Representing relational database designs in the UML.

Xin. Zhao
University of Windsor

Representing Relational Database Designs in the UML

By

Xin Zhao

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2001

0-612-67607-2

952178

APPROVED BY:

[illegible]

Dr. Quan Wen (External Reader)

Department of Economics

Bandyopadhyay

Dr. Subir Bandyopadhyay (Internal Reader)

School of Computer Science

Liwu Li

Dr. Liwu Li (Supervisor)

School of Computer Science

[illegible]

Dr. Xiaobu Yuan (Chair)

School of Computer Science

ABSTRACT

Database Reverse Engineering (DBRE) refers to mapping relational schemas into semantically enriched models. Most of researches on DBRE map relational databases into the ER or EER models. They differ in input requirements and output. Unlike previous DBRE approaches, this thesis provides a methodology for mapping relational databases into an object-oriented designs in the UML. A distinct advantage of our approach is that it unifies objected oriented application designs and relational database designs. In this thesis, we first provide the Unified Modeling Language (UML) representations for basic relation schemas, with rigorous definitions for relational model concepts given by the UML metamodel and Object Constraint Language (OCL). Then, the paper classifies relational databases based on the types and number of inclusion dependencies. Finally, a general approach for mapping relational databases into object-oriented designs in the UML is specified.

Key words: UML, relational schemas, object-oriented model, database reverse engineering, frameworks, Entity-Relation Model, OCL, function dependency, inclusion dependency.

ACKNOWLEDGEMENTS

I would like to express my deep appreciation to my advisor, Dr. Liwu Li. His patience and guidance make this work possible.

Also, I am appreciative of Dr. S. Bandyopadhyay for being my internal reader. His invaluable suggestions and comments are the great helps to the thesis.

I would specially thank Dr. Quan Wen for being the external reader from his busy schedules.

Specially, thank Dr. Xiaobu Yuan for serving as the chair of the committee.

Finally, I want to express my appreciation to my wife Bo Shen, for her encouragement, support.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 Introduction

1.1 Overview of This thesis

1.1.1 Motivation

The development of database application involves a close working relationship between the application developers and the database development teams. Their work is often interrelated and overlapped. Often, software developers follow object-oriented analysis and design methodologies, using the OO CASE tool, such as the UML. Database designers may use other design methodologies and design tools, such as the Entity-Relation diagrams (ERD). Since software developers and database designers use different tools and methodologies, and they speak different languages, the cooperation and communication between them are often the biggest challenges in database application development [22].

The UML can be used to model the whole database development process, from business requirements to physical data models. If data modelers use UML notation to model databases, they too can follow object-oriented analysis and design methodologies. So, the software development and database design process both use one unified language and one tool, thus giving a great boost to database application development. Such a method is called object-relation mapping methods[22]. Many authors have studied object-relation mapping methods[22]. Methods and tools of object-relation mapping, like Rational Rose [22] and Sybase PowerDesigner [20] have been developed.

Also, the object-relation approach has advantages over the ER model and overcomes some limitations from the ER model. In the ER model, because it treats entities and relationships differently, during conceptual design phases, it often give problems when making choices between entities and relationships. Also, the ER model has other limitations. It cannot express general inclusion dependency; it can only express a special case of inclusion dependency-foreign key. Last but not least, the ER model can not express functional dependency easily. This gives problems both in database design phases and database reverse engineering.

Unlike Rational Rose, which applies the UML in relational database design in a forward engineering way, in this thesis, we apply the UML in relational database reversing engineering. The first part of the thesis is to use the UML diagrams and Object Constraint Language to represent relational databases. The second part of this thesis is to transform the relational schemas, represented by the UML notation, back to object oriented designs in the UML. So, the database designs and database application designs are unified into object-oriented notations in the UML.

1.1.2 Overview Of Major Works of This Thesis

1.1.2.1 Representing relational databases in the UML

The relational model is concerned with three aspects: data structure, data integrity, data manipulation [3]. When representing relational schemas, the UML representation of relational databases should give a consistent view of relational databases, without ambiguity and losing information. In this thesis, we focus on representing data structures as well as data integrity. The challenging part of the work is that, for the concepts in relational model, the UML representation should give corresponding formal definitions in the UML.

The UML diagrams and OCL generally are often used to deal with domain specific problems. The UML diagrams are generally used to model object-oriented software designs, and Object Constraint Language (OCL) is used to express constraints of domain-specific software designs. However, to represent relational databases, we use the UML at more abstract level: UML metamodel level. The primary responsibility of the UML metamodel layer is to define a language for specifying models [18]. For each modeling element, the UML defines a set of abstract syntax, semantics, rules, by using OCL and natural language. In representing relational databases in the UML, we introduce a set of stereotypes, tagged values, and give the formal definition in OCL which conforms to the ways of UML definition for its model elements.
One traditional expression for relational schema is as follows:

```
   Supplier {S#, sname, status, city}
```

In this thesis, we give a UML representation for schema `Supplier` as follows:

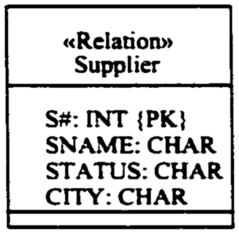


Figure 1.1 : A UML expression for relation `supplier`

In figure 1.1, we use the UML extension mechanisms to express integrity constrains: relation `supplier` is expressed as a stereotyped class; primary keys is expressed as tagged value {PK}. And, for each introduced stereotype and tagged value, we also give a formal OCL definition for its semantics at the UML metamodel level.

Comparing the two expressions above, we can see that the UML representation has the following advantages over traditional expressions in representing relational schemas:

- The UML represent relation schemas as stereotyped classes. The representation can be easily interpreted not only by programmers, but also by UML tools. And, this view provides foundations for integrating relational databases into object oriented designs.
- The UML expression clearly captures the integrity constraints of relations, including key constraints and other constrains. The expression has rigorous definition in the UML, thus eliminating ambiguities.
- The functional dependencies in relations can also be represented consistently in the UML, rather than expressed separately by additional symbols as traditional expressions. This will be addressed in later chapters.

1.1.2.2 Database Reverse Engineering

Database Reverse Engineering (DBRE) refers to understanding legacy databases and mapping them to a higher level of abstraction model , like the ER model. It deals with the problem of comprehending an existing system and recovering corresponding design specification [23].

The need for DBRE arises for many situations. In many organizations, there exists a large number of legacy database systems which lack of documentation, thus giving major difficulty to understand these legacy systems, and utilize them. Also, when integrating legacy systems into new systems, like providing a knowledge based interface to an existing database [19]. One way to solve these problems is to map the legacy systems into more semantically enriched model. In addition, DBRE facilitate technology transition [23]. A semantic enriched model will be needed, for example, in order to convert an existing database into an object-oriented database.

A wide range of DBRE methods has appeared since the beginning of 1980s. Most of them map the existing database systems into conceptual modes - the ER or EER model. However, each of these methods has its own characteristics. They differ in input requirements and output, and apply to different situations [14]. For example, Chiang [23,24] provides a DBRE method, which requires only part of information from legacy databases, including the database instance, relation schemas, primary keys and part of inclusion dependencies. The output of his approach is the Extended Entity-Relationship (EER) model. Similarly, Markowitz et al.'s [29] method needs key dependencies and key-based inclusion dependencies. Carellanos [1] gives a set of examples to map relational schemas into an object-oriented data model. In his paper, he also introduces some OO concepts, like Generalization, Cartesian aggregation, Interest Dependency etc.

DBRE approach in this thesis

In this thesis, we provide a new approach and methodology for DBRE. Unlike previous related works, we map relational databases into object-oriented designs in the UML. Comparing with the ER or EER model, the OO model in the UML captures more meanings. The constructs in the UML, like classes, associations, generalizations, are all well defined in the UML metamodel. A distinct advantage of our approach is that it unifies the OO designs and database designs. It will facilitate interoperations between relational databases and OO applications. For example, we can easily implement the mapped OO designs and integrate them into OO applications.

In our approach, we assume that database schemas are in third normal form (3NF) or BCNF; keys, inclusion dependencies (theses information can be retrieved from the database catalog or generated from database instances), data and code are also available for analysis.

Bases on the UML representations for relational databases, we classify the database schemas into binary types and star types according to the various types of Inclusion dependencies. Then, the classified schemas are mapped to an OO designs in the UML accordingly.

Our approach is generally as follows:

1. Basic Schema transformation: Inclusion Dependency Splitting, Candidate key splitting.
2.Classifying of relational databases, based on types for key-based Inclusion dependency.
3. Mapping the classified schemas accordingly.
3. Fusion and splitting for the mapped UML diagrams.

Let take one trivial example, for relation schemas:

EMPLOYEE (EMP_NUM, NAME, DEPARTMENT, SALARY)

DEPARTMENT (DEPT_ID, NAME, BUDGET)

Attribute DEPARTMENT in relation EMPLOYEE is a foreign key pointing to primary key DEPT_ID of relation DEPARTMENT. We map them back to object oriented designs in the UML as showed in figure 1.2:

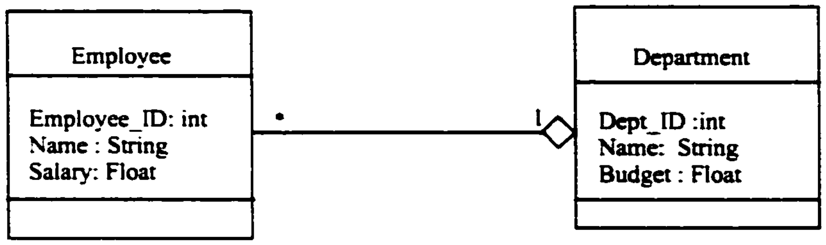


Figure 1.2 : An example of mapping OO designs from relational databases

In this example, the two relational schemas with foreign key dependency relationship are mapped into two classes with one-to-many aggregation relationship in the UML.

1.2 Organization Of This Thesis

This thesis is organized into six chapters.

Chapter 1 gives a brief introduction to the major works of this thesis with some trivial examples.

Chapter 2 introduces the UML. With the UML, this thesis focus on the architecture of the UML and Object Constraint Language. In particular, UML metamodel is addresses in detail.

Chapter 3 first give a brief introduction to the concepts of relational model. Then, a set of UML constructs are introduced to represent relational databases. To facilitate database reverse engineering, we classify relational databases according to two different categories.

Chapter 4 gives set of rule for transforming relational databases into object-oriented designs in the UML. At the end of chapter 4, a comprehensive example is introduced to illustrate the methods introduced in this thesis.

Chapter 5 summarizes the achievements of the thesis as well as some recommendations for the future work.

CHAPTER 2 The UML and Object Constraint Language (OCL)

2.1 Introduction to the UML

The unified Modelling Language (UML) is a standard language, released by OMG (Object Management Group) in 1997, for modeling software intensive systems. The UML can be used to specify, visualize, construct, and document software systems. A software system can be viewed from a number of perspectives. Different persons in a software system, such as end users, analysts, developers, testers, etc., may look at a software system in different ways at different times.

In an object-oriented software system, its architecture can be viewed from five interlocking views showed as follows:

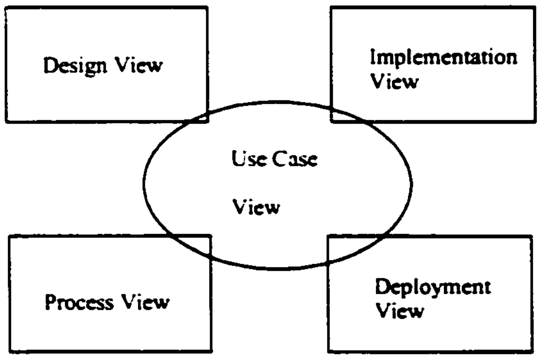


Figure 2.1: The UML modeling for a system's Architecture [9]

The UML supports all these views with a set of class diagrams.

Use case view: This view describes the requirements and functions of a system. End users, analysts, and testers can see it. In the UML, this view is modelled by `use case diagram`.

Design view: This view captures the functions of a system. This view is mainly used by developers and analysts. In the UML, `Class diagrams` and `Collaboration diagrams` model this view.

Process view: This view mainly encompasses the threads and processes of a system. In the UML, `active class diagram` represents the threads and processes.

Implementation view: This view encompasses the components and files that consist of the physical implementation of a system. In the UML, `component diagram` represents the configuration management of a system.
Deployment view: This view encompasses the nodes that how a software system' components are distributed. The `deployment diagram` describes this view.

Although the UML is a process-independent modeling language, it is well suited to develop object oriented systems in a user-case-driven, architecture-centric process. In this thesis, since we apply the UML to model structure of relational databases, we only deal with static part of the UML diagrams. In particular, class diagrams are talked in detail, and interaction diagrams are not addressed.

Class Diagrams

Classes are the important building blocks in object oriented systems. A class is actually an abstract data type. It describes the structure and behaviors of a set of objects. Classes can represent software things, hardware things, or conceptual concepts in an application domain. The UML provides a graphical representation for classes as the following figure:

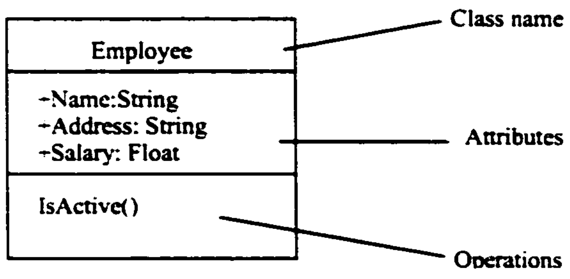


Figure 2.2: An example of class diagram

Attributes:

In a full form, the syntax of an attribute in the UML is :

```
[Visibility]name[multiplicity] [:type] [=initial-value]
[{property-string}]
```

The following are three pre-defined properties:

- `Changeable:`
- `AddOnly:`
- `Frozen:` this matches `const` in C++ and `final` in Java.

Operations

The syntax of an operation in the UML is
`[Visibility ] name [parameter-list] [:return type] [{property-string}]`

Visibility

- `Public:` Any outside classifier can use the feature.
- `Protected:` Any descendant of the class can use the feature.
- `Private:` only the classifier itself can use the feature.

Scope

Instance : each instance of the class holds its own value for the feature

Classifier: There exist only one value of the feature for all instances of the class. C++ and Java support this as `static` attributes and operations.

Multiplicity

Multiplicity means the number of instances a class may have. It applies to both classes and attributes of a class.

Relationships in the UML

A relationship is a connection among things. The UML defines four important relationships: dependencies, generalizations, associations, and realizations.

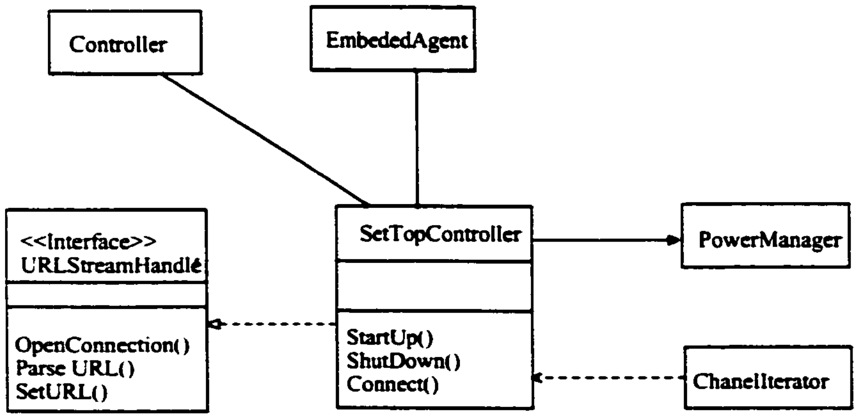


Figure 2.3: A class diagram with various relationships

Dependency:

It specifies that a change of one thing may affect another thing that uses it. A dependency is rendered as a dashed line. One common case of dependencies is that one class uses another class as an argument in the signature of an operation. If the interface and behavior of the used class changes, it affects the calling class. In figure 2.2, `ChanelIterator` is dependent on `SettopController`.

