JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 5, September-October 2003

UML Specification of Relational Database

Liwu Li, University of Windsor, Canada
Xin Zhao, FundMonitor, Canada

Abstract

Database reverse engineering (DBRE) recovers a database design using a semantic data model. Most of the existing works and tools for DBRE and database design specify relational database schemas with extended ER models. The Unified Modeling Language (UML) is a standard language for modeling software and database systems. We discuss how to extend the UML metamodel with elements for modeling relational dependencies. We also present techniques for converting structures of relational dependencies to UML constructs. The introduced metaelements and conversion techniques can be used in relational database design that is presented in the UML. They unify object-oriented software design and relational database design.

1 INTRODUCTION

Common tools for relational database design and database reverse engineering (DBRE) are based on extended ER models. An application that stores data in a database needs to design the database and the application. The application design focuses on business logic and GUI. The database design defines persistent data to be stored in the database. Software developers need to integrate database design and application design. Interface and overlapping problems challenge designers of database applications. A design in the Unified Modeling Language (UML) [OMG 2000] for a database application needs to represent a database design at an appropriate abstraction level.

The UML [OMG 2000] is a standard language for specifying models and designs in object-oriented software development. In this paper, we extend the UML metamodel with relational dependencies and discuss how to convert structures of relational dependencies occurred in a relational database design to UML constructs. The extension and conversion techniques can be applied to integrate a relational database design with a software design expressed in the UML.

In relational DBRE, relational dependencies can be discovered by examining relation instances [Petit1996]. A database designer can identify relational dependencies by studying the data used in an application. We shall regard a relational database design as composed of relation schemas interrelated with functional and inclusion dependencies.

Cite this article as follows: Liwu Li, Xin Zhao: "UML Specification of Relational Database", in *Journal of Object Technology*, vol. 2, no. 5, September-October 2003, pp. 87-100. http://www.jot.fm/issues/issue_2003_09/article1

Here, the task is to unify a relational database design and an application design that uses the database.

This paper is organized as follows. In the next section, the UML metamodel is extended with metaelements for specifying foreign key, candidate key, functional dependency, and inclusion dependency, which are essential concepts in the relational data model. Based on the introduced metaelements, we discuss how to convert individual relational dependencies and structures of relational dependencies to UML associations, aggregations, and compositions in Section 3. The discussion also shows the necessity of the metaelements for presenting relational database designs in the UML. We conclude the paper in Section 4.

2 EXTENDING THE UML METAMODEL

The UML Metamodel

The UML is based on a four-layer architecture, which consists of a meta-metamodel, a metamodel, a user-defined model or design, and objects [OMG 2000]. The UML metamodel is an instance of the meta-metamodel. It defines the UML language. A user-defined analysis model or system design presented in the UML is an instance of the UML metamodel. Application-specific data is stored in objects, which are created with classes specified in the design.

The UML metamodel specifies metaclasses, relationships between metaclasses, and standard metaelements. It defines well-formedness rules in the Object Constraint Language (OCL). It provides designers with three controllable extension mechanisms – stereotype, tagged value, and constraint. An application-specific *stereotype* is based on a metaclass defined in the UML metamodel. It may have tagged values and constraints. Metaclass `Dependency` in the UML metamodel abstracts semantic relationships between elements in user-defined models. A dependency in a user-defined model indicates that a change to the target element (supplier) of the dependency may cause a change to the source element (client) [OMG 2000, p. 3-82]. The UML metamodel defines standard stereotypes for various types of dependency.

Relational Keys

Following the UML User Guide [Rumba1999], a relation schema is represented with a class symbol stereotyped with keyword `persistent` in a UML class diagram. A tuple of the relation corresponds to an instance of the persistent class. In the following discussion, terms *relation schema* and *persistent class* are used as synonyms.

