Attribute_ObjTab of sys_Attribute (

primary key (Attr_Name, Attr_Owner),

Type_Attr_Ref references sys_OracleType_Attr_ObjTab);

/

create table sys_Method_ObjTab of sys_Method (

primary key (Method_Name, Method_Owner),

Type_Method_Ref references sys_OracleType_Method_ObjTab);

/

create table sys_Roleview_ObjTab of sys_Roleview (

primary key (Roleview_ID),

Root references sys_Root_ObjTab);

/
```

Appendix B

RDL Grammar

```
/*******************************************************************
Roleview definition grammar for parsing and creating the role-based
subschema in an O-R database
*******************************************************************/
// import java package which includes the user-defined classes
{
     import com.linkToJDBC.*;
}
/****************************************************************
Class: P
Extends: Parser
Date: April 2003
Author: Ling Wang
Desc: Roleview is defined in BNF syntax, then parsed and
translated by Antor-2.7.2. The definition is an extension of SQL:1999
****************************************************************/
class P extends Parser;
     options {
          k = 1;
     }
// begins by declaring global schema objects
{
     com.linkToJDBC.RoleDeclaration rd = new com.linkToJDBC.RoleDeclaration();
     com.linkToJDBC.ConnectToLing c = new com.linkToJDBC.ConnectToLing();
}
```

// PR (1)

// In a specification, one of more definitins are allowed, they are executed separately

```
specification

    :

    (definition)+

    EOF

    ;
```

// PR (2)

// Only one type of construct is allowed: a roleview schema and it is terminated by a semicolon (SEMI)

```
definition

    :

    {

        rd = new com.linkToJDBC.RoleDeclaration();

        c = new com.linkToJDBC.ConnectToLing();

    }

    ( roleview_dcl SEMI )

    {

        c.start();

    }

    ;
```

// PR (4)

// The roleview specification

```
roleview_dcl

    :

    ( "create" "roleview" roleview_name "as"

    "root" root_name "of" rootType_name "is" root_sql

    (role_dcl)*

    )

    ;
```

// PR (4a)

// The token is read and stored in memory

```
roleview_name

    :

    r:IDENTIFIER

    {
          c.setRoleViewName(r.getText());
     }
     ;
```

// PR (4b)

// The token is read and stored in memory

```
root_name
     :
     r:IDENTIFIER
     {
          c.setRootName (r.getText());
     }
     ;
```

// PR (4c)

// The token is read and stored in memory

```
rootType_name
     :
     r:IDENTIFIER
     {
          c.setRootTypeName (r.getText());
     }
     ;
```

// PR (4d)

// The role specification, which is allowed to be multiple

```
role_dcl
     :
     role_prefix role_name "of" roleType_name "is" role_sql
     {
          c.setRoleDeclaration(rd);
     }
     ;
```

// PR (4d_a)

// Clarify the type of role

```
role_prefix
     :
     "role" {rd.setIsMultiple(false);}
```

```
    |

    "multirole" {rd.setIsMultiple (true);}

    ;

// PR (4d_b)

// The token is read and stored in memory

role_name

    :

    r:IDENTIFIER

    {

        rd.setRoleName (r.getText());

    }

    ;

// PR (4d_c)

// The token is read and stored in memory

roleType_name

    :

    r:IDENTIFIER

    {

        rd.setRoleTypeName (r.getText());

    }

    ;

// PR (4d_d)

// Role SQL statement is read as a record and stored in memory

role_sql

    :

    s:  STATEMENT

    {

        rd.setSql (s.getText());

    }

    ;

// PR (4e_a)

// Root SQL statement is read as a record and stored in memory

root_sql

    :

    (
          n:STATEMENT
          {
               c.setRootSql(n.getText());
          }
     )+
     ;
/****************************************************************
Class: L
Extends: Lexer
Date: April 2003
Author: Ling Wang
Desc: Antlr-2.7.2 lexer specification of the parser class
****************************************************************/
class L extends Lexer;
// Tokens
IDENTIFIER
     :
     ( 'a'..'z' | 'A'..'Z' | '*' | '-' | '_' | ',')+
     ;
// Tokens as a record
STATEMENT
     :
     ('(' | ')') !  (~('\r' | '\n' | ';'))+
     ;
// Whitespace
WS
     :
     ( ' ' | '\t' | '\r' '\n' { newline(); } | '\n'
     { newline(); })
     {$setType(Token.SKIP);} //ignore this token
     ;
// Punctuation
SEMI
     :
     ';'
     ;
```

Appendix C

RDL Role Declaration Class