The UML defines a number of stereotypes for dependencies. These stereotyped dependencies apply dependency relationship among classes, among packages, among use cases, among state machines, etc.

Generalization

A generalization is a relationship between a general thing and a more specific thing. Generalization is supported directly by OO programming languages with Inheritance.

Association

An association is a structural relationship. It specifies that objects of one thing are connected to objects of another.

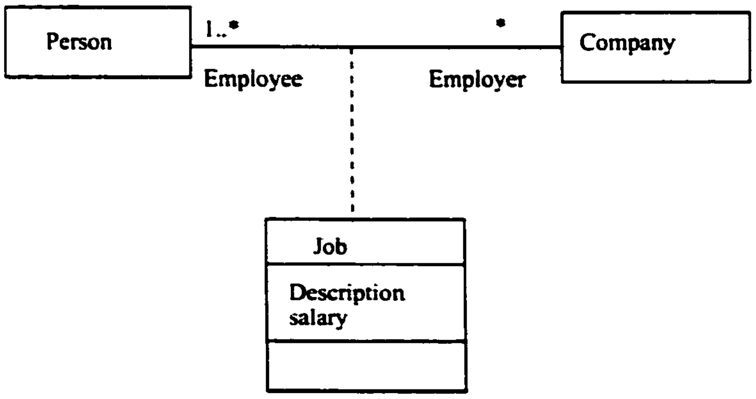


Figure 2.4: An example of association class

An association has a number of properties:

The role at each end of the association: It specifies the role of a class at the near end play.

Multiplicity: It means the number of instances a class may have. In figure 2.4, for each object of company, there exists at least one object of person working for it; each person can work for any number of companies.

Navigation: The associations can only navigate in a given direction.

Aggregation: This is a whole/part relationship.

Association classes

In the UML, an association class is a modeling element that has both association and class properties. In figure 2.4, class `Job` is an association class.

Realization

Most of the time, a realization specifies the relationship between an interface and the class that implements the interface.

Packages

In the UML, a package is a mechanism for organizing elements into groups. Packages should be loosely coupled and very cohesive. A package may own elements. It is a composite relationship between a package and the elements it owns.

The visibility of elements in a package can be controlled just as attributes are controlled in a class. There are two kinds of relationships between packages: dependencies and generalizations. The following diagram shows one example of package:

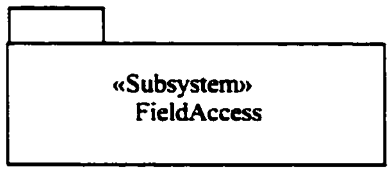


Figure 2.5: An example of package

2.2 Extension Mechanisms of the UML

The UML provides a rich set of modeling concepts and notations that have been carefully designed to meet the needs of typical software modeling projects. However, in some cases, users may need new modeling elements, or need to attach domain-specific information to a modeling element. The UML metamodel provides three extension mechanisms to let users add new building blocks, create new properties, and specify new semantics. These three extension mechanisms are Constraint, Stereotype, and TaggedValue [18].

Stereotype

Stereotypes provide a way of classifying UML elements so that they behave in some way like new "virtue" metamodel elements. A stereotype must refer to a base UML element and a stereotype share the attributes, associations, and operations of its base class, but it may have additional constraints and different meaning. Unlike Constraints and tagged values, any model element can only be marked as most one stereotype.

At UML metamodel level, the Sterotype metaclass is a subtype of GeneralizableElement. If a sterotype is a subtype of another stereotype, it inherits all of the constraints and tagged value from its stereotype super-type.

UML predefines a set of stereotypes for its model elements. The following diagram is one example:

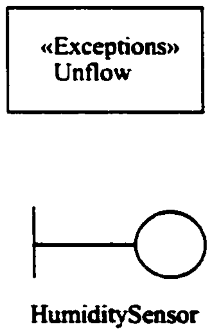


Figure 2.6: Stereotypes

In figure 2.6. class `HumiditySensor` is rendered as a new symbols; class `Underflow` is stereotyped as «Exceptions».

Constraint

In the UML metamodel, a Constraint can be directly attached to a built-in model element as well as a stereotyped element to give semantic restrictions to that Model element. Constraints apply to both UML metamodel elements and user-level model elements. Users are free to choose languages to write constraints, such as Object Constraint Language, programming languages, mathematical notations, or natural language.

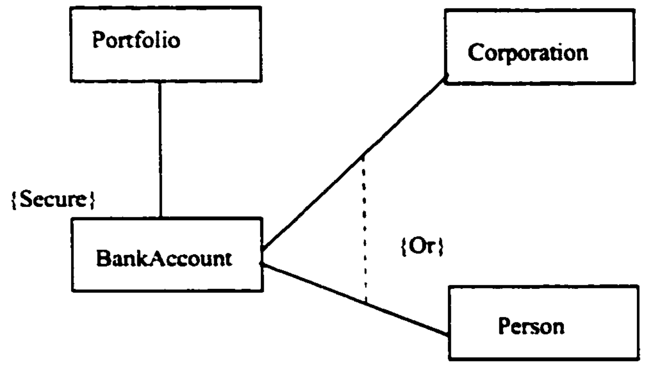


Figure 2.7: Constraint

In figure 2.7, constraint {secure} specifies that, across an association between `Portfolio` and `BankAccount`, communication is encrypted, and constraint {Or} specifies that class `BankAccount` can not associate class `Corporation` and class `Person` at the same time.

TaggedValue

A tagged value is a (Tag, Value) pair that allows users to add arbitrary information to any model element as well as a stereotyped element. A tag is a name string that is unique for a given element. Values should be represented as strings to give arbitrary information for a tag. The interpretation of a tag is beyond the scope of UML, and users or tool generators must determine it. For a model element, whether it is stereotyped or not, it can specify a list of tags. Like Constrains, tagged values apply both to UML metamodel elements and user-level model elements. Constraints and tagged values are uses to extend the semantics of model elements which they are attached to.

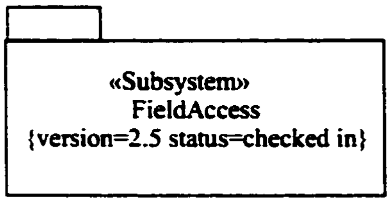


Figure 2.8: Tagged Value

In figure 2.8, Two tagged values are defined for the subsystem `FieldAccess`. The tag "version = 2.5" denotes the current version of the element. The tag "status = checked in" denotes the status of that element.

Abstract Syntax of Extension Mechanism [18]

The complete abstract syntax for the extension mechanisms is expressed as the diagram below [18]:

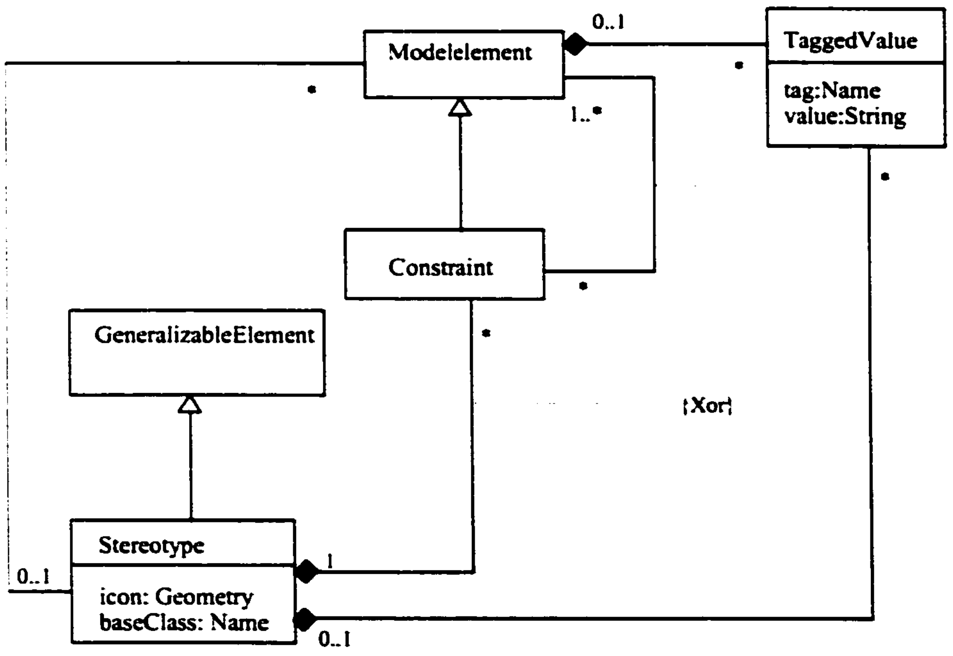


Figure 2.9:Abstract syntax of Extension Mechanisms

2.3 The UML metamodel

The UML metamodel is defined as four-layer metamodel architecture, including meta-metamodel, metamodel, model, user objects (user data) [18]. The descriptions for each model is summarized in the following table.

Layer	Description	Example
Meta-metamodel	Defines the language for specifying metamodels	`MetaClass, MetaAttribute`
metamodel	An instance of a meta-metamodel. Defines the language for specifying a model	`Class,Attribute, Association`
model	An instance of a metamodel. Defines a language to describe an information domain.	`Employee, department`
User objects (user data)	An instance of a model. Defines a specific information domain.	`<smith>, <human resource>`

Table 1: UML metamodel architecture

In this thesis, to model relational databases, we mainly deal with the UML at metamodel level. So, we will focus our discussions on the UML metamodel.

Metamodel

In the UML architecture, metamodel is an instance of a meta-metamodel. The primary responsibility of the metamodel layer is to define a language for specifying models. The UML specification [18] defines metamodel as a logic model rather than a implementation model. In the UML, the metamodel is described in a semi-formal manner using the following views:

- Abstract Syntax: The abstract syntax is presented in a UML class diagram showing the attributes of the metaclasses and their relationships. The attributes of a metaclass are enumerated with a short explanation.

- Well-formedess rules: The well-formedess rules refer to the static semantics of UML metaclasses. The rules must be satisfied for the construct to be meaningful. The rules can be specified by the OCL or natural language.
- Semantics. The meaning of the constructs is defined using natural language.
- Standard Elements: For each metaclass, it can define a set of standard elements using UML extension mechanisms. The standard elements include stereotypes, constraints, and tagged-values.
- Notes: This may contain rationales for the uses of the constructs, and examples illustrating the uses of the constructs. The notes are all written in natural language.

For example, element `Association` is one construct in the UML. At the UML metmodel level, the semantic and syntax for element `Association` is defined as follows [18]:

Association:

An association defines a semantic relationship between classifiers. The instances of an association are a set of tuples relating instances of the classifiers.

Attributes

Name: The name must be unique

Associations

Connections: an association consists of at least two AssociationEnds.

Stereotypes

Implicit

Standard Constraints

Xor

Tagged values

Persistence

Also, element `association` has many well-formedess rules. One rule is defined by natural language and OCL as follows:

`The AssociationEnds must have a unique name within the Association.`

```
Self.allConnections->forall (r1,r2|r1.name=r2.name implies r1=r2)
```

From the example above, we can see that one rule for element `association` is written in the OCL, and the OCL statement conforms to the syntax of element `association`.

Model

A model is an instance of a metamodel. The model layer is used to define a language describing an information domain. Any user's design, like class diagrams, sequence diagrams, activity diagrams, are all in model level.

User object

A user object is an instance of a model. In the UML, object diagrams is in the level of user object. A object diagram is a snapshot of a class diagram. In figure 2.9 below, object `P` is an instance of class `Person`, and object `D1` is an instance of class `Department`.

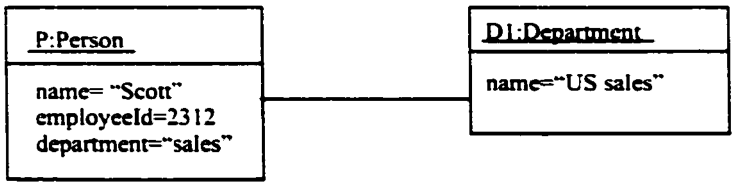


Figure 2.10: An Object Diagram

2.4 Object Constraint Language (OCL)

When applying UML in software modeling, there often exist needs to describe additional constraints on object models. The additional constraints can be expressed by natural language. However, this approach often results in ambiguities. Object Constraint Language (OCL) is developed to avoid ambiguous constraints. OCL is a formal language for expressing constraints [18]. In the UML semantics, OCL is used to specify the well-fomedness rules in the abstract syntax. Also, at model level, UML modelers can use OCL to specify application-specific constraints. OCL has the following characteristics:

- It is a pure expression language. When an OCL expression is evaluated, it has no side effects.
- It is a modeling language, not a programming language. It main purpose is to express constrains as a supplement to the UML diagrams.
- It is a typed language. It predefines a set of types.
- It roots in first-order-logic.

There are two basic types of constraints. One Constraint is stereotyped as `«invariant»`, which is associated with a Classifier. This constraint is used to express static constraints, which must be true for all instances of that type; Another Constraint is stereotyped as `«precondition»`, and `«postcondition»`. This constraint is associated with operations or methods to express dynamic constraints.

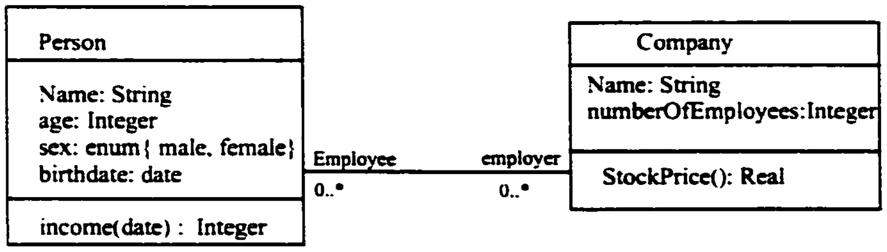


Figure 2.11: A class diagram example

Each OCL expression is written in the context of a UML model. In the class diagram above, for example, there are some constraints:

One static constraint is that the number of employees in any company must always exceed 50. This can be expressed by the OCL expression as follows:

```
Context Company inv:
      Self.numberOfemployee>50
```

An alternative expression is:

```
Context c: Company inv:
      c.numberOfEmployees>50
```

Also, constraint can be associated with operations or methods to express dynamic constraints. In the UML metamodel, the general expression for specifying dynamic constraints is:

```
Context Typename::operationName(param1 : type1,...) : ReturnType
      Pre parameterOk: param1
      Post resultOk:  result
```

In the expression above, the name `parameterOk` and `resultOk` are attributes of the metaclass Constraints inherited from ModelElement. In the diagram 2.10, for example, one constraint for operation `income (date)` is expressed as follows:

```
Context Person:: income (d: Date) :Integer
      Post : result = age * 10000
```

Types and Properties in OCL

In OCL, like any other typed languages, a number of basic types are predefined, including `Boolean, Integer, Real, String`. These types are available to the modelers all the time and independent of any specific user model. In addition, all classifiers from the UML model, such as `classes`, `interfaces`, are all types in OCL.

In OCL, attributes, association-ends, and side-effect-free methods and operations are all referred as properties.

For example, in figure 2.10, in the context of company, we can write constraint "The number of employees of a company must not be zero" as:

```
Context company
      Inv: self. employee->notEmpty
```

The expression above shows Association End `employee` is used as a property.