In a UML model or design, a tagged value introduces a named property for a modeling element. An attribute in a persistent class may be a part of the primary key of the relation schema. We shall attach tagged value `{PK}` to each attribute in the primary key. In the UML metamodel, we can use the following OCL invariant constraint to characterize the relational concept primary key. For a tuple `i` and an attribute set `X`, expression `i.X` in an OCL expression denotes the projection of tuple `i` over `X`.

```
context Class inv:
let primaryKey : Collection(Attribute) =
                       self.allFeatures -> collect(f|f.PK)
self.allInstances -> forAll(i, j | i <> j implies
                       i.primaryKey <> j.primaryKey)
```

In a persistent class, we can attach a tagged value `{FK = (`*`relationName`*`, `*`i`*`)}` to an attribute of the class. The tagged value indicates that the attribute belongs to the *`i`*th foreign key that references relation *`relationName`*. If the name *`relationName`* of referenced relation is irrelevant to the discussion, the tagged value can be simplified to `{FK = `*`i`*`}`. Furthermore, when no ambiguity may occur, the tagged value can be simplified to `{FK}`.

In a persistent class, we can attach a tagged value `{CK = `*`i`*`}` to an attribute. The tagged value indicates that the attribute belongs to the *`i`*th candidate key of the relation schema. If the number *`i`* is irrelevant to the discussion, tagged value `{CK = `*`i`*`}` can be simplified to `{CK}`. By assuming that a persistent class has at most one candidate key, the candidate key can be constrained with the OCL expression:

```
context Class inv:
let candidateKey : Collection(Attribute) =
                       self.allFeatures ->
                             collection(f|f.CK)
self.allInstances -> forAll(i, j | i <> j implies
                       i.candidateKey <> j.candidateKey)
```

Inclusion and Functional Dependencies

In the relational data model, an inclusion dependency $R_1(X) \subseteq R_2(Y)$ is a dependency between an attribute set `X` in a relation schema R_1 and an attribute set `Y` in a relation schema R_2. The related client schema R_1 and supplier schema R_2 may be identical. Inclusion dependencies between attribute sets of same relation schema are important clues to hidden classes [Petit1996].

We introduce stereotype `inclusion` based on the UML metaclass `Dependency`. In the UML, an *inclusion dependency* $R_1(X) \subseteq R_2(Y)$ is defined as a dependency between persistent classes R_1 and R_2 such that the following OCL expression holds:

```
R1.allInstances -> forAll(i | R2.allInstances ->
                           collect(j|i.X = j.Y) -> size > 0)
```

For an inclusion dependency $R_1(X) \subseteq R_2(Y)$, if `Y` is a candidate key of R_2, the dependency is *key based*. If `Y` is the primary key of R_2, the dependency is a *foreign key dependency*.

As shown in the UML class diagram in Fig. 2.1, an inclusion dependency $R_1(X) \subseteq R_2(Y)$ between persistent classes R_1 and R_2 is stereotyped with keyword `inclusion`. The source relation R_1 is the *client*, and the target relation R_2 the *supplier*. A tagged value with property name `ID` for the dependency specifies the pair of attribute sets `X` and `Y`. A foreign key dependency is a special type of inclusion dependency. It is stereotyped with keyword `foreign_key`.

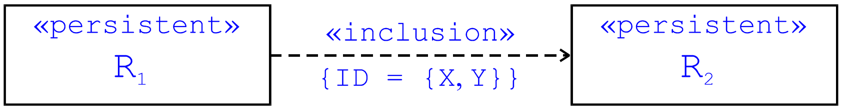


Fig. 2.1 An inclusion dependency

For a relation schema `R` and attribute subsets `X` and `Y` of `R`, a functional dependency `R: X -> Y` will be denoted with stereotype `functional`, which is based on metaclass `Dependency` and which is constrained with the OCL expression:

```
R.allInstances -> forAll(i, j | i.X = j.X implies
i.Y = j.Y)
```

For a candidate key `K` of a persistent class `R` and any attribute subset `A` of class `R`, the candidate key implies a functional dependency `R: K -> A` in a UML class diagram.