```
/*****************************************************************************
* Class: RoleDeclaration
* Desc: read the role token, specified in class ConnectToLing
* Author: Ling Wang
* Date: April 2003
****************************************************************************/
// import the library, including the user-defined package
package com.linkToJDBC;
import java.sql.*;
import java.util.*;
import java.io.*;
import java.lang.*;

public class RoleDeclaration {
     private String roleName;
     private String roleTypeName;
     private String roleSql;
     private int numberOfAttr;
     private Vector attrs = new Vector();
     private boolean isMultiple;

     // constructor
     public RoleDeclaration () {
          roleName = null;
          roleTypeName = null;
```

```
        roleSql = null;
        isMultiple = false;
    }
    // overload
    public RoleDeclaration (RoleDeclaration r) {
        roleName = r.getRoleName();
        roleTypeName = r.getRoleTypeName();
        roleSql = r.getSql();
        isMultiple = r.getIsMultiple();
    }
    // overload
    public RoleDeclaration (String r, String t, boolean m,
                            String sql) {
        roleName = r.toUpperCase();
        roleTypeName = t.toUpperCase();
        isMultiple = m;
        roleSql = sql;
    }
    public String getRoleName () {
        return roleName;
    }
    public String getRoleTypeName () {
        return roleTypeName;
    }
    public String getSql () {
        return roleSql;
    }
    public int getNumberOfAttr () {
        return numberOfAttr;
    }
    public Vector getAttrs () {
        return attrs;
    }
    public boolean getIsMultiple () {
        return isMultiple;
    }

    public void setRoleName (String role) {

         roleName = role.toUpperCase();

    }

    public void setRoleTypeName (String type) {

         roleTypeName = type.toUpperCase();

    }

    public void setIsMultiple (boolean m) {

         isMultiple = m;

    }

    public void setNumberOfAttr (int number) {

         numberOfAttr = number;

    }

    public void setAttrs (Vector lists){

         attrs = lists;

    }

    // remove unwanted token

    public void setSql (String sql) {

         String tmp = sql.replace(')', ' ');

         roleSql = tmp.replace('(', ' ' );

    }

}
```

Appendix D

RDL Roleview Declaration Class

```
/******************************************************************************
Class: ConnectToLing
Desc: read the token and store the token in pre-defined meta-tables.
Author: Ling Wang
Date: April 2003
******************************************************************************/
// import the library and user-defined package
package com.linkToJDBC;
import java.sql.*;
import java.util.Vector;
import java.util.Random;
import java.sql.Types;


public class ConnectToLing {
     private String rootSql;
     private String url = "jdbc:oracle:thin:@kiwi.isg.computing.dcu.ie:1521:kiwi";
     private String user = "lwang";
     private String pwd = "lwang";
     private Connection con;
     private Statement stmt;
     private Statement rootStmt;
     private Statement roleStmt;
     private Statement attrStmt;
     private Statement updateRootStmt;
     private Statement roleviewStmt;
```

```
    private String roleviewName;
    private String rootName;
    private String rootTypeName;
    private String rootSuperTypeName;
    private int numberOfAttr;
    private Vector roleDecList = new Vector();
    private Vector attrList = new Vector();