Collections

In OCL, Collection is a predefined abstract type. It includes three concrete types: Set, Sequence, and Bag. A Set contains no duplicate unordered elements, and a Bag may contain duplicate elements, and elements in a Sequence are ordered. Below are some examples:

```
      Set {1,5,10,6}
      Bag {3,3,5,5,8 }
      Sequence { "ape", "nut" }
```

Collections have a set of operations. One important operation used often in this thesis is `Select` and `ForAll`. One form of `Select` is:

```
      Collection->select (boolean-expression)
```

The result of a Selection is a sub set of a collection. For example, in figure 2.10, the following OCL expression specifies that a collection of all the employees older than 50 is not empty:

```
      Context Company inv:
      Self.employee->select(age>50)->notEmpty
```

One form of ForAll operation is:

```
      Collection->forAll(v:type|boolean-expression-with-v}
```

For example, in the context of a company of figure 2.10, the following OCL expression specifies the employees who have the first name "Jack":

```
       Context Company inv:
      Self.employee->forAll(forename = 'Jack')
```

CHAPTER 3 The UML representation for relational databases

3.1 Introduction to relational model

The relational model, introduced by Codd [6,7], is by far the dominant model of modern database technology. The relational model is concerned with three aspects: data structure, data integrity, and data manipulation [3].

A definition of relation

Date [3] gives a definition of relation as follows:

Given a collection of domains D_1, D_2, ..., D_n, a relation consists of two parts, a heading and a body. The heading consists of a fixed set of {attribute-name: domain-name} pairs, denoted as follows:

$\{<A_1: D_1>, <A_2: D_2>, ..., <A_n: D_n>\}$

The body consists of a set of tuples. Each tuple consists of a set of <attribute: attribute-value> pairs, denoted as follows:

$\{<A_1: V_1>, <A_2: V_2>, ... ,<A_n: V_n>\}$

V_i is a value from domain D_i and A_i is an attribute name.

This definition denotes the data structure of relational model. From the definition, we can conclude the properties of relations as follows:

- There are no duplicate tuples in a relation.
- Tuples in a relation are unordered.
- All attribute values are atomic

One informal representation for a relation schema is as follows:

- `TableName (attribute₁,attribute₂,…,attributeₙ)`

Relational data integrity

Candidate keys

The data integrity of the relational model is subject to a very large number of integrity rules. Among these rules, the concept of candidate key and foreign key, further clarified by Codd [6] , are the general integrity features. Date [3] gives the definition of Candidate key as follows :

Let R be a relation, a candidate key, say K, for R is a subset of the set of attributes of R, such that no two distinct tuples of R have the same value for K, and no proper subset of K has the uniqueness property.

For a given relation with more than one candidate key, the designated candidate key for that relation is primary key.

Foreign keys

Let R_1.R_2 be two relations, a foreign key FK in R_2 is a subset of the set of attributes of R_2, such that there exist a candidate key in R_1 and each value of FK in R_2 must have the identical value in R_1. The referential integrity is that the database must not contain any unmatched foreign key values.

Other data integrity constraints

Date [3] summarizes database-specific integrity rules, or business rules, as follows:

- Domain rule: it specifies the legal value for a given domain.
- Attribute rules: it specifies the legal value for a given attribute.
- Relation rules: it specifies the legal values for a given relation. The rule only refers to the given relation without referencing any other domains and relations.
- Database rule: it specifies the legal values for a given database. Database rule inter relates two or more distinct relations.

Functional dependency

Armstrong [32] first formalizes the theory of Functional Dependency (FDs). He states that a functional dependency (FD) is a many-to-one relationship between two sets of attributes within a given relation. One definition for FD is given below [32]:

Given a relation R, A and B are subset of the set of attributes of R, the function dependency FD $A \rightarrow B$ holds if and only if each value in A has associated with exactly one value in B.

From the definition above, we can see that any candidate key in a relation R functionally determines all attributes in relations R.

Further, Armstrong [32] gives three inference rules to computer all FDs from a given set FD (the rule more usually called Armstrong's axioms). They are listed as follows:

Let A, B, C be arbitrary subsets of the set of attributes of the given relation R, then:

1. Reflexivity: if B is a subset of A, then A→B.
2. Augmentation: if A→B, then AC→BC.
3. Transitivity: if A→B and B→C, then A→C.

Multivalued-dependency

Fagin [28] first gives a sound theoretical introduction to the notion of Multivalued dependency (MVD). He states that MVD is of a new kind of dependency and a generalization of Functional dependency (FD). The following table illustrates the concepts:

Employee	*Child*	*Salary*	*Year*
Hilbert	Hubert	$35k	1976
Hilbert	Hubert	$40k	1977
Gauss	Tom	$40k	1975
Gauss	Tom	$50k	1976
Gauss	Greta	$40k	1975
Gauss	Greta	$50k	1976
Pythagorus	Peter	$15k	1975

Table 2: An example of multivalued dependency

In the table above, relation *Employee (Employee, Child, Salary, Year)* conforms to BCNF [6]. The primary key for relation Employee consists of all its attributes, with no FDs in the relation. However, the relation Employee involves a good deal of redundancy. In the table, many attribute values are stored redundantly, name Hubert, for example, is stored twice in the table above. In addition, this relation leads to certain update anomalies. For example, to insert a new row for Employee Hilbet with a new child, say Jim, it is necessary to create two new tuples, one is (Hilbert, Jim, $35k, 1976), another tuples is

(Hilbert, Jim, \$40k, 1977). Similarly, if you want to delete a salary entry for an employee, you have to delete all tuples for the employee with that salary entry. In the table above, deletion of salary \$50k for Employee Gauss involves two tuples, one is (Gauss, Tom, \$50k, 1976), another tuple is (Gauss, Greta, \$50k, 1976).

To address the anomalies above, Fagin's [26] definition of Multivalued dependence is as follows:

Let R be a relation, and A,B,C be arbitrary subsets of the set of attributes of R, B is multidependent on A ($A \rightarrow\rightarrow B$) if and only if the set of values of B depends only on the set of values of A and independent of the set of values of C.

In addition, Fagin [26] gives two other different views of MVDs. Each view stands for a definition for MVD.

1. In relation R (A, B, C), $A \rightarrow\rightarrow B|C$ holds if and only if R equals to the join of its projection R1 (A, C) and R2 (A, B).
2. $A \rightarrow\rightarrow B$ holds for R (A, B, C) if and only if, whenever (x, y, z) and (x, y', z') are tuples of R, then so are tuples (x, y, z') and (x, y', z).

In relation R (A, B, C) , the value set of B is only dependent on the value set of A and independent of the value set of C. Similarly, the value set of C is only dependent on the value set of A and independent of the value set of B.

3.2 Mapping relational schemas to the UML classes

The relational model is concerned with three aspects: data structure, data integrity, and data manipulation [3]. In the UML, a class can naturally represent a relational schema, while an object of a class represent a tuple in a relation. In the UML metamodel level, we use stereotyped class «Relation» to represent a relation schema.

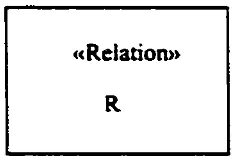


Figure 3.1 A stereotyped class representing a relation schema

Properties of a relation	**Properties of a class**
no duplicate tuples	*no identical objects*
tuples are unordered	*Objects in a class are unordered*
all attributes are atomic	*Attributes can be any data type*

Table 3: Comparisons between relations and classes

From the table above, we can see that an attribute of a class can be of any data type, and it can be arbitrarily complex, while attributes of a relation can only be atomic. However, to model relational schemas, we can restrict the attributes of classes stereotyped as «Relation». Here, the definition of **cardinality** and **degree** of relations can be expressed in the UML metamodel as follows:

Degree: Given a relation R , the number of attributes for R is called its degree. The OCL definition of cardinality for class R is:

```
self.allFeatures.select (f|f.name ="attribute")→size
```

Cardinality: Given a relation R, the number of tuples for R is called its cardinality. The OCL definition of degree for class R is:

```
Self.allInstances.collection→size
```

Domain vs. Data types

Date [3] states that a domain is a named set of scalar values (a scalar value represents an atomic value). A domain provides a pool of values. Further, Mr. Date states that a domain is a **data type**. SQL 92 standard defines a set of primitive data types. In comparison, the UML metamodel defines a set of data types.

ANSI SQL 92 data type	***The UML data type***
CHARACTER(n) OR CHAR(n)	String
VCHAR	String
NUMBER(P,S) OR DECIMAL(P,S)	Float
INTEGER OT INT	Integer
SMALLINT	Integer
FLOAT(P)	Float
REAL	Float
DATE	Stereotype «type»

Table 4: ANSI SQL 92 data type and UML data type

From the table above, we can see that there is no matching data type in the UML for Date in SQL. So, to facilitate model relation schemas, we can define a new data type `Date` in the UML.

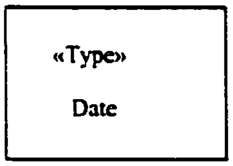


Figure 3.2 Date type

3.3 Representing Integrity Constrains in the UML

3.3.1 Primary keys

In the UML, we can use tagged values to add additional properties for attributes. We also can use stereotype «PK» to classify the attributes as primary key, stereotype «FK» to classify attributes as foreign keys. However, since any model element can use at most one stereotype, and in some cases, an attribute can server as a part of primary key as well as a part of foreign key in a relation, using stereotype is not a choice here. So, we attach tagged value {PK} to an attribute to state the attribute is a part of primary key, tagged value {FK} to an attribute to denote that the attribute is part of foreign key, tagged value {Unique} to an attribute to denote that the attributes is a candidate key.

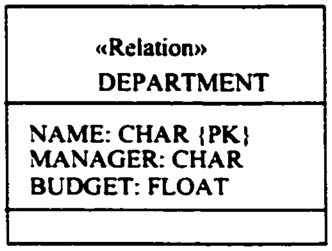


Figure 3.3 An example of UML representation for a relation

As for composite keys or composite foreign keys, we give all the attributes tagged value {PK} or {FK} respectively. For a relation R (class R), the definition of primary key by OCL in UML metamodel is as follows:

```
    PrimaryKey=Self.allFeature→collection(f|f.tagged="PK")
    self.instance.forAll(i,j |i.PrimaryKey<>j.PrimaryKey)
```

Similarly, we can give the OCL definition of Candidate keys for relation R as follows:

```
CandidateKey=Self.allFeature→collection(f|f.tagged="Unique")
self.instance.forAll(i,j |i.CandidateKey<>j.CandidateKey)
```

3.3.2 Inclusion dependency

An Inclusion dependency refers to a functional dependency between two sets of attributes [15]. These two set of attributes can reside in the same relation or different relations. A special case for Inclusion dependency is foreign key dependency, in which one set attributes is the key of a relation schema.

In the UML metamodel, more formally, Let R_1 and R_2 be two classes stereotyped as «Relation», and X, Y be subset of attributes of R_1, R_2 respectively, the semantic of inclusion dependency $R_1 (X) \subseteq R_2 (Y)$ represented in OCL is as follows:

```
R1.allInstances.forAll (i:R1|R2.allInstances→ collection( j:
R2|i.X = j.Y).size =1)
```

Inclusion dependency $R_1 (X) \subseteq R_2 (Y)$ can be expressed by UML diagram as follows:

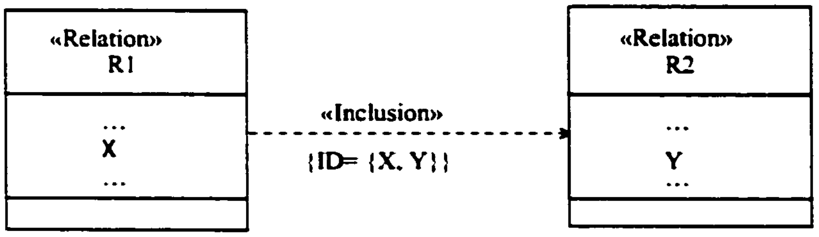


Figure 3.4: Inclusion dependency

In the figure above, a stereotyped dependency and tagged value pair are used to denote the inclusion dependency $R_1(X) \subseteq R_2(Y)$.

An inclusion dependency can be key-based or non key-based. For a inclusion dependency, $R_1(X) \subseteq R_2(Y)$, if Y is a candidate key of R_2, this inclusion dependency is key based. Furthermore, if Y is the primary key of R_2, the inclusion dependency is a foreign key dependency.

Key-based inclusion dependency

Given two relation schema R_1 and R_2, if $R_1.X \subseteq R_2.Y$, X is a foreign key of R_1 and Y is a candidate key of R_2, we define it as key-based inclusion dependency [19], rendered as $R_1.X \rightarrow R_2.Y$. In the UML, a key-based inclusion dependency is stereotyped as "Foreign key" or "FK".

More formally, Let R_1 and R_2 be two relation schemas, X is one foreign key attribute of R_1, Y is a candidate key of R_2. A key-based dependency $R_1.X \rightarrow R_2.Y$ can be formally defined by OCL as follows:

```
X = R1.allFeatures→collect(f| f.tagged="FK")
Y = R2.allFeatures→collect(f| f.tagged="PK")
R1.allInstances.forAll(i:R1|R2.allInstances→collection(
j:R2|i.X=j.Y).size=1)
```

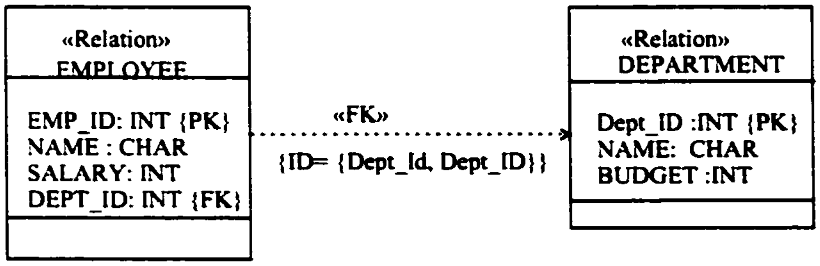


Figure 3.5: An example for key-based inclusion dependency

3.3.3 Basic types for key-based Inclusion dependency

In the following part, since all non key based dependencies can be transformed into key-based dependency [19], we only deal with key-based inclusion dependencies (foreign key dependency). According to the types of left side and right side of a key-based Inclusion dependency, we can further classify them into different categories. This will facilitate future database reverse engineering.

- *Non key- to- key Dependency*

This type of dependency is represented as non-key → key. It refers to that, given two relation schema R_1, R_2, for dependency $R_1.X \rightarrow R_2.Y$, X is a foreign key set of R_1 and Y is the key attribute set of R_2. In the UML, a key-based dependency is stereotyped as "Foreign key" or "FK".

One example is showed as figure 3.6.

- *Partial key- to- key dependency*

This type of dependency is represented as *partial key→ key.* In this type of dependency, the left side is part of the key attributes of a relationship, and the right side is the key attributes of a relation schema.

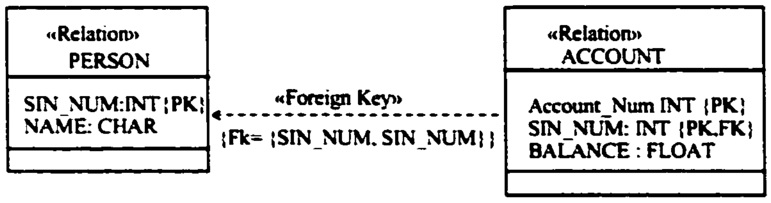


Figure 3.6: An example for partial key- to- key dependency

In the figure above, the primary key of relation ACCOUNT consists of two attributes: Account_Num and SIN_NUM, but attribute SIN_NUM also is a foreign key.

- *Key- to- key dependency*

This type of dependency is represented as *key → key.* Both sides of the dependency are key attributes. More formally, Let R_1, R_2 be two relation schemas, X is the key attribute set of R_1, Y is the key attributes set of R_2. $R_1.X \rightarrow R_2.Y$ can formally defined by OCL as follows:

```
      X = R1.allFeatures → collect(f| f.tagged="PK")
     Y = R2.allFeatures→collect(f| f.tagged="PK")
  R1.allInstances.forAll (i:R1 | R2.allInstances→ collection(
   j: R2|i.X = j.Y).size =1)
```

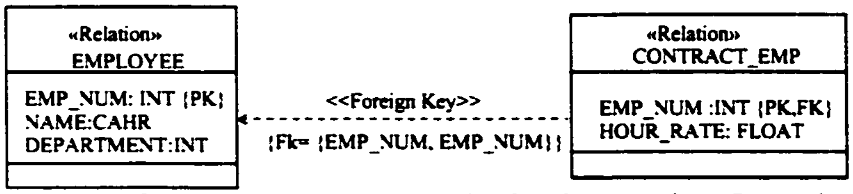


Figure 3.7: An example for key-to-key Dependency

- *Identifying dependency*

In addition to the categories for foreign key dependencies defined above, according to its semantics, a foreign key dependency can either be an identifying dependency or a non-identifying dependency. An identifying dependency corresponds to a weak entity [19].

