Adaptive Schema Design and Evaluation in an Object-Oriented Information System

Ling Liu

University of Alberta, Department of Computing Science
615 General Service Building, Edmonton, Alberta, T6G 2H1 Canada

Abstract. This paper develops a formal framework for characterizing "normalization" in object-oriented schema design. We introduce style rules for achieving validity, minimality, and extensibility of an object-oriented schema, and the style rules that help to eliminate update anomalies and undesirable existence dependency loops. As a result, the adaptiveness of an object-oriented schema against future requirement changes is increased. This set of style rules can be used as not only a user-oriented method for designing the object-oriented schema, but also a means for validating quality of a schema and for transforming an object-oriented schema into a better style in terms of adaptiveness and robustness.

1 Introduction

An object-oriented information system, like any other software system, is seldom designed in "one shot", but rather explored through an iterative process of refinement, adaptation, and improvement. Change requirements may occur at various stages in the development cycle of the system and for a number of reasons [2,5,14,19]. For example, changes may be required (i)when experience shows how the system can be improved, (ii)when user needs a change and additional functionality has to be integrated, or (iii)when the universe of discourse modeled by the system changes and the system needs to be adapted. Due to the increased complexity in both schema design and development of application programs working with the schema, the robustness and adaptiveness of an object-oriented schema against future requirement changes become increasingly important. An object-oriented schema is *adaptive* and *robust* if it correctly uses the object-oriented modeling concepts, and can be easily adapted to changing requirements by encouraging as much information localization and data abstraction as possible, such that the overall impact of schema changes or database update can be minimized or reduced.

Most research on object-oriented design has so far been concentrated on how to use object-oriented concepts in logical design and conceptual modeling. However, there is surprisingly very little on evaluation of what constitutes a robust and adaptive schema in anticipation of future requirement changes. Many desired properties of a schema design (such as validity, extensibility, minimality, normality) [3] are only vaguely understood in the context of object-oriented models. Recall the relational database theory, quality design and evaluation of a schema is defined by the presence and absence of certain normal forms. Unfortunately, these normal forms have been developed only in the context of special classes of rules, namely functional and multi-valued dependencies, aiming at elimination of update anomalies implied in a relational schema. Therefore, the relational definition of schema quality is rather restrictive and excludes many other important issues, such as the complexity and extensibility of a schema, the normal form for subclass relationships, for existence dependencies, inclusion dependencies, and so on. We firmly believe that progress in quality design and evaluation of an object-oriented schema can only be made by methodologies that are able to integrate design rules with quality measures, and that clearly address the issues of adaptiveness and robustness of an object-oriented schema with important theoretical and practical implications.

In this paper we develop a formal framework for characterizing "normalization" in object-oriented schema design. Thank to the fact that schema transformation has been explored quite systematically, and many criteria have been introduced to ensure that a schema transformation preserves the nature and the validity of the original schema [1,4,9,11], this paper thus focuses more on how the desired properties of a schema design should be understood in the context of object-oriented models and what are the important design and evaluation rules for building highly adaptive object-oriented schema. Our primary goal is aimed to automate quality design and evaluation for building an adaptive object-oriented schema, which on the one hand eliminates the undesired dependencies and minimizes the impact of schema modifications, and on the other hand, is able to preserve all the necessary dependencies and thus the database integrity constraints required by applications during schema enhancement process and schema change management.

2 The Reference Object Data Model

Let **O** be an infinite set of objects, each has a unique, system-supplied identifier, and **P** denote an infinite set of property/role functions. The term **object** is used in its most general form. It may stand for a tuple (row) in a relational table, an entity (or a relationship) in the entity-relationship model, an entity in a semantic data model, or an object in an object-oriented data model. Objects are grouped into classes based on a set of common properties/roles, and only accessible through their property functions defined in their classes. A **property** function $p \in \mathbf{P}$ can be a value of a primitive base type (e.g., String, Integer, Real, Boolean, etc.), an object instance of a class, a function of arbitrary complexity, or an object method. Each $p \in \mathbf{P}$ has a name and a signature (i.e., domain types). In the sequel, for simplicity, we assume that every $p \in \mathbf{P}$ has a unique property identifier. It is known that to determine the identity of two property functions is equally hard as to prove the equivalence of two programs.