    // connect to database
    private void connectionDB() throws Exception {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        con = DriverManager.getConnection(url, user, pwd);
    }
    // close to database
    private void closeDB() throws Exception {
        con.close();
    }
    // set token to be upper case in order to query in SQL
    public void setRoleViewName(String s) {
        roleviewName = s;
        roleviewName = roleviewName.toUpperCase();
    }
    // set token to be upper case in order to query in SQL
    public void setRootName(String s) {
        rootName = s;
        rootName = rootName.toUpperCase();
    }
    // set token to be upper case in order to query in SQL
    public void setRootTypeName(String s) {
        rootTypeName = s;
        rootTypeName = rootTypeName.toUpperCase();
    }
    // trim the token and can be recognised by JDBC
    public void setRootSql(String s) {
        String tmp = s.replace(‘)’, ‘ ’);
        rootSql = tmp.replace('(', ' ');
    }
    // insert a new RoleDeclaration to the vector
    public void setRoleDeclaration(RoleDeclaration rd) {
    roleDecList.add(new RoleDeclaration(rd));
    }
    // write into meta table sys_Roleview
    private void writeToSys_roleview() throws Exception {
        roleviewStmt = con.createStatement();
        // generate the OID for each roleview
        String rid = generateObjectID();
        // string stores the query for updating
        StringBuffer insert = new StringBuffer();
        insert.append("insert into sys_roleview_ObjTab \n
            select '");
        insert.append(roleviewName);
        insert.append("', '");
        insert.append(rid);
        insert.append("', REF(c)\n from sys_Root_ObjTab c
            where \n ");
        insert.append("c.Root_Name = '");
        insert.append(rootName + "' ");
        insert.append("\n");
        // add into the batch file, only execute when 'commit' called
        roleviewStmt.addBatch(insert.toString());
        int [] counts = roleviewStmt.executeBatch();
    }
  // write into meta table sys_Root
    private void writeToSys_root() throws Exception {
        stmt = con.createStatement();
        rootStmt = con.createStatement();
        // string stores the supertype of root based UDT
        StringBuffer sql = new StringBuffer();
        sql.append("select supertype_name from user_types
            where type_name = '");
```

```
sql.append(rootTypeName + "'");
// execute the query
ResultSet rs = stmt.executeQuery(sql.toString());
// retrieve the results
while (rs.next()) {
     rootSuperTypeName = rs.getString("SUPERTYPE_NAME");
}
// close the query statement
stmt.close();
// string stores the query of updating meta table
StringBuffer firstRow = new StringBuffer();
firstRow.append("insert into sys_Root_ObjTab \n
     select '");
firstRow.append(rootName);
firstRow.append("', REF(c), '");
firstRow.append(rootSuperTypeName + "', ");
firstRow.append(numberOfAttr +
", 0, sys_RoleList(), sys_AttrList(), sys_MethodList()");
firstRow.append("\n from sys_OracleType_ObjTab c \n
     where c.Type_Name = '");
firstRow.append(rootTypeName);
firstRow.append("' \n");
// insert the first row sys_Root
rootStmt.executeUpdate(firstRow.toString());
// string stores the query for updating nested tables
for (int i = 0; i < attrList.size(); i++) {
     StringBuffer nstTables = new StringBuffer();
     nstTables.append(
     "insert into table ( \n select Root_Attribute from
     sys_Root_ObjTab r \n");
     nstTables.append("where r.Root_Name = '");
     nstTables.append(rootName + "') \n");
     nstTables.append("select '");
     nstTables.append( (String) attrList.elementAt(i)
     + "', '");
```

```
            nstTables.append(rootTypeName +
            "', REF(c) from sys_OracleType_Attr_ObjTab c \n");
            nstTables.append("where c.Attr_Name = '");
            nstTables.append( (String) attrList.elementAt(i)
            + "' and c.Type_Name = '");
            nstTables.append(rootTypeName);
            nstTables.append("' \n");
            // execute the nested table updating
            rootStmt.executeUpdate(nstTables.toString());
        }
        System.out.println("sys_Root_ObjTab insert is
        completed.");
    }
    // update meta table sys_Root after sys_Role insertion completes.
    private void updateSys_Root() throws Exception {
        updateRootStmt = con.createStatement();
        // loop of vector stores as role tokens.
        for (int i = 0; i < roleDecList.size(); i++) {
            RoleDeclaration r = (RoleDeclaration) roleDecList.get(i);
            // string stores the updating query
            StringBuffer nstTables = new StringBuffer();
            nstTables.append(
                "insert into table ( \n select RoleList from
                sys_Root_ObjTab r \n");
            nstTables.append("where r.Root_Name = '");
            nstTables.append(rootName + "' ) \n");
            nstTables.append("select REF(c) ");
            nstTables.append("from sys_Role_ObjTab c \n
            where c.Role_Name = '");
            nstTables.append(r.getRoleName() + "'");
            nstTables.append("\n");
            // execute the update
            updateRootStmt.executeUpdate(nstTables.toString());
        }
        System.out.println("sys_Root update is completed.");
```