As shown in the UML class diagram in Fig. 2.2, a functional dependency `R: X -> Y` will be stereotyped with keyword `functional`. A tagged value with property name `FD` is used to specify the pair of source and target attribute sets `X` and `Y` of the functional dependency.

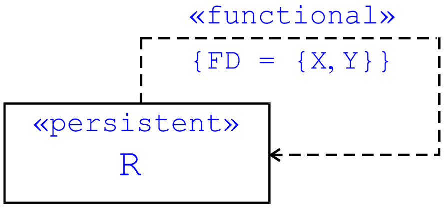


Fig. 2.2 A functional dependency

Inclusion Dependency Clustering

Given a relation schema `R`, the number of inclusion dependencies in which `R` serves as a client and the number of inclusion dependencies in which `R` serves as a supplier are denoted with $D_{out}(R)$ and $D_{in}(R)$, respectively. The numbers $D_{out}(R)$ and $D_{in}(R)$ can be defined in the OCL as follows. Here, we use a metaclass named `Inclusion` to abstract inclusion dependencies.

```
context R: Class inv:
D_out(R) = Inclusion.allInstances ->
                         select(d|d.client = R) -> size
D_in(R) = Inclusion.allInstances ->
                         select(d|d.supplier = R) -> size
```

An inclusion dependency connects two persistent classes R_1 and R_2 with $D_{out}(R_1) \geq 1$ and $D_{in}(R_2) \geq 1$. If the classes R_1 and R_2 are identical, the inclusion dependency is *self-referencing*; otherwise, it is a *binary* inclusion dependency.

Let R, R_1, …, R_n with $n \geq 2$ be persistent classes. If for each integer i with $1 \leq i \leq n$, X_i is an attribute set of R, Y_i is an attribute set of R_i, and $R(X_i) \subseteq R_i(Y_i)$ is an inclusion dependency, we say that the n inclusion dependencies form a *star structure*. If $n \geq 3$, a subset of the persistent classes R_1, …, R_n can form a star structure with the common client class R.

Let R_1, R_2, …, R_n be persistent classes. If for each integer i with $1 \leq i < n$, X_i is an attribute set of R_i, Y_i is an attribute set of R_{i+1}, and $R_i(X_i) \subseteq R_{i+1}(Y_i)$ is an inclusion dependency, we say that the $n - 1$ inclusion dependencies form a *path structure*. A path structure of length greater than 2 entails subpath structures.

3 REPRESENTING DATA DEPENDENCIES IN THE UML

Self-Referencing Dependencies

A functional dependency in a relational database design can be expressed in a UML class diagram with a `functional` dependency, which is a semantic constraint that applies to only one persistent class.

In database normalization, functional dependencies can be used to detect hidden persistent classes. For example, persistent class `Employee` shown in Fig. 3.1(a) satisfies functional dependency `{ssn} -> {empID, dept, job, salary}`. By separating the source attribute set from class `Employee`, we derive a new class `Person` shown in Fig. 3.1(b). Since class `Person` can be used and inherited by other classes, the design in Fig. 3.1(b) is more robust than that in Fig. 3.1(a). In Fig. 3.1(b), a UML aggregation is used to join classes `Employee` and `Person` so that each employee object is linked to a person object, which holds the employee's social security number, name, and address. Thus, the functional dependency is represented with an aggregation.