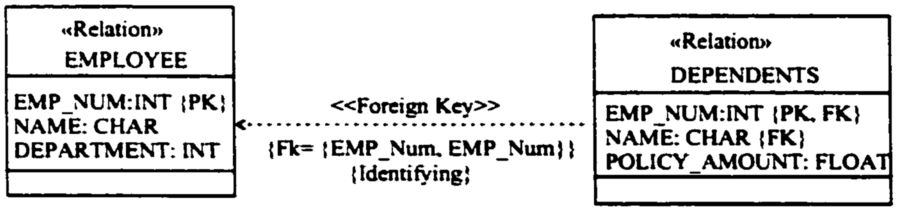


Figure 3.8: An example for identifying Dependency

An identifying dependency is a dependency between two dependent tables, where the child table cannot exist without the parent table. All of the primary keys of the parent table become both primary and foreign key columns in the child table. In figure 3.9, an employee can purchase insurance policy to cover their dependents. If an employee is eliminated from table Employee, the correspondent records on Dependents table should be eliminated as well. The identifying relationship is represented as a tagged value {identifying} on the dependency.

More formally, Let R_1 and R_2 be two relation schemas, X is a foreign key of R_1, Y is a candidate key of R_2. The identifying foreign key dependency can formally defined by OCL as follows:

```
    X = R1.allFeatures→collect(f| f.tagged="FK")
    Y = R2.allFeatures→collect(f| f.tagged="PK")
  R1.allInstances.forAll (i:R1 | R2.allInstances→collection(
     j: R2 | i.X = j.Y).size =1)
  R2.allInstances.forAll (i:R2 | R1.allInstances→Exist(
     j: R1 | i.Y = j.X))
```

In the UML, we can also use composition to represent identifying relationship between two relations showed as follows:

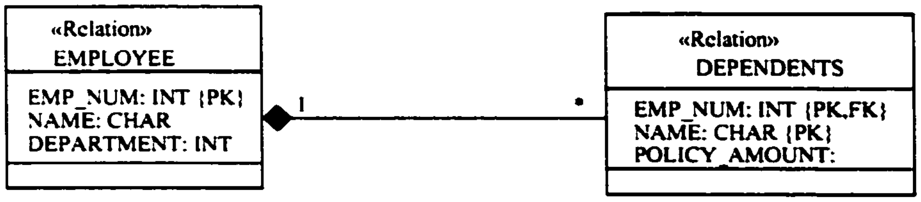


Figure 3.9: An example for identifying Dependency

3.3.4 Representing other integrity constraints in the UML

In relational model, there are domain constraints, relation constraints, database constraints [3]. In the UML, OCL can express these integrity constraints elegantly without ambiguity. In supplier and part database, for example:

```
Supplier (S#, SNAME, STATUS, CITY)
Part (P#, PNAME, COLOR, WEIGHT, CITY)
SP (S#, P#, QTY)
```

One domain rule is "the quantity number must large than zero and less than 500". This can be expressed by OCL expression as follows:

```
Context SP inv:
    Self.quantity<=500 and self.quantity>=0
```

One example of relation constraint is " suppliers in London must have status 20". The OCL expression is:

```
Context supplier inv:
 self.allInstances.forAll(k|k.CITY="LONDON" implies
                    k.STATUS=20)
```

To express the constraints in UML diagrams, one straightforward way is to attach notes to classes.

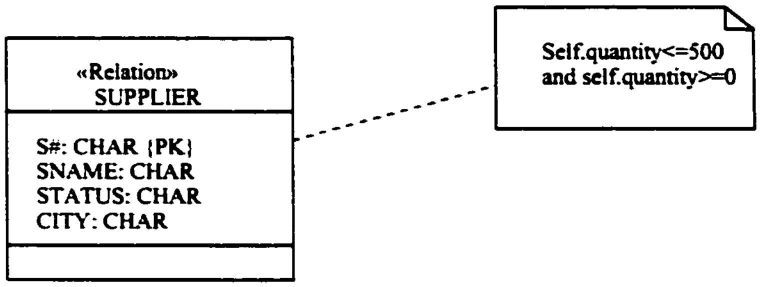


Figure 3.10 : Expressing integrity as notes for class Supplier

However, this method has some drawbacks. For a class diagram, there may have many notes, some are general information in natural language, and some are for integrity constraints in OCL expression. Thus, a code generator must distinguish these notes and other general notes accordingly. In addition, adding many notes for a diagram makes it unclear and cumbersome.

In data modeling, all the integrity constraints, including key constraints, domain constraints, database constraints, are static properties for a database. In the UML, these properties can be expressed as stereotyped operations, so code generators can interpret these operations in both forward and reverse engineering way.

To model constraints, we can define two set of stereotyped operations from two different point of views. One set of operations use stereotypes, like «check», «trigger». These operations directly link to logic database designs.

For example,

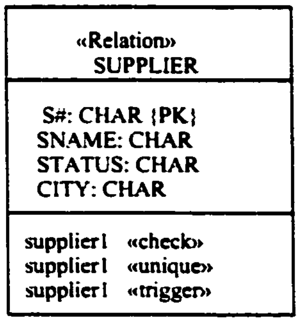


Figure 3.11: An example for stereotyped operations

«check» supplier1: this operation defines domain constraint that can be implemented by "check" condition of SQL, for example, "the status of supplier must large than 10" is one of this kind.

«trigger» supplier3: this operations defines table constraints , which can be implemented by triggers. One example is "no supplier with status less than 20 can supply any part in a quantity greater than 50".

Another set of stereotyped operations use «domain rule», «relation rule», «database rule». The semantic of these stereotypes conform to each constraint respectively. This method only give the semantics of constrains in the diagram, and let the code generator to implementation the constraints, thus providing a more flexible solution.

3.4 Representing relational dependencies in the UML

Since functional dependencies describes relationships between attributes within a relation, we need to model a subset attributes in the UML. Here, we introduce a stereotype «Property» for class. The stereotype restricts the class to contain only attributes.

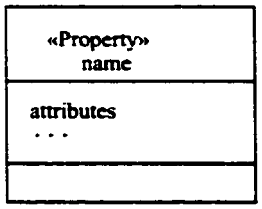


Figure 3.12: A diagram representing attributes

In UML metamodel, this constraint is expressed by OCL as follows:

```
Self.allFeatures .forAll (f|f.oclIskindOf (Attribute))
```

In the UML, we can use a many-to-one association to represent a functional dependency. The following is functional dependency diagram for relation supplier in UML.

For example, a relation `supplier (S#, sname, City)` has functional dependencies: `S#→sname` and `S#→city`. These two functional dependencies are represented by the figure 3.1.

For any relation with a set of functional dependencies, we can use a stereotyped class diagram to represent functional dependencies. By this way, the dependency constrains in a relation is preserved , and can be transformed, interpreted by tool generators.

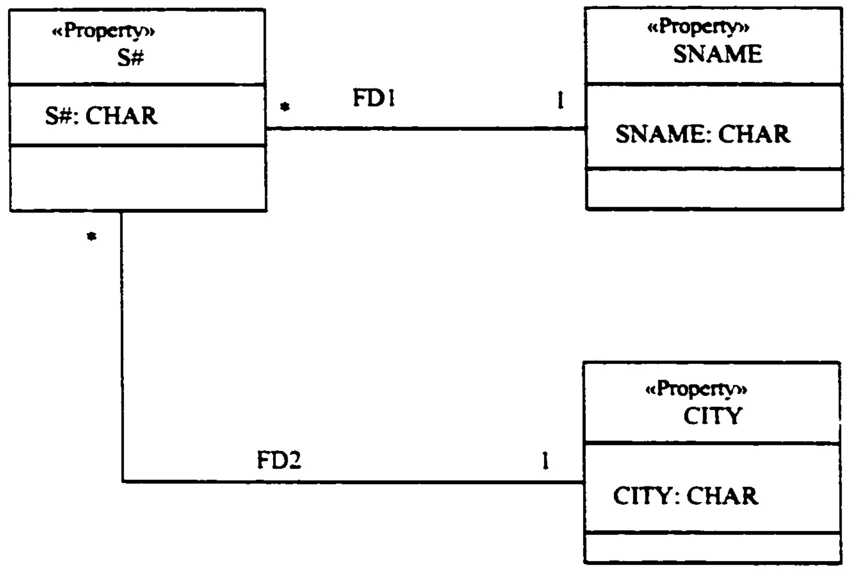


Figure 3.13: A class diagram representing functional dependencies

3.5 Classification of relational databases

Since all relational schemas are evolved through certain design phases, and are transformed from common semantic models, like the ER model. The semantic enriched relationship in the ER model, such as many-to-one, many-to-may, IS-A relationship, etc., are all mapped into flat inclusion dependencies in relational model. So, it is natural to analysis the basic type of referential diagrams [3] in relational database. This will facilitate Database Reverse Engineering (DBRE) as well as the UML representation for relational databases.

Degree of Key-based Inclusion dependencies for a relation schema

For a relation schema, it may have a number of foreign keys. Each foreign key can be a single attribute or composite attribute. The degree of foreign key for a relation schema is important for relation detecting. Since after relation splitting, all the inclusion dependencies are key based [19].

A relation schema may have several foreign keys, and there may exist relational schemas, which are foreign-key dependent on it. To distinguish various types of relational schemas, we introduced two concepts: D_{out} and D_{in}. Given a relation schema R, the degree of foreign keys in which R serves as client side is defined as $D_{out}(R)$. The number of inclusion dependencies in which a schema serves as key side is defined as $D_{in}(R)$. More formally, in the UML metamodel level, for a class R, these two concepts defined by OCL are as follows:

```
allDependencies=self.allContents→Union(c|c.oclIskindOf(
    Dependency))

Dout(R)=self.allDependencies→select (d|d.client=self)→size
Din(R)=self.allDependencies→select (d|d.supplier=self)→size
```

In section 3.3.3, we classify the inclusion dependencies into several types. For a whole database, we use concept D_{out} and D_{in} to classify relational schemas into different types.

Then, for the grouped schemas, we further classify them by the types of inclusion dependencies. So, we use a divide-and -conquer method in DBRE.

Binary type

This type of referential diagram consists of two schemas, with one schema foreign key dependent on the other schema. This is the most basic relationship, and it roots in binary relationship in conceptual model. It can be further classified by the type of inclusion dependency as stated in 3.3.3. Later we will see the mappings based on this types. It is illustrated in figure 3.15.

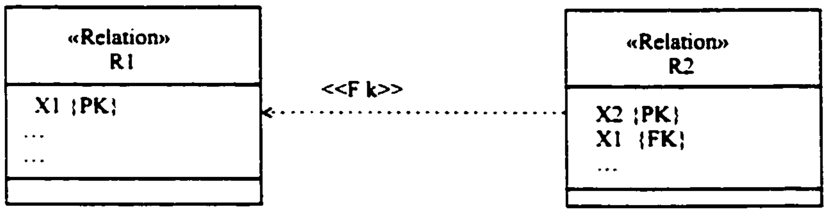


Figure 3.14: Binary type

To make more formally without ambiguity, Let R_1,R_2 be two relation schemas, X_1 is the key attribute set of R_1. Y_1 is the foreign key attribute set of R_2, if $R_2.Y \rightarrow R_1.X$ and $Dout(R_2) = 1$ and $Din(R_2) = 0$, we say the relationship of this two schemas is in binary type.

Star type

This type of referential diagram consists of several schemas, with one schema foreign key dependent on all other schemas. Schemas in this type may have various relationships in conceptual model, depending on whether the schema with foreign keys has its own designated primary key or not. The star type is illustrated as figure 3.16. In the figure, relation R has a number of foreign keys, each of them pointing to a schema. The relationship among these schemas can be treated as many separate binary relationships or as a whole unit.

More formally, Let R, R_1, R_2,..., R_n be relation schemas, X_1 is the key of R_1, Y_i ($1 \leq i \leq n$) is one of foreign keys of R. if $R.Y \rightarrow R_1.X_1$ and $R.Y_2 \rightarrow R_2.X_2$,...,$R.Y_n \rightarrow R_n.X_n$, $D_{out}(R) = n$ and $D_{in}(R) = 0$, we say the relationship among these schemas is in star type.

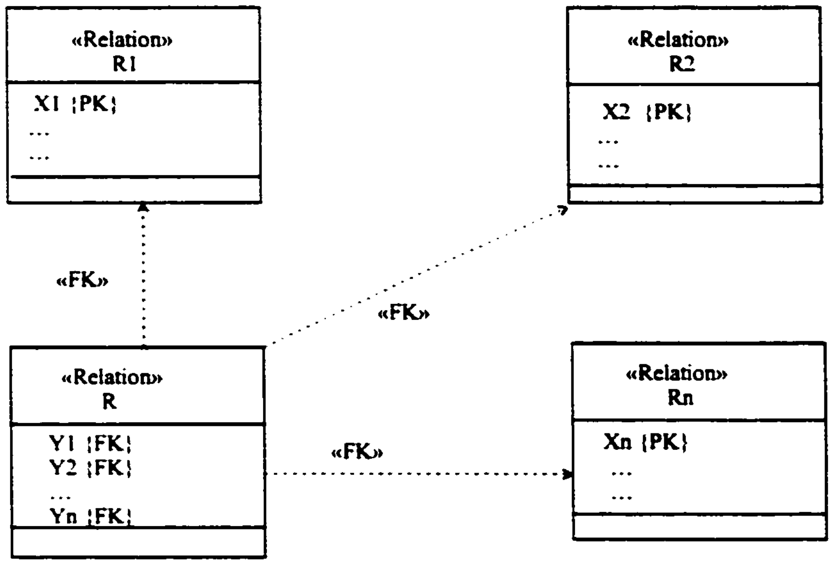


Figure 3.15: Star type

Snow flake type

This type of referential diagram consists of several schemas, with each schema foreign key dependent on another schema. In figure 3.17, R_1 is foreign key dependent on R_2, R_2 is foreign key dependent on R_3 . and so on. This type obvious contains several binary relationships. But, it may roots in schema normalization based on function dependencies in a schema. So, it worth to stand out as a special type.

More formally. Let R_1, R_2...R_n be relation schemas, X_1 is the key attribute set of R_1, Y_1 is the foreign key attribute set of R_1, $R_1.Y_1 \rightarrow R_2.X_2$, $R_2.Y_2 \rightarrow R_3.X_3$ ... and $R_{n-1}.Y_{n-1} \rightarrow R_n.X_n$, $D_{out}(R_i) = 1$ and $D_{in}(R_i) = 0$, we say the relationship among these schemas is in snow flake type.