```
}

// write into meta table sys_role

private void writeToSys_role() throws Exception {

    // loop of size of vector stores roles

    for (int i = 0; i < roleDecList.size(); i++) {

        RoleDeclaration r = (RoleDeclaration) roleDecList.get(i);

        roleStmt = con.createStatement();

        // string stores the update query of first row of meta table

    StringBuffer firstRow = new StringBuffer();

        firstRow.append("insert into sys_Role_ObjTab \n

        select '");

        firstRow.append(r.getRoleName());

        firstRow.append("', REF(o), REF(r), '");

        firstRow.append(r.getIsMultiple());

        firstRow.append("', ");

        firstRow.append(r.getNumberOfAttr() +

        ", 0, sys_AttrList(), sys_MethodList()");

        firstRow.append(

        "\n from sys_OracleType_ObjTab o, sys_Root_ObjTab

        r \n where o.Type_Name = '");

        firstRow.append(r.getRoleTypeName());

        firstRow.append("' and r.Root_Name = '");

        firstRow.append(rootName + "'");

        firstRow.append("\n");

        // execute the update for first row

        roleStmt.executeUpdate(firstRow.toString());

        // for the purpose of insert nested tables

        for (int j = 0; j < attrList.size(); j++) {

            // string stores the query

            StringBuffer nstTables = new StringBuffer();

            nstTables.append(

            "insert into table ( \n select Role_Attribute

            from sys_Role_ObjTab r \n");

            nstTables.append("where r.Role_Name = '");