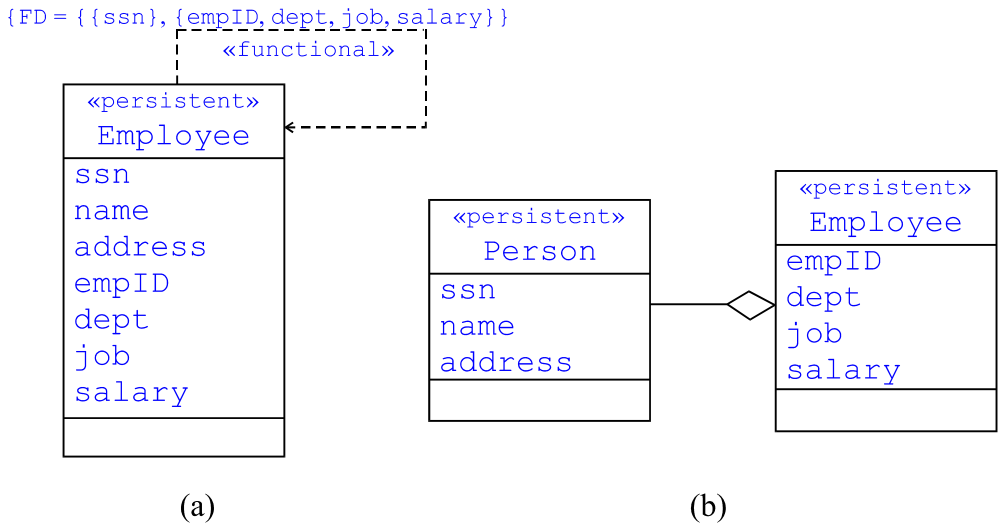
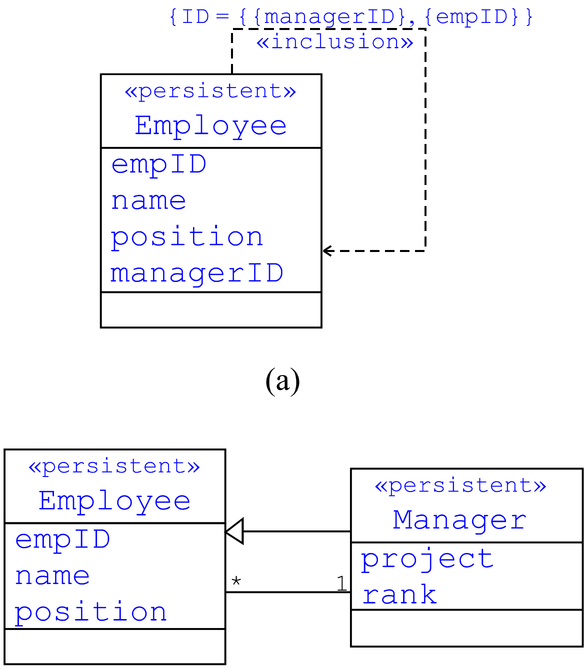


Fig. 3.1 A functional dependency with a hidden object type

Not only functional but also inclusion dependencies may be self-referencing. Self-referencing inclusion dependencies may imply hidden persistent classes as well. Some of them may be represented as various types of association in a UML design. For example, persistent class `Employee` in Fig. 3.2(a) satisfies inclusion dependency `Employee(managerID)` $\subseteq$ `Employee(empID)`, which maps an employee object to an employee object that represents a manger managing the former employee. The self-referencing inclusion dependency can be used to derive a new class `Manager` and a many-to-one association between persistent classes `Employee` and `Manager`. Note that a manager is also an employee. Therefore, class `Manager` inherits class `Employee`.



(a)

(b)