The term **class** serves a dual purpose. A class imposes a *type description* which consists of a finite set of property functions as a common interface, and meanwhile represents a *set* of objects that conforms to its type. Each class is described by a unique class name, a type description, and a set membership. We use **type**(u) to denote the type of class u and **properties**(u) to denote the set of property functions defined in the type of class u, including both locally defined and inherited functions. For each $p \in$ **properties**(u), we refer to the domain class of p by **domain**(p,u) and the collection of objects that belong to class u by **content**(u) $=_{\mathrm{def}} \{o \mid o \in u\}$. The membership predicate "$\in$" is defined based on object identities [18]. Let $X = \{p_i \mid p_i \in \mathbf{properties}(u),\ 1 \le i \le n\}$ be a set of properties of class u. Then, **domain**(X,u) $= \cup_{i-1,\ldots,n}$ **domain**(p_i ,u).

Definition 1 Let u and v denote any two classes. We say that u is a **subset** of v, denoted as $u \subseteq v$, if and only if (iff) for any $o \in$ O, if o is a member of class u, then o is a member of class v.

Definition 2 (subtype) Let u and v denote any two classes. We say that u is a **subtype** of v, denoted as $u \le v$, iff (i) **properties**(u) $\supseteq$ **properties**(v) and (ii) ($\forall p \in$ **properties**(v))(**domain**(p,u) $\subseteq$ **domain**(p,v)).

Definition 3 (subclass) Let u and v denote any two classes. We say that u is a **subclass** of v, denoted as u **is-a** v, iff $u \subseteq v$ and $u \le v$.

Definition 4 (construction relationship) Let u and v denote any two classes. If class u has property p defined by **domain**$(p,u)=v$, then we say that class u has a direct **construction** (composition) relationship labeled p with class v.

An object-oriented schema describes the logical structure of objects and classes. We formalize an object schema in terms of a class dictionary graph, so-called *schema graph*. Two kinds of relationships are distinguished among classes: inheritance relationships (called *specialization* edges) and object reference relationships (called *construction* edges). We refer to the set of *is-a* relationships of a schema as the **specialization hierarchy** and the set of *object construction* relationships as **construction hierarchy**. An object construction hierarchy may consist of several disconnected subgraphs with possibly multiple construction edges, and construction loops or self construction loops.

Definition 5 (schema graph) An object schema G is defined as a labeled, directed class dictionary graph or so-called **schema graph**, denoted as $G = (V, L, E)$ where V is a finite set of class vertices with a vertex **Object** as the *root* class; L is a set of labels, each described by a character string; and $E = EA \cup EC$ is a set of edges where EA is a binary relation on $V \times V$, representing *is-a* (*specialization*) edges; and EC is a ternary relation on $V \times V \times L$, representing object *construction* edges. Thus a schema graph may also be denoted by $G = (V, L, EA, EC)$.

Definition 6 (*specialization reachable*: $\Rightarrow^*$) Let $G = (V, L, EA, EC)$ be a schema graph. A class $u \in V$ is **specialization reachable** from class $v \in V$, denoted by $v \Rightarrow^* u$, iff one of the following conditions is satisfied: (i) $u = v$; (ii) $(v, u) \in EA$; (iii) $\exists w \in V$ s.t. $w \neq u$, $w \neq v$, and $v \Rightarrow^* w$, $w \Rightarrow^* u$.