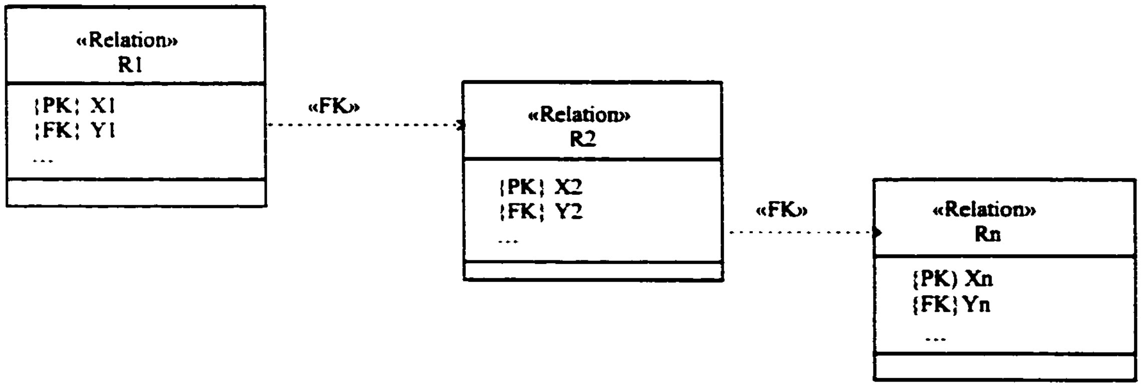
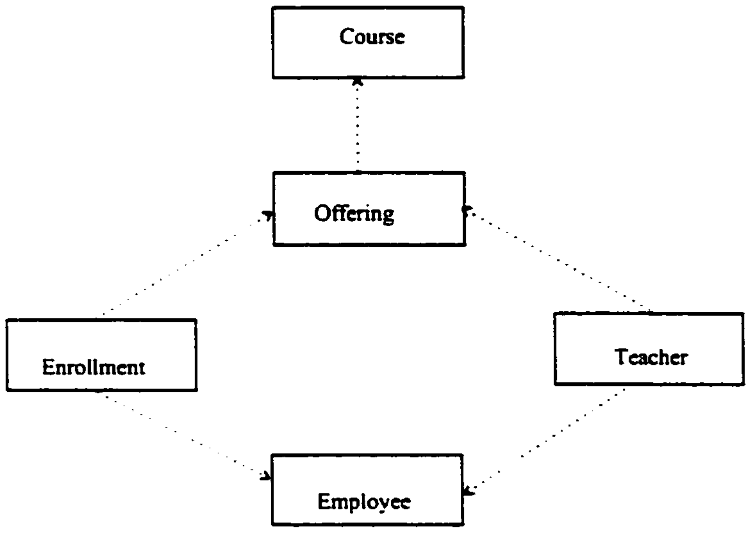


Figure 3.16: Snowflake type

Hybrid type

This type of referential diagram consists several schemas, with some schemas foreign key dependent on other schemas and these schemas also have foreign key dependencies pointing to them.. This type of schema consists of basic type mentioned above. Hybrid type is illustrated as figure 3.18. In the figure, three schemas `Teacher`, `Offering`,



`Employee`, are in star type. Also, the three schemas -`Offering`, `Employee`, `Enrollment`, are in star type.

Figure 3.17: Hybrid type

CHAPTER4 Applying the UML in Database Reverse Engineering

4.1 Introduction to database Reverse engineering

Software reverse engineering deals with the problem of comprehending an existing system and recovering corresponding design specifications [14]. As one branch of software reverse engineering, database reverse engineering (DBRE) has gained wide interests since mid 1980s. A common motivation for performing reverse engineering is that the original design specifications and design process were not well documented. The application of DBRE technology has many advantages, such as facilitate maintenance and redesign, help the integration of databases. Also, in case of converting an existing database into an object oriented database, or integrating several databases, it is essential to obtain a conceptual model from existing databases.

Many papers has been published to address DBRE approaches. They differ in input requirements and output, and apply to different situations [14]. Chiang's [24,25] method requires the database instance, relational schemas, primary keys and part of inclusion dependencies as input; Markowitz et al.'s [30] method needs key dependencies and key-based inclusion dependencies; Navathe's [2] method requires the relational schemas in 3NF or BCNF. The output of these method is the ER or EER model. Petit et al's [13] method requires the relation schemas, data instances and code as input. All the output of these methods is the ER or EER model.

Chiang [24] defines database reverse engineering as the process of examining an existing database system to identify the database's contents and their interrelationships, and recover information about the application domain semantics. Through database reverse engineering, the flat database schemas are mapped into a semantically enriched data model, which is closer to the application domain and easy to understand. He also provides an abstract algorithm for schema translation.

With different input requirements, Paul [19] gives a detailed algorithm for all the three steps. However, since the reverse engineering needs some extent of human interactions. The mapping may take different results depends on different situations. Also, his method does not consider the evolution of functional dependencies in relational schemas. His approach is generally as follows:

1.Candidate key splitting. If a relation schema contains several keys, it may corresponds to more than one object type. However, whether such a schema should be mapped into different objects depends on the domain semantics, and should be decided by users.

2. Inclusion Dependency Splitting. This happens when foreign key dependency is not key based, that is, a foreign key does not point to a primary key. This step splits such a schema into a set of schemas so that all foreign keys point to the primary keys.

3. Folding transformation. This step removes all inclusion dependency cycles.

In comparison, Johanesson's [20] method is more related with this thesis. In his paper, he assumes that all relations schemas are in third normal form (3NF), and the input for his method requires keys, candidate keys and inclusion dependencies.

There is another work closely related to this thesis. Carellanos [1] gives a set of examples to map relational schemas into an object-oriented data model. In the paper, he also introduces some OO concepts, like Generalization, Cartesian aggregation, Interest Dependency etc. He did shed some light on this topic. However, he mapped relation schemas into an OO data model, not OO designs as proposed in this thesis, the purpose of his approach is to overcome heterogeneity between different database systems. In addition, his approach is not complete; the concepts he introduced are not defined.

Motivations and Advantages of mapping schemas into OO designs

Database applications involve two parts of work: application designs and database designs. Applications designs describes the business logic, GUI layers, and produce, in object oriented designs, class diagrams and interaction diagrams; Database designs describe the persistent part of applications and produce the data model representing the logic and physical implementation of a database. Traditionally, these two parts are separate. Database designs involve cooperation among business analysts, data modelers, users having domain knowledge. The most widely used tool for database design is the ER or EER related tools. The ER model has enveloped a lot since it first appearance in 1970s. The fundamental constructs in the ER model are entities, relationships and attributes. Today, there are many ER or EER representations and ER design tools available on the market.

Application designs involve software developers. The main trend in today's software development follows object oriented technology. In object oriented designs using UML tools, software developers use the logical class diagram to represent the main static view of applications, and use UML interaction diagrams to model the dynamic view of applications, etc. For the whole database application designs, database designs and application designs may be conducted separately. But eventually, software developers need to represent data models in the application designs. The area of interface and overlap between these two parts often give the most challenging aspect of database application development [23]. To integrate data model in object oriented designs, we need a high level understanding on database schemas. This can be represented by the ER model. However, if the application designs follow an object oriented methodology using the UML, it is not easy to integrate the ER model into the UML diagrams. In our approach, we map the flat database schemas into object oriented designs in the UML, and the mapped database schemas can be naturally integrated into object oriented application designs. For example, a database table can be modeled as a persistent class and integrated into class diagrams.

Input requirements for DBRE

Lurdes [15] provides a list of items for the input and the assumptions required by the DBRE method and he states that the current DBRE methods differ in these items listed as follows:

- Semantic Knowledge. This includes attributes semantics and attributes name consistency.
- Data and Code

- Candidate keys
- Foreign keys or key-based inclusion dependencies
- Non key functional dependencies (3NF)
- Non key-based inclusion dependencies
- Human input

In this thesis, since we deal with modern databases, some information for a database, such as candidate keys, foreign keys are assumed to be available (these information can be retrieved from the database catalog). Usually, data and code are also available for analysis. We assume database schemas are in third normal form (3NF) or BCNF.

4.2 Basic transformations

- **Inclusion dependency transformation**

There are two kinds of inclusion dependencies: non-key based and key based. The first step of schema transformation is to transform non-key based inclusion dependencies into key-based inclusion dependencies [19]. In modern databases, non-key based inclusion dependencies are not widely supported. In this thesis, we only briefly illustrate such transformation by the following example:

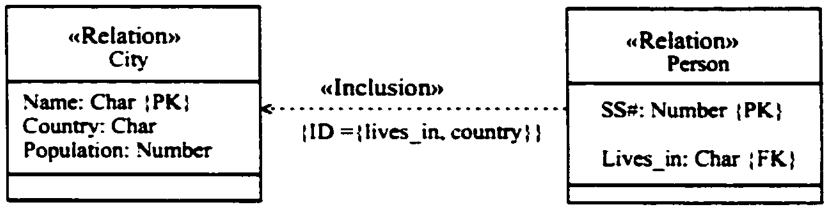


Figure 4.1: An example for inclusion dependency

In figure 4.1, foreign key attribute `LIVES_IN` of relation `PERSON` is foreign key dependent on non-key attribute `country` of relation `CITY`. To transform this non-key based inclusion dependency into key-based, we first map relation `City` into two objects: `CITY` and `COUNTRY`. The foreign key dependency is mapped into many-to-one association in the UML as showed in figure 4.2. The schemas in figure 4.1 is translated into the following OO model:

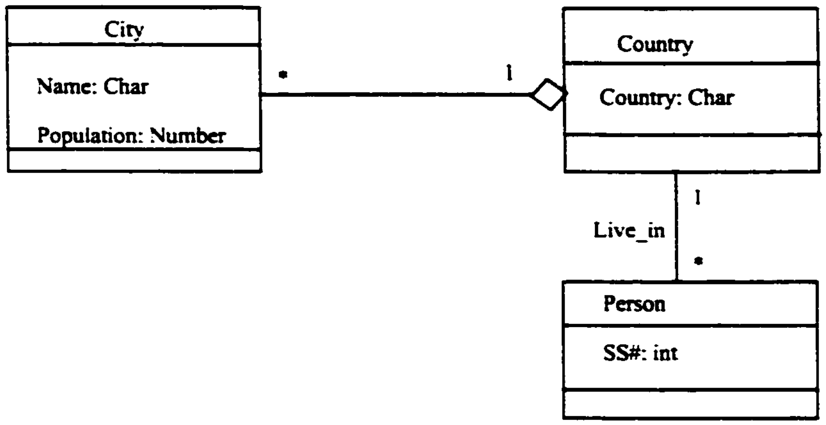


Figure 4.2: An OO mapping example

- **Candidate key splitting.**

For relations with more than one candidate keys, each candidate key may represent a separate object in real world. When mapping schemas with more than one candidate keys, it is basically up to the users to decide based on the domain semantics. Chiang [23] and Johannesson [19] have discussed candidate key splitting. In this thesis, we propose that candidate splitting is also influenced by the inclusion dependencies pointing to them.

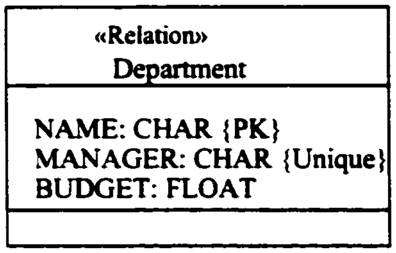


Figure 4.3: An example for candidate key splitting

In figure 4.3, relation DEPARTMENT has two keys, one is primary key Name, the other is candidate key MANAGER. This schema may be split into two schemas- DEPARTMENT and MAMNAGER with one-to-one association as showed by the following figure:

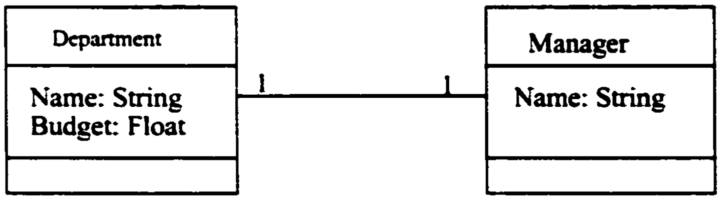


Figure 4.4: An OO mapping for candidate key splitting

In some cases, a candidate key does not mean distinct object type in real world. For example, relation STUDENT has two candidate keys, one is STUDENT_ID, the other is NAME. Candidate key name is an attribute of student, and it does not mean a different object type. However, if a candidate key servers as the key side of an inclusion dependency, it should be split accordingly. We give the mapping implications for candidate key splitting as follows:

Mapping implication: *Let R, R' be two relation schemas, X_1, X_2, ..., X_n be candidate keys of R individually, for any candidate key X_i, if $R'(Y) \subseteq R(X_i)$ holds, then split R to $R-R(X_i)$, $R(X_i)$.*

- **Null value**

For relation schemas, only non-key attributes are allowed to be null [3]. If an attribute is null-allowed, that means some objects has null value with this attribute. This intuitively maps to a special class in OO model. However, it is up to the users to make the decision according the domain semantics.

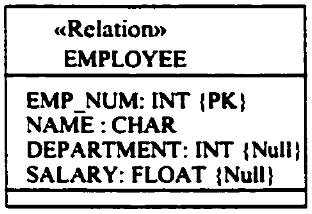


Figure 4.5: An example of null attributes

In figure 4.5, relation Employee has two null-allowed attributes: DEPARTMENT and SALARY. Based on the semantics of the domain, there may exist some privileged employees who do not belong to any department, and some special employees who may get salary from other companies. So, relation EMPLOYEE in the figure 4.5 may map to the following OO model:

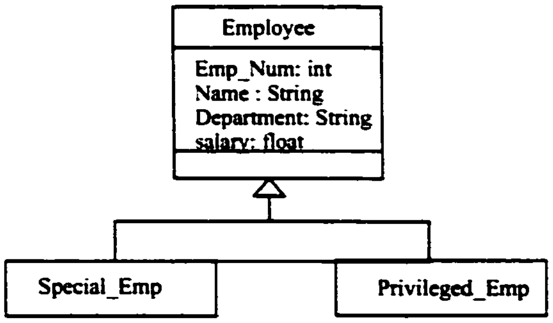


Figure 4.6: An example for null attributes to OO model

Generally, for relation schemas with null-allowed attributes, for each null-allowed attribute, the schema may have a distinct object type in OO model. Again, such transformation is finally decided by users. To put the transformation more formally, the mapping implication is given as follows:

Mapping implication: *Let R be a relation schema, A_1, A_2, be null-allowed attributes of R, in object oriented model, map R into R, R_1=R-R_1(A_1), R_2=R-R (A_2), while R_1, R_2 be subclasses of R..*

4.3 Mapping relational database schemas into object-oriented models

4.3.1 Mapping referential diagrams in binary type

For a relational database, at logic model level, what we see is a flat system of tables related by inclusion dependencies. When transforming a database into object oriented model, we adopt a divide-and-conquer method, that is, we divide the whole system into a number of subsystems according to the classifications in 3.5 (binary type, star type, etc.), then transform each subsystem accordingly.

As defined section 3.5, schemas in binary type involves two schemas, and can be further classified by its foreign key dependencies. To map this type of schemas into object oriented model, various domain semantics and different foreign key dependencies are needed.

Non key -to- key

As defined in 3.3.3, general dependency is non-key→key dependency. It roots in many-to-one relationship in the E-R model. However, when mapping it to OO model, the dependency may map to an aggregation or many-to-one association, depending on the domain semantics, and should be decided by the users. The example given below talks about only simple key attributes. In case of composite key (primary key consists of more that one attributes), the mapping is simply the same.