Fig. 3.2 An inclusion dependency implies a hidden subclass `Manager`

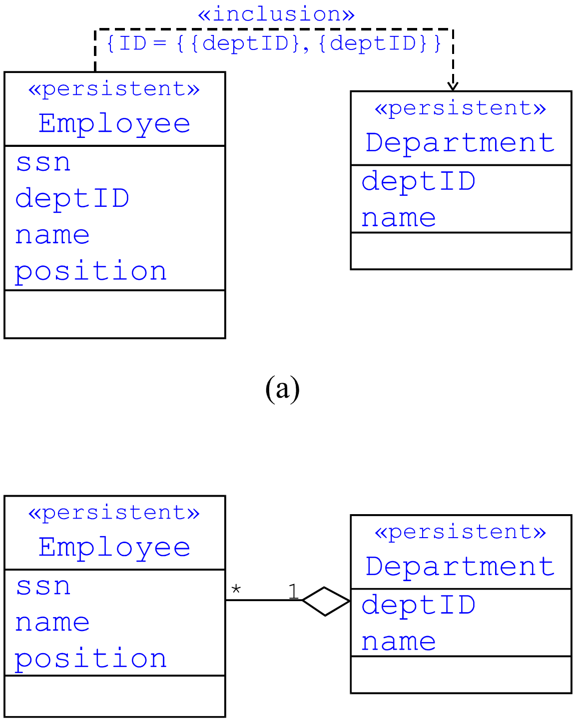
Binary Inclusion Dependencies

A general inclusion dependency in a database design represents a many-to-many semantic dependency, which may or may not denote a UML association. For example, attribute `courses` of class `Instructor` depends on attribute `courseID` of class `Course`. The dependency requires each course taught by an instructor have a valid course ID. It is a many-to-many inclusion dependency, which can be represented in a UML design with the stereotype `inclusion` introduced in Section 2.3.

A binary inclusion dependency may be a foreign key dependency, which denotes a many-to-one or one-to-one relationship. For example, an inclusion dependency between persistent classes `Employee` and `Position` describes assignment of employees to positions. Each employee must be assigned to a specific position. The inclusion dependency represents a many-to-one association.

When presenting a many-to-one inclusion dependency in a UML design, the dependency may be represented as a many-to-one association or an aggregation. It is also possible that the client class can be merged into the supplier class so that the total number of classes can be reduced and the association or aggregation can be eliminated from the design.

For example, inclusion dependency `Employee(deptID)` ⊆ `Department (deptID)` denotes a foreign key dependency. In the UML, we say that each department aggregates a collection of employees. Therefore, the inclusion dependency shown in Fig. 3.3(a) can be converted to the aggregation shown in Fig. 3.3(b).



(a)

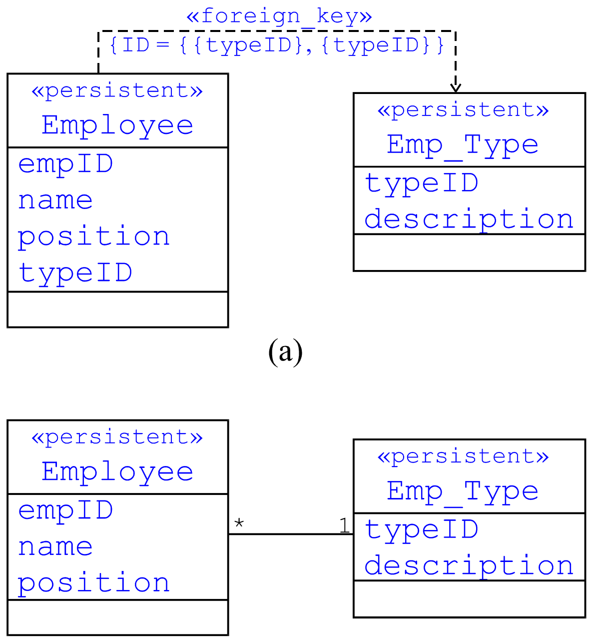
(b)

Fig. 3.3 An inclusion dependency implies an aggregation

Assume a relation schema `Emp_Type` that abstracts employee categories. Each employee is described with an `Emp_Type` tuple. The relationship between relations `Employee` and `Emp_Type` can be described with a foreign key dependency shown in Fig. 3.4(a). Also assume that both the persistent classes `Employee` and `Emp_Type` have reasons to exist independently. The foreign key dependency can be represented with a many-to-one association shown in Fig. 3.4(b).

An inclusion dependency may denote a composition in a UML class diagram. For example, relation schemas `Dependent` and `Employee` can be related with inclusion dependency `Dependent(empID)` $\subseteq$ `Employee(empID)`, where `empID` is the primary key of relation `Employee`. Due to the strong bond between an employee and the employee's dependents, the inclusion dependency shown in Fig. 3.5(a) can be converted to a UML composition between classes `Dependent` and `Employee` shown in Fig. 3.5(b).

By the above discussion, a single binary inclusion dependency may denote an association, an aggregation, or a composition. It is also possible that the inclusion dependency may not denote any association and, therefore, cannot be represented with any association. A binary inclusion dependency may be a clue for merging the related classes.