            nstTables.append(r.getRoleName() + "') \n");
```

```
                    nstTables.append("select '");
                    nstTables.append( (String) attrList.elementAt(j)
                    + "', '");
                    nstTables.append(r.getRoleTypeName() +
                    "', REF(c) from sys_OracleType_Attr_ObjTab c
                    \n");
                    nstTables.append("where c.Attr_Name = '");
                    nstTables.append( (String) attrList.elementAt(j)
                    + "' and c.Type_Name = '");
                    nstTables.append(r.getRoleTypeName());
                    nstTables.append("' \n");
                    // execute the nested tables update
                    roleStmt.executeUpdate(nstTables.toString());
                }
            }
        }
        // write into meta table sys_Attribute
        private void writeToSys_attribute(String name, String
        typeName, String sql) throws Exception {
            stmt = con.createStatement();
            attrStmt = con.createStatement();
            // execute the query that retrieve the metadata
            ResultSet rs = stmt.executeQuery(sql);
            ResultSetMetaData rsmd = rs.getMetaData();
            numberOfAttr = rsmd.getColumnCount();
            // insert queries
            for (int i = 1; i <= numberOfAttr; i++) {
                attrList.add(rsmd.getColumnLabel(i));
                // string stores the query
                String inst = "insert into sys_Attribute_ObjTab
                select ' ";
                StringBuffer insert = new StringBuffer();
                insert.append(inst);
                insert.append(rsmd.getColumnLabel(i));
                insert.append("', '");
            insert.append(name);
            insert.append(
                "', REF(s) from sys_OracleType_Attr_ObjTab s
                where s.Attr_Name = '");
            insert.append(rsmd.getColumnLabel(i));
            insert.append("' and s.Type_Name = '");
            insert.append(typeName + "'");
            // add to the batch file
            attrStmt.addBatch(insert.toString());
        }
        // execute till 'commit' is called
        int[] updateCounts = attrStmt.executeBatch();
        // close the statement of retrieving metadata
        stmt.close();
        System.out.println("sys_Attribute_ObjTab insert is
        completed.");
    }
    // write into meta table sys_method
    private void writeToSys_method() {}
    // the only public method be called from ANTLR specification
    public void start() {
        // catch all the execeptions throwed by private methods
        try {
            connectionDB();
            copyOracleMeta();
            con.setAutoCommit(false);
            writeToSys_attribute(rootName, rootTypeName,
            rootSql.toString());
            writeToSys_root();
            for (int i = 0; i < roleDecList.size(); i++) {
                RoleDeclaration roles = (RoleDeclaration)
                roleDecList.get(i);
                writeToSys_attribute(roles.getRoleName(),
                roles.getRoleTypeName(),
                roles.getSql());
                roles.setNumberOfAttr(numberOfAttr);
                roles.setAttrs(attrList);
            }
            writeToSys_role();
            updateSys_Root();
            writeToSys_roleview();
            con.commit(); // start execute all statements
            con.setAutoCommit(true);
            System.out.println("Roleview Created.  \n");
        }catch (Exception ex) {
            System.err.println(ex.getMessage());
            // if any of statements is failed, rollback all insertions
            try {
                con.rollback();
                System.out.println("The transaction is rolled
                back.\n");
            }catch (SQLException s) {
                System.err.print(s.getMessage());
            }
        }finally {
            try {
                if (attrStmt != null) {
                    attrStmt.close();
                }
                if (rootStmt != null) {
                    rootStmt.close();
                }
                if (roleStmt != null) {
                    roleStmt.close();
                }
                if (updateRootStmt != null) {
                    updateRootStmt.close();
                }
                if (roleviewStmt != null) {
                    roleviewStmt.close();
                }
                closeDB();
            }catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
    // generate OID for roleview
    private String generateObjectID() {
        StringBuffer uid = new StringBuffer();
        Random random = new Random();
        String roleviewId = null;
        // set this random to be 32 bit binary
        for (int i = 0; i < 4; i++) {
            int tmp = random.nextInt();
            uid.append(Integer.toHexString(tmp));
        }
        roleviewId = uid.toString();
        return roleviewId;
    }
    // copy the new values from Oracle meta tables to sys_OracleType_Objtab,
   // sys_OracleType_Attr_ObjTab and sys_OracleType_Method_ObjTab;
    private void copyOracleMeta () throws Exception{
      stmt = con.createStatement();
      // update the meta table sys_OracleType
      StringBuffer oracleType = new StringBuffer();
      oracleType.append("insert into sys_OracleType_ObjTab\n");
      oracleType.append("select * from sys_OracleType_View\n");
      oracleType.append("where type_name not
      in (\n");
      oracleType.append("select type_name from sys_OracleType_ObjTab)");
      // update the meta table sys_OracleType_Attr
      StringBuffer oracleTypeAttr = new StringBuffer();
      oracleTypeAttr.append("insert into sys_OracleType_Attr_ObjTab\n");
      oracleTypeAttr.append("select * from sys_OracleType_Attr_View\n");
```

```
        oracleTypeAttr.append("where (type_name, attr_name) not
        in (\n");
        oracleTypeAttr.append("select type_name, attr_name
        from sys_OracleType_Attr_ObjTab)");
        // update the meta table sys_OracleType_Method
        StringBuffer oracleTypeMethod = new StringBuffer();
        oracleTypeMethod.append("insert into sys_OracleType_Method_ObjTab\n");
        oracleTypeMethod.append("select * from sys_OracleType_Method_View\n");
        oracleTypeMethod.append("where (type_name, method_name)
        not in (\n");
        oracleTypeMethod.append("select type_name, method_name
        from sys_OracleType_Method_ObjTab)");
        stmt.executeUpdate(oracleType.toString());
        stmt.executeUpdate(oracleTypeAttr.toString());
        stmt.executeUpdate(oracleTypeMethod.toString());
        stmt.close();
    }
}
```

Appendix E

RDL Main Class

```
/**************************************************************************
Class: run
Desc: executable main class
Author: Ling Wang
Date: April 2003
**************************************************************************/
import java.io.*;

class run{
   public static void main(String[] args) {
// read the input from a saved file
      File f = new File("D:\\createRoleview\\RoleviewDefinition\\input.txt");
      FileInputStream finput = null;

      try {
         finput = new FileInputStream(f);
      }catch (Exception e) {
         System.err.println("exception:  " + e);
      }
      try {
         L lexer = new L(new DataInputStream(finput));
         // call ANTLR specification
         P parser = new P(lexer);
         parser.specification();
      }catch (Exception ex) {
         ex.printStackTrace();
      }
   }
}
```