In figure 4.2, the two schemas have a non key -to-key dependency. Generally, the dependency is mapped to a many-to-one association. But, in this example, in OO model, the two objects has an aggregation relationship. The correspondent OO model for figure 4.2 is as follows:

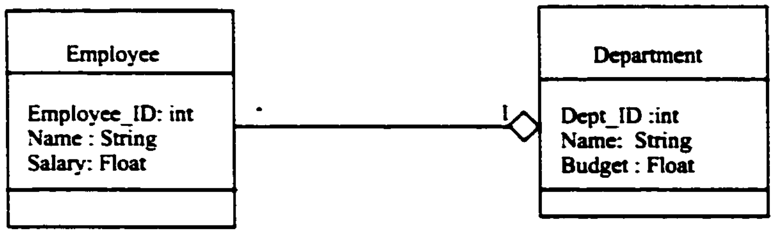


Figure4.7: An OO model for schemas in figure 4.2

In figure 4.8, schemas EMPLOYEE and EMP_TYPE has a general dependency. The schema is transformed into a plain many-to-one association in figure 4.9:

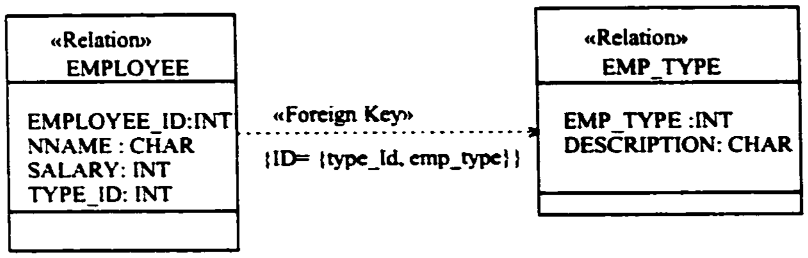


Figure 4.8: A general dependency example

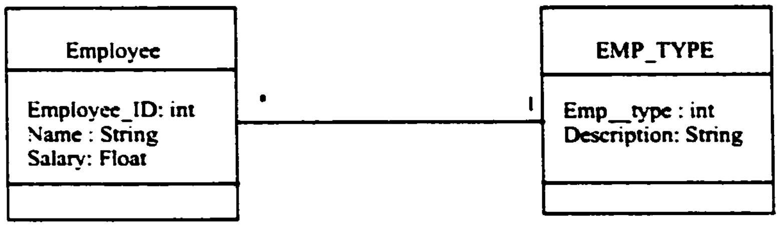


Figure 4.9:An OO model for a general dependency example

There is a special case: self-referencing [3]. For example,

EMP (EMP#, SALARY, MGR_EMP#)

In relation EMP. MGR_EMP# represents the employee number of the manager of the employee. and EMP# is the primary key. MGR_EMP#→EMP# holds for this relations. In OO model. relation EMP correspond to two classes:

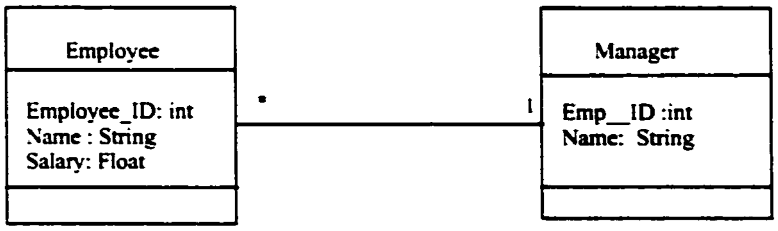


Figure 4.10:An OO model for unary dependency

Partial key -to -key

As defined in 3.3.3, another foreign key dependency is `partial key-key`. It roots in various relationships in semantic model. For identifying dependency in figure 3.9, it maps to a composition as showed below:

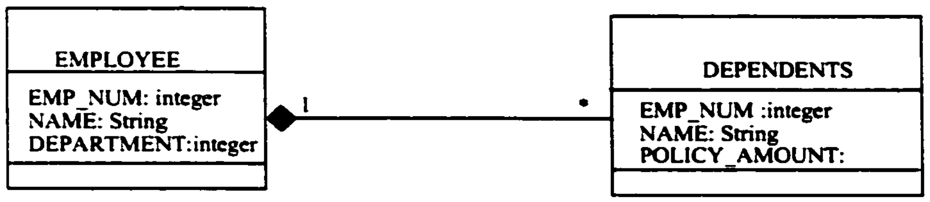


Figure 4.11:An OO model for figure 3.9

For an general partial key-to key dependency in figure 4.12, it maps to an generalization relationship in figure 4.13.

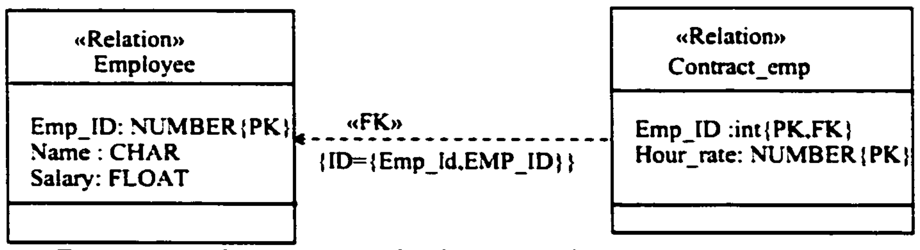


Figure 4.12:An example for Partial key- to- key dependency

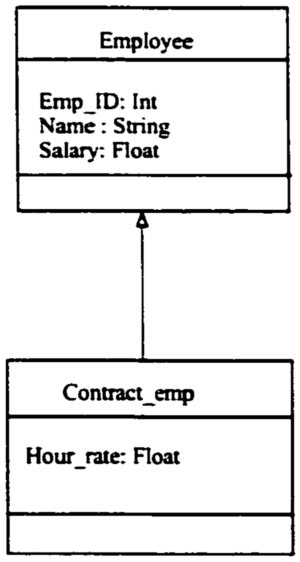


Figure 4.13: A generalization relationship

Key- to- key

This case means that key attributes in one relation also serve as foreign key.

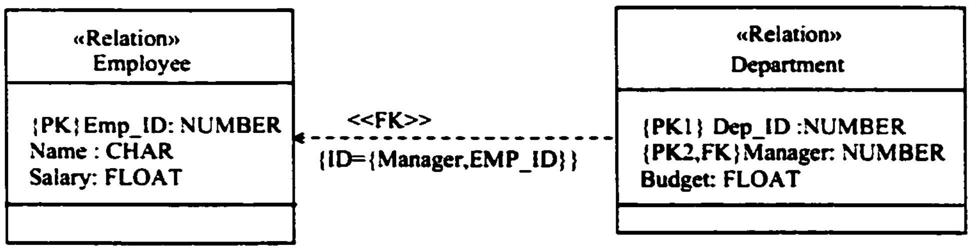


Figure 4.14: An example of key-to-key

In the example above, relation DEPARTMENT has two candidate keys, one is DEPT_ID and the other is MANAGER. Key MANAGER also is a foreign key pointing to key of relation EMPLOYEE. According to the mapping method, relation department should be split into two relations: DEPARTMENT and MANGER.

So, the OO model for the schemas above is:

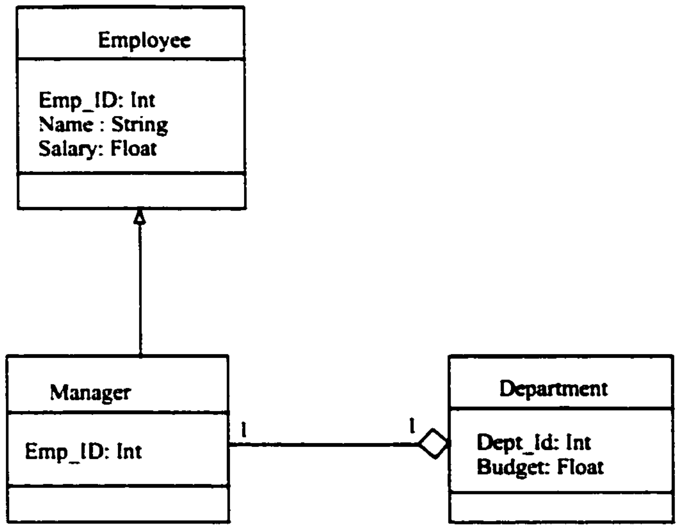


Figure 4.15: An OO model for a generalization relationship

We summarize the mapping rule for binary type as the following implications:

Mapping Implication *: For two schemas R1, R2 in binary type, that is* $D_{out}(R_1)=1$ *and* $D_{in}(R_2)=1$*, depending on the type of inclusion dependency, schemas has different OO mappings. If the inclusion dependencies in* R_1 *is the type of partial key- to -key, consider map to many-to-to one association or aggregation. If the inclusion dependencies in R1 is the type of key- to -key, consider map to generalization or composition. The choices are made by users according to the domain semantics.*

4.3.2 Mapping referential diagrams in Star Type

As defined in 3.3.3, schemas in Star type involves three or more schemas, with one schema foreign key dependent on other schemas. In other words, there is a relation R, it meets D_{out} (R)>=2. For schemas in Star type, depending on the type of foreign dependency, the schemas have different mappings.

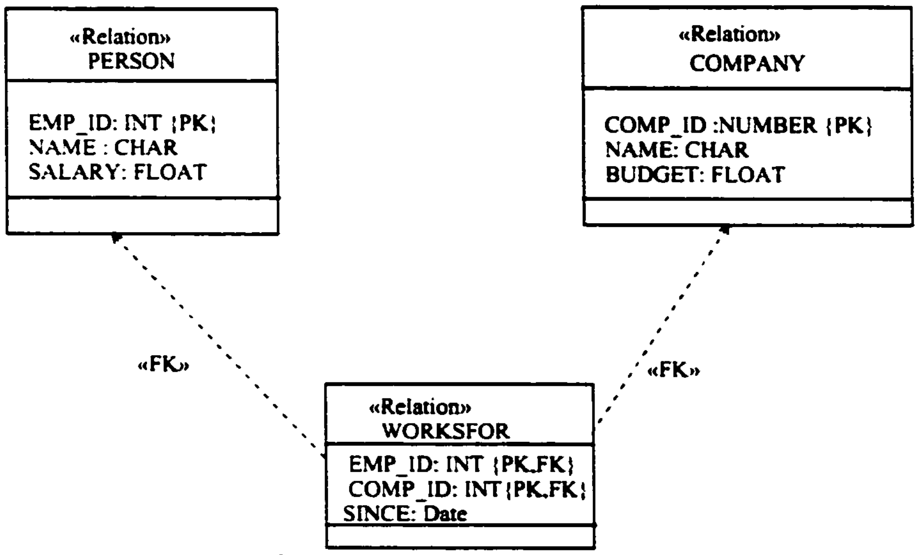


Figure4.16: A star type example

In figure 4.16, the three schemas is in Star type. Relation WORKSFOR has no designated primary key itself and has two foreign key attributes. The two inclusion dependencies is the type of `partial key -to key`. The primary key of relation WORKSFOR is composed by two foreign keys.

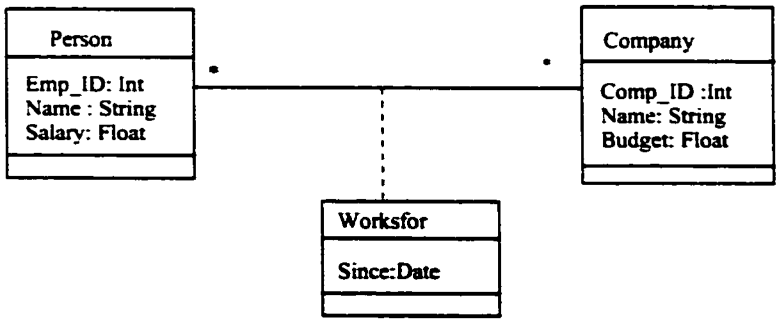


Figure 4.17: An OO model for star schema

So, relation WORKSFOR is not an object type and should maps to an association class. If, for example, relation WORKSFOR has no attribute SINCE, this relation only maps to a plain association.

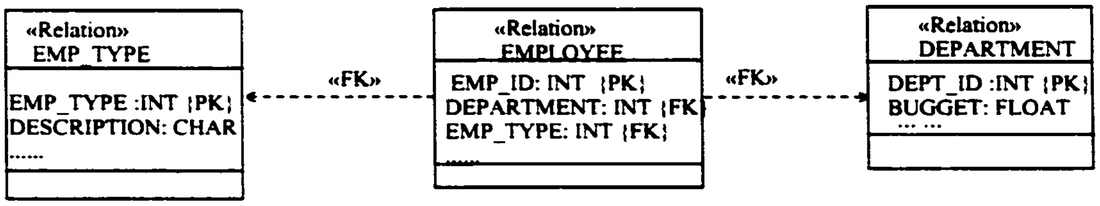


Figure 4.18: A star schema example

In case of the foreign dependencies are type of general dependency. These dependencies are treated separately, and each should maps to one association. In figure 4.18, relation EMPLOYEE has a desiganated primnary key and two foreign keys. In this example, from the domain knowledge , we can see that these two foreign key dependencies should be treated seperatedly, and relation EMPLOYEE is a distinct class. So, we map it to three classes with two associations as follows:

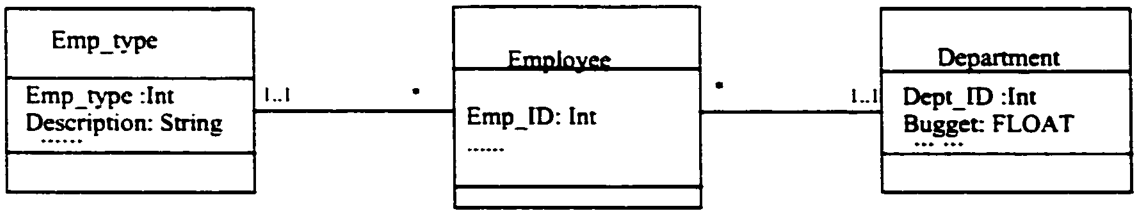


Figure 4.19: An OO model for star schema

A special case for this type schema is that: one schema has two `partial key -to-key` dependencies, and these two dependencies point to the same relation schema. In figure 4.20, relation Request has two foreign keys, and the two inclusion dependency is in type `partial key-to -key`.

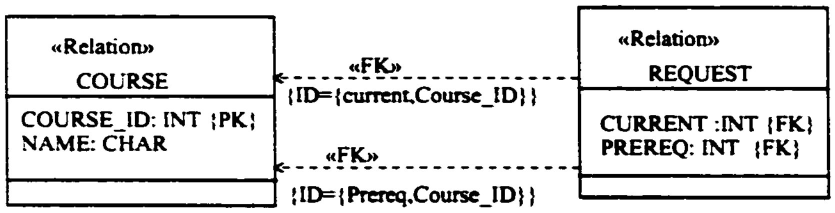


Figure 4.20: A recursive relationship

The relation schemas map to a recursive relationship of object type `Course` as showed in figure 4.21.

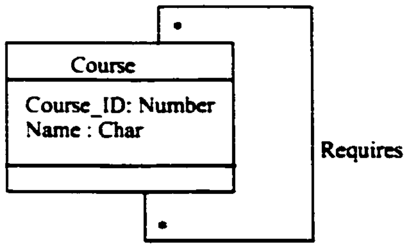


Figure 4.21: An OO model for recursive relationship

***Mapping Implication** : For schemas R_1,R_2,R_3 in star type, that is, there exist a schema R_i, $D_{out}(R_i)=2$ and inclusion dependency set I. If all inclusion dependencies in R_i are type of partial key- to- key, consider map R_i to many-to-many association . If R_i has a designated primary key and its inclusion dependencies are type of non key- to -key, map R_i to a separate object type in OO model.*

In case of schemas in Star type involving more than three schemas. That is, for total n schemas, there is a relation schema R_i, D_{out} (R_i) = n-1. Relations in this type may root in ternary relationship in semantic model [12]. The paper [12] summarizes ternary relationships in E-R model as: functional relationship, partial functional relational relationship, and general relationship. Each of these relationships converts to a different relation model.