(a)

(b)

Fig. 3.4 A foreign key dependency is converted to a many-to-one association

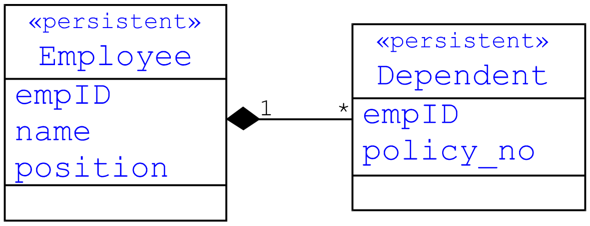


Fig. 3.5 A composition derived from an inclusion dependency

Star Structure

Assume a persistent class `R` that has out-degree $D_{out}(R) \geq n \geq 2$. Some of the inclusion dependencies with client `R` may form a star structure that represents an `n`-ary association. For example, persistent classes `Person`, `Company`, and `Works_For` in Fig. 3.6(a) form a star structure with $n = 2$. Class `Works_For` uses foreign keys `ssn` and `company_name` to reference classes `Person` and `Company`, respectively. As shown in Fig. 3.6(b), the foreign key dependencies can be converted to an association class `works_for` in the UML.

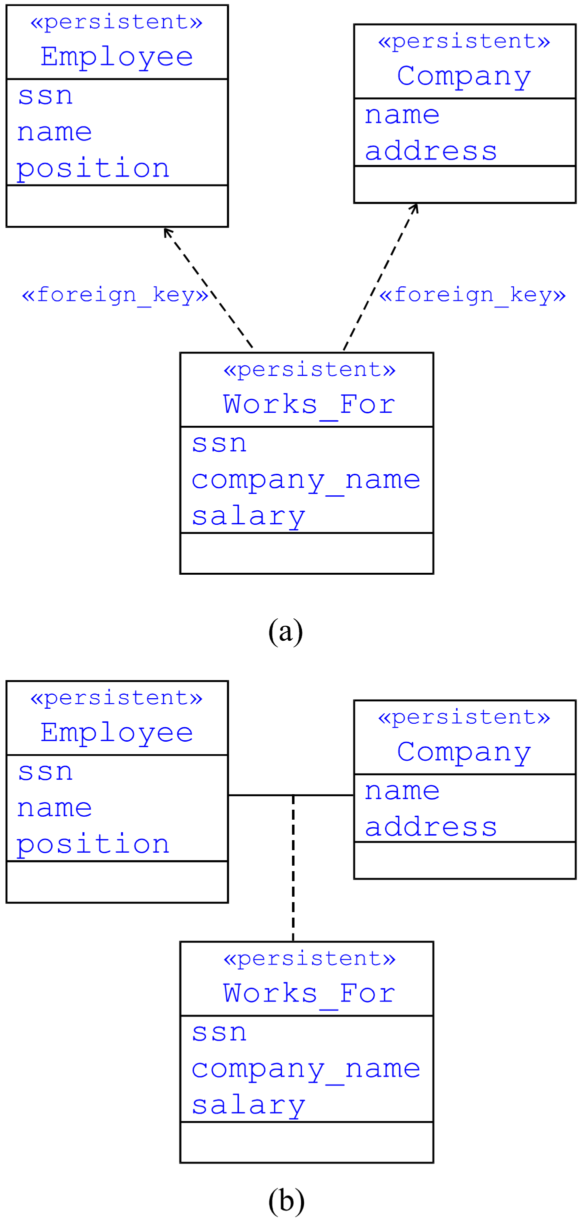
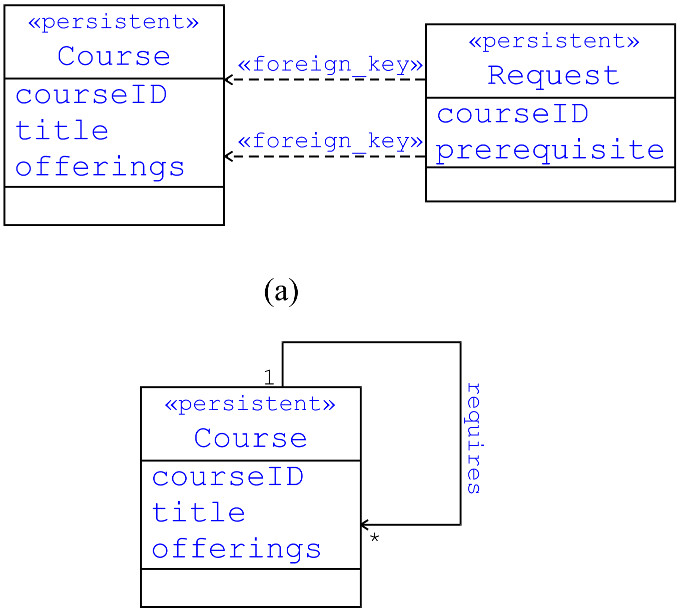


Fig. 3.6 A star structure that represents a binary association

Two or more inclusion dependencies in a star structure may share a supplier. The client and supplier of an inclusion dependency in a star structure may be identical. For example, persistent class `Request` in Fig. 3.7(a) includes foreign keys `courseID` and `prerequisite`, both of which reference supplier class `Course`. In Fig. 3.7(b), the dependencies are represented with a UML one-to-many self-referencing association `requires` between class `Course` and the class itself.



(a)

(b)

Fig. 3.7 Inclusion dependencies with a common supplier and client

Inclusion dependencies in a star structure may be semantically unrelated at all and, therefore, cannot be represented with a single `n`-ary association in the UML design. Hence, they should be implemented separately. For example, persistent class `Employee` may use foreign keys `deptID` and `project_title` to reference classes `Department` and `Project`, respectively. The two foreign keys are independent from each other. They should be implemented separately.

In summary, a star structure of inclusion dependencies may be converted to a binary or an `n`-ary association with `n` ≥ 3. By investigating the inclusion dependencies, we can decide whether the dependencies can be converted to an association. A binary association can be implemented in object-oriented languages directly.

Path Structure

Due to the 1-NF requirement, a relational database design cannot use complex data elements. It may need a path structure to support data navigation. By considering path structures that are formed with inclusion dependencies, we may merge persistent classes so that the number of classes and dependencies can be reduced. Like a single inclusion dependency, a path structure in a relational database design may represent an association, an aggregation, or a composition in a UML design.

For example, inclusion dependencies `Part(typeID)` ⊆ `Part_Type (typeID)`, `Part_Type(typeID)` ⊆ `Supplying(typeID)`, `Supplying (supplierID)` ⊆ `Supplier(supplierID)` compose a path structure, which connects relation schemas `Part`, `Part_Type`, `Supplying`, and `Supplier`. In a UML design, relation schemas `Part`, `Part_Type`, and `Supplying` may be combined into a single persistent class `Part`. The inclusion dependencies can be represented with a many-to-many association between persistent classes `Part` and `Supplier`. The association is used to record which parts are supplied by which suppliers.

4 CONCLUSION

We regard a relational database design as a set of relation schemas and a set of dependencies between the relation schemas. We discuss how to specify relational database design in the UML. The UML metamodel is extended with stereotypes `inclusion`, `foreign_key`, and `functional`, which are based on the UML metaclass `Dependency` and which abstract inclusion dependencies, foreign key dependencies, and functional dependencies presented in a relational database design. We show that some of the relational dependencies can be converted to associations, aggregations, or compositions in the UML. By merging persistent classes, the number of dependencies may be reduced.

By investigating relational database design and relational dependencies from the viewpoint of a UML designer, we see that some functional and inclusion dependencies cannot be represented with UML associations or with other types of UML modeling element. These dependencies justify our stereotypes `inclusion`, `foreign_key`, and `functional`. Other functional and inclusion dependencies are disguised associations, aggregations, or compositions in the UML.