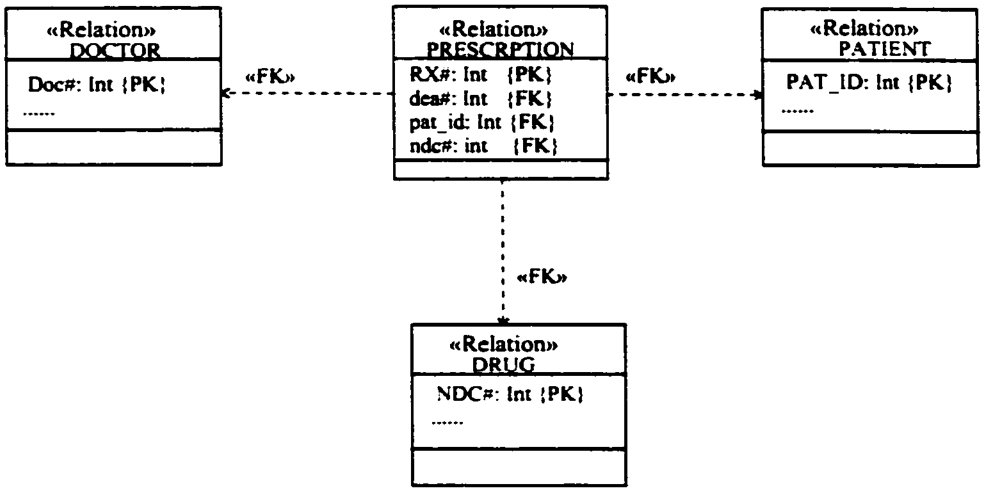


Figure 4.22: A ternary relationship

In the figure above, relation PRESCRIPTION has a designated primary key RX#, and three foreign key attributes. The foreign key dependencies of relation RESCRIPTION are all in type of non key-to-key. Since it has a designated primary key, it maps to a distinct object type in OO model. For the foreign key dependencies, they can be treated separately. So, it maps to the following OO models:

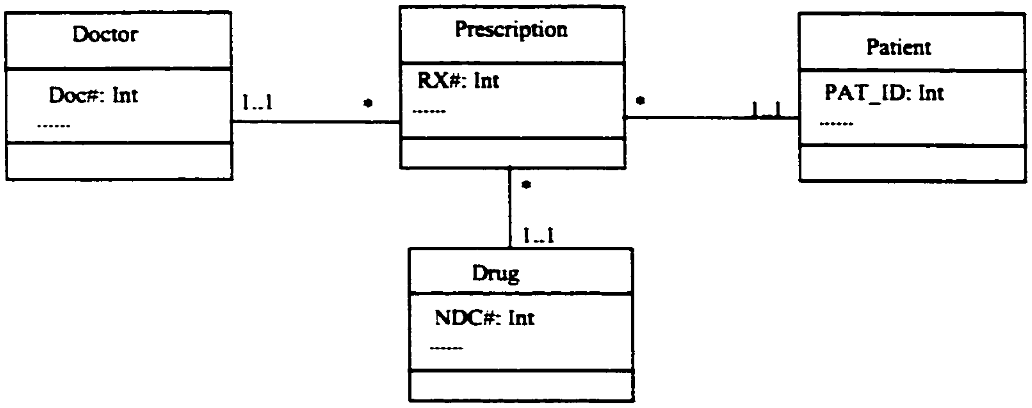


Figure 4.23: An OO model for ternary relationship

Let take another example, in figure 4.24, the key of relation `SUPPLIERS` consists of three foreign key attributes: `SUP#`, `PROJ#`, `PART#`. Each of these three attributes serves as a foreign key pointing to a different relation. Since the relation has no designated primary key itself, it is not a distinct object type in the real world. So, it maps to an ternary association as showed in 4.25:

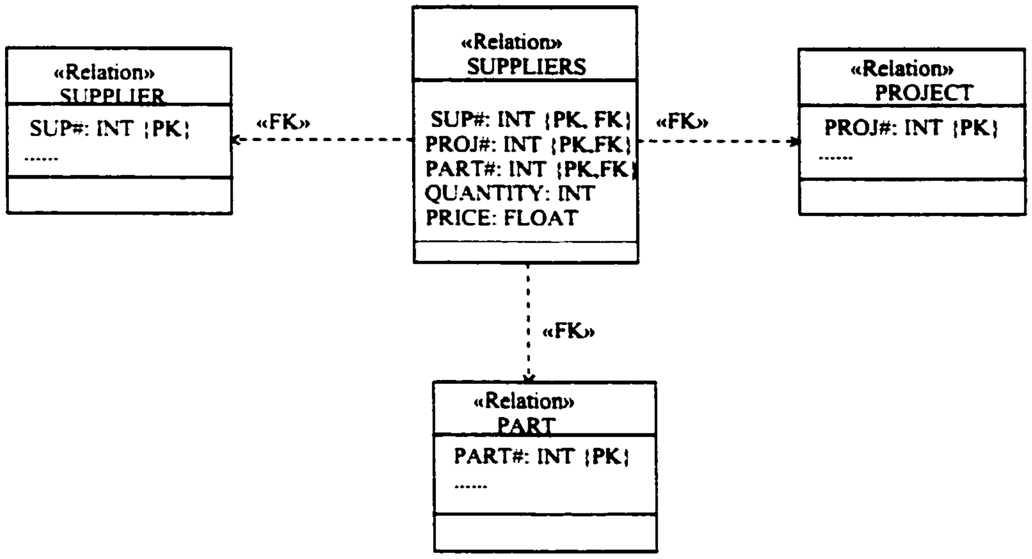


Figure 4.24: A ternary relationship

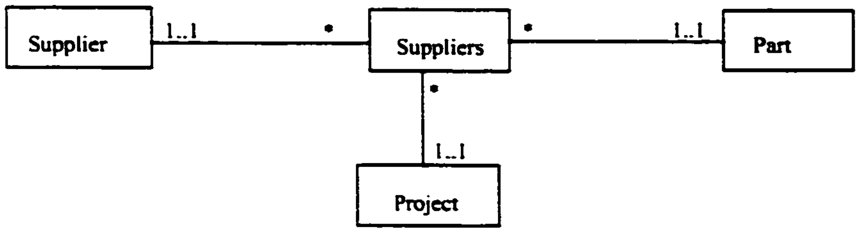


Figure 4.25: An OO model for a ternary relationship

For the OO model in figure 4.25, `supplier` is a ternary association. Because in OO model, ternary association is hard to implement. We can use an alternative binary association to represente it. In figure 4.26, we treat association supplier as a object type, the ternary relationship is transformed into three many-to-one associations.

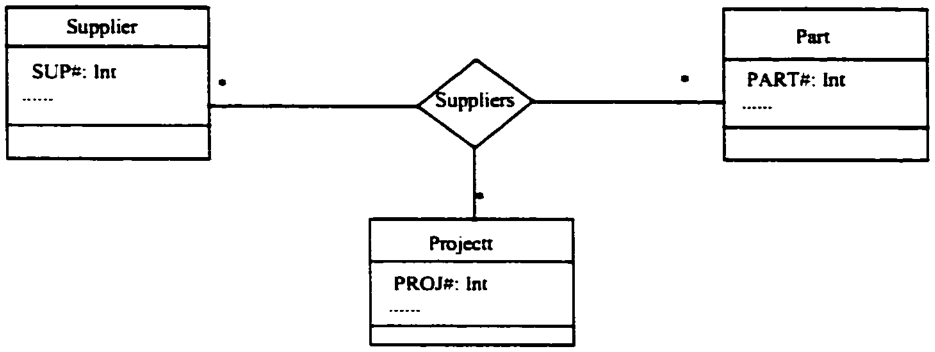


Figure 4.26: A binary representation for a ternary relationship

Hybrid Star Type

In figure 4.27, the primary key of relation ADVISES also servers as the foreign key pointing to another relation STUDENT. Relation advises also has two foreign keys, FAC_ID, AREA_ID.

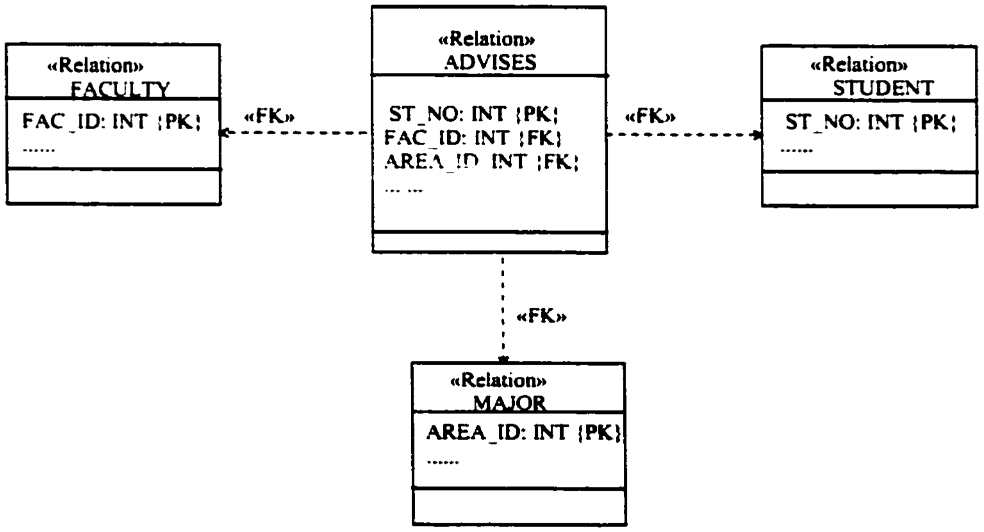


Figure 4.27: A hybrid ternary relationship

***Mapping Implication** : For a set of relation schemas,$R_1,R_2,...,R_n$ if exist a schema R_i, such that D_{out} (Ri)=n-1, consider map schemas to ternary relationships. If R_i has a designated primary key, map R_i to a separate class in object oriented model. If R_i has no designated primary key, map R_i to a ternary relationship with association class in object oriented model.*

4.3.3 Apply the mapping rules in an example

We now present a general method for translating a relational schema into an object oriented model based on the rules that we introduced in previous chapters.

1.Basic Schema transformation: Inclusion Dependency Splitting, Candidate key splitting.

2.Classifying of relational databases, based on types for key-based Inclusion dependency.

3. Mapping the classified schemas accordingly.

4. Fusion and splitting for the mapped UML diagrams.

To illustrate the method above, we now give a comprehensive example :

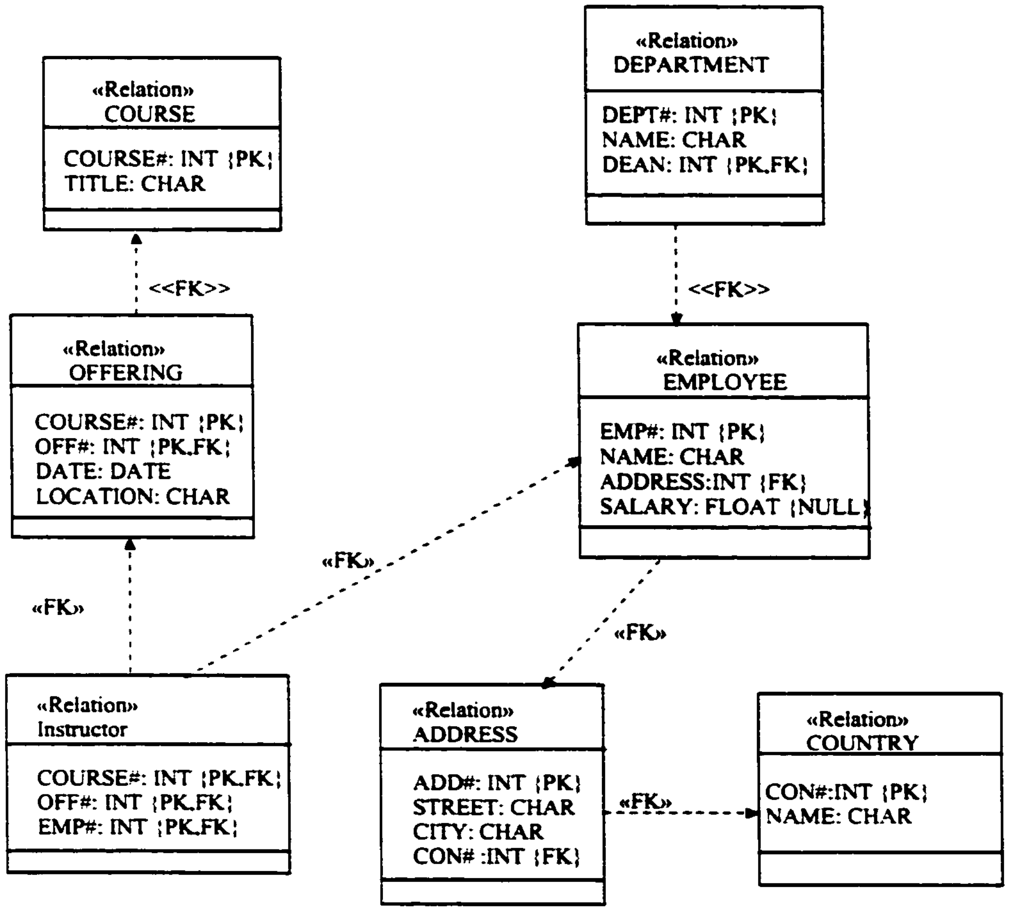


Figure 4.28: A comprehensive example

In the example above, relation DEPARTMENT has two keys: DEPT# and DEAN. Accoring to uur mapping method, relation DEPARTMENT should be splited into two relations:

DEPARTMENT and DEAN, and relation DEAN has a inclusion dependency relationship with DEPARTMENT showed as follows.

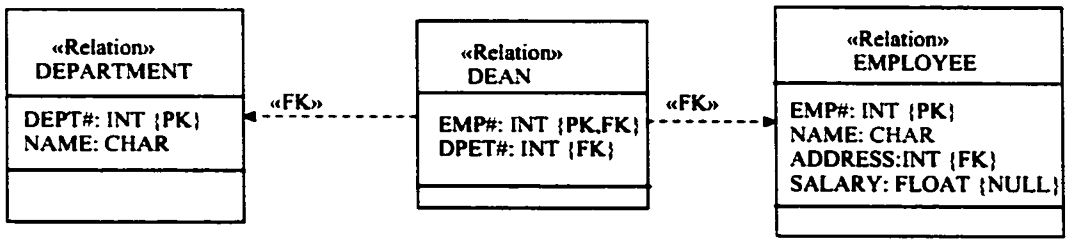


Figure 4.29: Part of schemas after candidate Splitting

Now, according to the second step, we classify the schemas bases on the types we introduced. For this example, we can see that the inclusion dependency between relation COURSE and OFFERING is partial key-to-key, so it maps to a generalization in object oriented model. Relation INSTRUCTOR has no designated primary key and it is foreign key dependent on relation OFFERING and EMPLOYEE, so it maps to an association in object oriented model. The inclusion dependencies between DEPARTMENT and EMPLOYEE, between EMPLOYEE and ADDRESS are non key-to-key. So, these general dependencies map to associations. After step 3, the transformed OO model is in figure 4.30.

After analyzing figure 4.30, we can see that class Employee, Address, Country is in snow flake type (defined earlier). From domain knowledge, we can see that Address, Country can be treated as attributes of class Employee. So, we can aggregate these three classes into class Employee with some attributes functional dependent on other attributes. In addition, for attribute SALARY of relation EMPLOYEE, since it is null allowed, according out mapping rules, this deserve a separate class inherited from class Employee. The finished object model for figure 4.28 is showed in figure 4.31.