An `n`-ary association for $n \geq 3$ in a UML design is difficult to implement. The above discussion indicates that an `n`-ary association for $n \geq 3$ can be implemented with a set of foreign key dependencies. The 1-NF requirement on a relational database schema may force a path structure of inclusion dependencies in the database design. The above discussion shows that the path may be reduced to an association in the UML by merging persistent classes. The introduced stereotypes `inclusion`, `foreign_key`, and `functional` are indispensable for some relational database designs. They improve the expressive power of the UML.

REFERENCES

[Blaha1998] M. Blaha and W. Premerlani: *Object-Oriented Modeling and Design for Database Applications*. Prentice-Hall, 1998.

[Catel1991] M. Catellanos and F. Saltor: "Semantic enrichment of database schemas: An object oriented approach". In *First International Workshop on Interoperability in Multidatabases Systems*, Editors Y. Kambayashi et al, pages 71-78, 1991.

[Chian1995] R. Chiang: "A knowledge-based system for performing reverse engineering of relational databases". *Decision Support Systems*, 13: 295–312, 1995.

[Chian1997] 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.

[Chian1994] 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.

[Dey1999] D. Dey, V.C. Storey, and T.M. Barron: "Improving database design through the analysis relationships". *ACM TODS*, 24(4), December 1999.

[Haina1993] J.-L. Hainaut, C. Tonneau, M. Joris, and M. Chandelon: "Schema transformation techniques for database reverse engineering". In *Proc. of the 12th Int. Conf. on Entity-Relationship Approach*, Eds. R. Elmasri and V. Kouramajian, pages 353–372, Arlington. Texas, USA, Dec. 1993.

[Henra1998] J. Henrard, et al: "Program understanding in databases reverse engineering". In *Int. Workshop on Program Comprehension*, 1998.

[Jesus1998] L. P.-de Jesus and P. Sousa: "Selection of reverse engineering methods for relational databases". Technical Report, IST, March 1998. http://asterix.ist.utl.pt/˜mlp/pubs/selmethod.ps.

[Johan1994] P. Johannesson: "A method for transforming relational schemas into conceptual schemas". In *Proc. of the 10th Int. Conf. on Data Engineering*, Editor Rusinkiewicz, pages 115– 122, Houston, 1994.

[Marko1990] V. Markowitz and J. Makowsk: "Identifying extended entity-relationship object structures in relational schemas". *IEEE Transactions on Software Engineering*, 16(8), 1990.

[OMG 2000] OMG: *OMG Unified Modeling Language Specification*, version 1.3, March 2000.

[Petit1996] 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.

[Preme1994] W. Premerlani and M. Blaha: "An approach for reverse engineering of relational databases". *CACM*, 37(5), May 1994.

[Rumba1999] J. Rumbaugh, l. Jacobson, and G. Booch: *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[Signo1994] 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*, LNCS Volume 881, Dec. 1994.

[Watti1996] C. Wattiau and J. Akoka: "Reverse engineering of relational database physical schemas". In *Proc. of the 15th Int. Conf. on Conceptual Modeling*, Editor B. Thalheim, pages 372–391, Cottbus, Germany, October 1996.

[Winan1991] J. Winans and K.H Davis: "Software reverse engineering from a currently existing IMS database to an Entity-Relationship Model", in *Entity-Relationship Approach: The Core of Conceptual Modeling*, Editor H. Kangassalo, pages 334-348, Elsevier Science, Amsterdam, 1991.

About the authors

Dr. Liwu Li is a faculty member in School of Computer Science at University of Windsor, Canada. His research interests include object-oriented language design and implementation, object-oriented software analysis and design, and software process design and execution. He can be reached at liwu@uwindsor.ca.

Xin Zhao received his M. Sc. Degree in Computer Science in 2001 from University Of Windsor, Canada. He is currently working at FundMonitor as a software engineer. His skills include RDBMS (SQL server), several programming languages (Java, PowerBuilder), and UML. He can be reached at timzhao@fundmonitor.com.