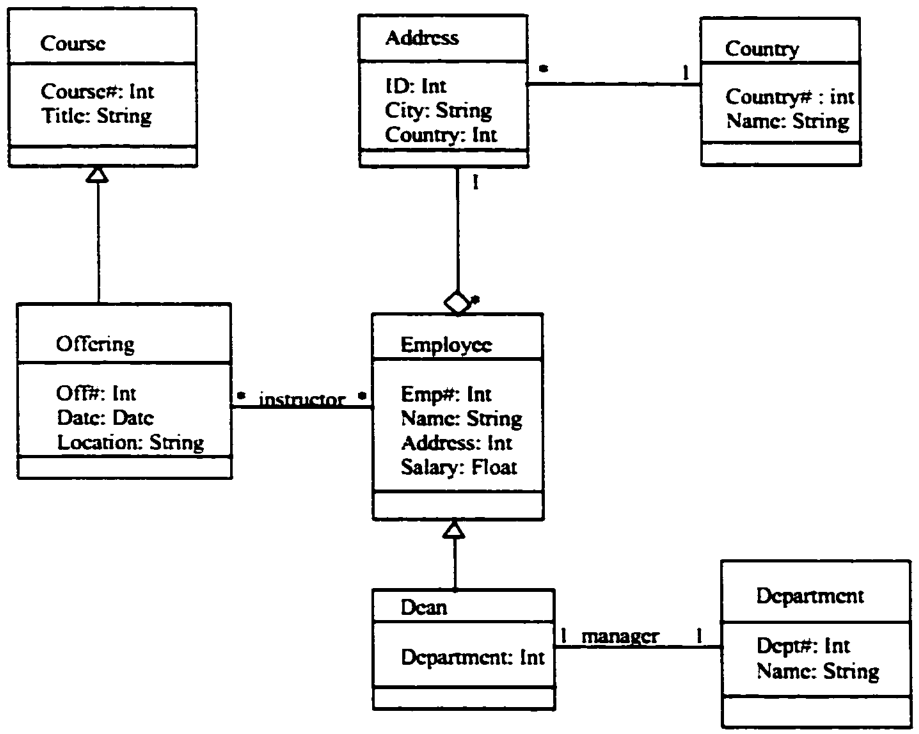


Figure 4.30: An OO model for schemas in figure 4.28

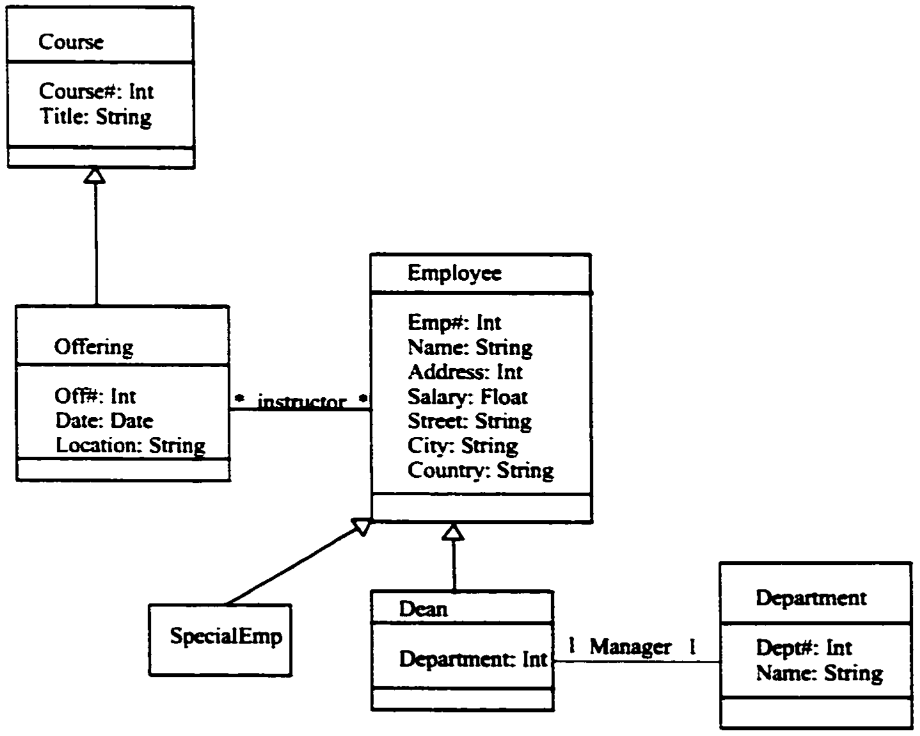


Figure 4.31: A finished OO model for schemas in figure 4.28

4.3.4 More discussions on DBRE

For DBRE approaches, it is important to ensure that the produced semantic models, whether in ER or object oriented model, possess as least as much information as the original relational schemas. With regard to this question, Paul [20] gives formal proofs that his method can produce semantic model which carrying more information than original schemas. In this thesis, we transform relational databases into object oriented models. Since object oriented is not a formal model like relational model, the two models possess information at different levels. So, to test whether the transformation is correct, we propose that this can be conducted in two ways.

An object oriented model in the UML notation, the constructs and associations all have well defined meanings. For classes representing persistent data, rather that implement them in object oriented languages, we translate them into relational tables. Such a method, called object-relation approach, has been proposed by Rational Rose [23]. If we follow a certain object-relation approach, we can map the object oriented models back to relational databases. However, the new mapped databases may not be exactly the same as the original one, but capture same information. So, the correctness of object-relation mapping is much a practical problem, which can be best answered by experiments, rather than a theoretical problem, which can be formally reasoned.

In figure 4.32, there are three classes tagged as {persistent}. The tagged classes represent that they are not transient as other classes. Usually, they map to database tables. Also, we notice that class `CourseManager` is associated with class `Course`. These two classes represent two different designs. Class `CourseManager` is transient class in application design; it capture business rules such as add a course; it is implemented by object oriented languages. Class `course` is a persistent class; it can capture some business rules and maps to a database table. So, at design phase, application design and database are unified into UML representation. Next, for persistent classes, we can follow object-relation approach to transform them into databases tables.

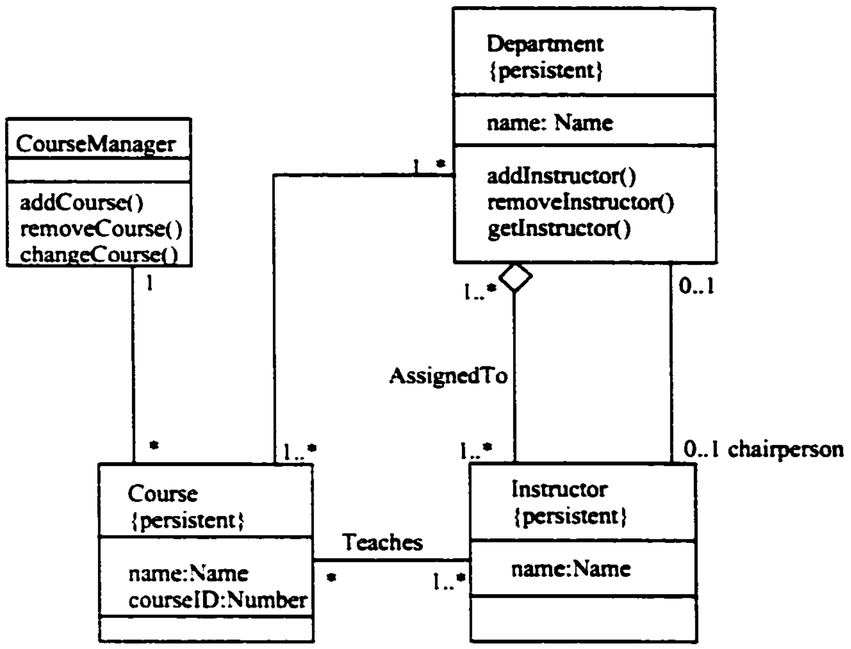


Figure 4.32: An OO model with persistent classes (part)

CHAPTER 5 Conclusion

5.1 Conclusion

In this thesis, we first give a brief yet deep introduction to the UML and its underlying structures, including UML metamodel and Object Constraint Language. Then, we briefly introduce the concepts of relational model. The thesis consists of two parts: representing relational databases in the UML; transform relational databases into object-oriented designs in the UML.

5.1.1 Innovation

The UML is built on a four layer model architecture. Traditionally, users deal with the UML at model level. In this thesis, we propose that the UML can also be used to represent relational databases. To model relational databases, we introduced a set of extended constructs, specifically designed for representing relational databases, using UML extension mechanisms. All the introduced constructs are defined in the UML metamodel level with Object Constraint Language (OCL). The UML representation may interest database designers as well as application developers. Also, we introduced a new approach for database reverse engineering. In this approach, we transform the relational schemas back to object-oriented designs in the UML. With our approach, the logic databases and database designs are unified into the UML notations. This will give a great boost to database development.

5.1.2 Achievements

The major contributions of this thesis is as follows:

- We first apply the UML notation to represent relational databases. In doing so, we introduce a set of new constructs using UML extension mechanisms. In addition, for the new constructs, to make the semantic clear and conform to the relational database concepts, we provide rigorous definitions for them using OCL. We believe that the UML representation of relational database has distinct advantages: the representation is unified into the UML notation; the representation can be interpreted by UML tool generators.

- We provide a general method of transforming relational schemas into object oriented designs. In particular, we first classify relational schemas in the UML representations by the types of inclusion dependencies. Then, we provide transformation rules from classified relational schemas to objected oriented model in the UML. The major advantage of our method is that it unifies object oriented application designs and database designs.

5.2 Future Work

For DBRE approaches, one important issue to address is that whether the mapping rules is complete, that is whether all relational structures in database implementations are covered by the mapping rules. With DBRE approach in this thesis, the set of mapping rules root in database conceptual design technology. We come to the rules by analyzing the various types of conceptual design models, such as binary relationship, ternary relationship, generalization, etc. We believe that the rules covers a reasonably range of database structures. However, there may exist some database structures which are not covered by our mapping rules. Essentially, the completeness of a DBRE approach is largely an empirical question which can be best addressed by testing on real databases [24]. Some researches have addressed this issue as well. A prototype system, called the Knowledge Extraction System (KES) [24,25] has been developed to conduct the testing of their DBRE approach. With regard to the approach in this thesis, we expect a prototype system to be developed to test the completeness and integration with object oriented application designs.

Rational rose [23] has applied the UML in relational database designs in a forward engineering way. And, most DBRE tools transform relational databases into the ER or EER model. For notations and methods introduced in this thesis, we expect a tool generator which understands our notations and apply our transformation rules in the future.

6. References

1. [Catellanos 1991], M. Catellanos amd F. Saltor. "Semantic Enrichment of Database Schemas: An Object Oriented approach". In First International Workshop on Interoperability in Multidatabases systems, Ed, Y. Kambayashi et al, pp71-78,1991.

2. [C.Batini 1992], C. Batini, S. Ceri, and N. Shamkant. "Conceptual Database Design - An Entity-Relationship Approach". Benjamin Cummings, 1992.

3. [C.J.Date 1995], C.J.Date. "An Introduction to Database Systems". 6TH edition, Addison-Wesely Publishing, 1995.

4. [Comyn-Wattiau 1996], Comyn-Wattiau and J. Akoka. "Reverse engineering of relational database physical schemas". In B. Thalheim, editor, Proc. of the 15th Int. Conf. on Conceptual Modeling, pages 372–391, Cottbus, Germany, Oct. 1996.

5. [Debabrata Dey 1999], Debabrata Dey, Veda C. Storey, Terence M. Barron. "Improving Database Design through the Analysis Relationships". ACM Transactions on Database systems,Vol.24, No. 4, December 1999.

6. [E.F.Codd 1970], E.F.Codd. "A Relational Model of Data for Large shared Data Banks". CACM 13, No.6, June 1970.

7. [E.F.Codd 1972], E.F.Codd. "Further Normalization of the Database Relational Model". In Data Base Systems, Courant Computer Science Symposia Series 6. Englewood cliffs, N.J:Prentice Hall, 1972.

8. [E.F.Codd 1988], E.F.Codd. "Domains, Keys, and Referential Integrity in Relational Databases". InfoDB3, No.1, Spring 1988.

9. [Grady Booch 1999] ,James Rumbaugh, Ivar Jacobson and Grady Booch. "The Unified Modeling Language User Guide". Addison-Wesley publishing, 1999.

10. [Java Blend white paper2001], Java Blend white paper. Sun Microsystems Inc., 2001.

11. [J. Hainaut 1993], J.-L. Hainaut, C. Tonneau, M. Joris, and M. Chandelon. "Schema transformation techniques for database reverse engineering". In R. Elmasri and V. Kouramajian, editors, Proc. of the 12th Int. Conf. on Entity-Relationship Approach, pages 353–372, Arlington. Texas, USA, Dec. 1993.

12. [J. Henrard 1998], J. Henrard, V. Englebert, J.-M. Hick, and R. D. "Program understanding in databases reverse engineering". In Int. Work-shop on Program Comprehension, 1998.

13. [J.-M. Petit 1996], J.-M. Petit, F. Toumani, J.-F. Boulicaut, and J. Kouloumd-jian. "Towards the reverse engineering of denormalized relational databases". In Proc. of the 12th Int. Conf. on Data Engineering, New Orleans, USA, Feb. 1996. IEEE Press.

14. [J.Winans 1991], J.Winans and K.H Davis. Software reverse engineering from a currently existing IMS database to an Entity-Relationship Model, in H. Kangassalo, ed., "Entity-Relationship Approach: The Core of Conceptual Modeling" (Elsevier Science, Amsterdam, 1991) 334-348.

15. [L. Pedro 1998], L. Pedro-de Jesus and P. Sousa. "Selection of re-verse engineering methods for relational databases (extended version)". Technical report, IST, Mar. 1998. http://asterix.ist.utl.pt/~ mlp/pubs/selmethod.ps.

16. [Lurdes 1998], "Selection of Reverse engineering Methods for Relational Databases". Lurdes Pedro-de-Jesus, Pedro Sousa.

17. [M. Blaha 1998], M. Blaha and W. Premerlani. "Object-Oriented Modeling and Design for Database Applications". Prentice-Hall, 1998.

18. [OMG 1999], "OMG Unified Modeling Language Specification", version 1.3, June 1999.

19. [O. Signore 1994], O. Signore, M. Loffredo, M. Gregori, and M. Cima. "Using procedural patterns in abstracting relational schemata". In Proc. of the 13th Int. Conf. on Entity-Relationship Approach, volume 881 of LNCS, Dec. 1994.

20. [P. Johannesson 1994], P. Johannesson. "A method for transforming relational schemas into conceptual schemas". In Rusinkiewicz, editor, Proc. of the 10th Int. Conf. on Data Engineering, pages 115– 122, Houston, 1994. IEEE Press.

21. [PowerDesigner 2001], PowerDesigner 8.0 Beta version documentation, January, 2001.

22. [P.P.Shan Chen 1976], Peter Pin-Shan Chen. "The Entity-Relationship Model-Toward a Unified View of Data". ACM TODS1, No.1 ,March 1976.

23. [Rational rose 2000], "Rational rose white paper". Rational Inc, 2000.

24. [R.Chiang 1994], R. Chiang, T. Barron, and V. Storey. "Reverse engineering of relational databases: Extraction of an EER model from a relational database". Data & Knowledge Engineering, 12:107– 142, 1994.

25. [R.Chiang 1995], R. Chiang. "A knowledge-based system for performing re-verse engineering of relational databases". Decision Support Systems, 13:295–312, 1995.

26. [R.Chiang 1997], R. Chiang, T. Barron, and V. Storey. "A framework for the design and evaluation of reverse engineering methods for relational databases". Data & Knowledge Engineering, 21:57- 77, 1997.

27. [Ronald Fagin 1977], Ronald Fagin. "Multivalued Dependencies and a New Form for Relational Databases". ACM TODS2, NO.3, September 1977.

28. [Ronald Fagin 1977], Ronald Fagin. "A Complete Axiomatization for functional and Multivalued Dependencies". Proc.1977 ACM SIGMOD International Conference on Management of Data, Toronto, Canada, August 1977.

29. [Ronald Fagin 1977], Ronald Fagin. " A Normal Form for Relational Database That is Based on Domains and Keys". ACM TODS 6, No3, September 1981.

30. [V. Markowitz 1990], V. Markowitz and J. Makowsk. "Identifying extended entity-relationship object structures in relational schemas". IEEE Transactions on Software Engineering, 16(8), Aug. 1990.

31. [W. Premerlani 1994], W. Premerlani and M. Blaha. "An approach for reverse engineering of relational databases". Communications of the ACM, 37(5), May 1994.

32. [W.W.Armstrong 1974], W.W.Armstrong. "Dependency Structure of Data Base Relationships". Proc. IFIP congress, Stockholm, Sweden, 1974.

VITA AUCTORIS

Xin Zhao was born in 1969 in Henan, P.R.China. He graduated from Huazhong University of Science and Technology where he obtained a B.E in Computer Science & Engineering in 1990. He is currently a candidate for the Master's degree in Computer Science at the University of Windsor and hopes to graduate in Fall 2001.