Definition 7 (*construction reachable*: $\rightarrow^*$) Let $G = (V, L, EA, EC)$ be a schema graph. A class vertex $u \in V$ is **construction reachable** from class vertex $v \in V$, denoted by $v \rightarrow^* u$, iff one of the following conditions is verified:

(i) $\exists l \in L$ s.t. $(v, u, l) \in EC$.
(ii) $\exists w \in V$, $\exists l' \in L$ s.t. $v \Rightarrow^* w$ and $(w, u, l') \in EC$.
(iii) $\exists w \in V$ s.t. $w \neq u$, $w \neq v$, and $v \rightarrow^* w$, $w \rightarrow^* u$.

When a class vertex u is *specialization reachable* from class vertex v, then for any other vertex $w \in V$, if w is construction reachable from u, we may prove that w is also construction reachable from v (specialization inheritance) [18].

Definition 8 (construction closure operator of a class) Let $G = (V, L, EA, EC)$ be a schema graph and $Parts : V \rightarrow 2^V$ be a function defined as follows: For any $u \in V$,

- If $u \in V$ is one of the primitive base classes, such as *String, Real, Integer, Date, Boolean,* then $Parts(u) =_{def} \varnothing$.
- Otherwise, $Parts(u) =_{def} \{ w \in V \mid (u,w,l) \in EC, l \in L \}$.

We define the construction closure operator $Parts^*$ by $Parts^*(u) =_{def} \cup_{k=1,\ldots,|V|} Parts^k(u)$, where $Parts^1(u) =_{def} Parts(u)$ and $Parts^k(u) =_{def} Parts(Parts^{k-1}(u))$ for $k>1$.

Informally speaking, $Parts(u)$ identifies the set of class vertices anchored at vertex u through construction edges. It is the set of domain classes of those properties that are locally defined in class u. Whereas $Parts^*(u)$ computes the set of classes that are construction reachable from the vertex u. It includes all the classes that are used directly or indirectly in construction hierarchy of class u. A schema G is called construction *type-closed* if and only if $Parts^*(u) \subseteq V$ holds in G. We assume that all the schema graphs concerned in the sequel are *construction type-closed*.

3 Style Rules for Validity, Extensibility and Minimality

The validity of an object-oriented schema guarantees that it should have no-dangling class vertex, and satisfy the cycle-free specialization axiom and the unique construction label axiom. These are the baselines of the schema for correct use of the object-oriented modeling concepts [2,12,15,19].

Definition 9 (Validity Normal Form: V-NF)
Let G = (V, L, EA, EC) be a schema graph and V-NF denote a collection of valid schema graphs. The schema graph G belongs to V-NF iff G satisfies the no-dangling class axiom, the cycle-free specialization axiom, and the unique construction label axiom. We refer to this set of schema graphs as V-NF schema graphs.

Extensibility of a database schema is defined by the adaptiveness and the flexibility of the schema in anticipation of future schema changes. The crucial factor for extensibility is to reduce the impact of a schema modification on the entire structure of the existing schema [2,13], on the organization of the databases [14], and on the workload required for rewriting of the existing application programs [21]. The basic principle for achieving better extensibility in an object-oriented database system is to advocate minimal coupling between abstractions (e.g., methods, procedures, and modules) and to provide a certain degree of information hiding and information localization in order to reduce the cost of software changes [17]. Two axioms that we use for promoting extensibility are *information localization* and *minimization of multiple inheritance* (see [16] for detail).

Definition 10 (Inheritance Normal Form: I-NF)
Let G = (V, L, EA, EC) be any schema graph and I-NF denote a collection of adaptive schema graphs by making the best use of inheritance feature. A schema graph G belongs to I-NF if and only if (i) G belongs to V-NF and (ii) G satisfies the information localization axiom and the minimization of multiple inheritance axiom.

A schema is said to be semantically *minimal,* if no concept in the schema can be expressed through composition of the other concepts; more specifically, if no concept can be deleted from the schema without losing information. Redundancy in an object-oriented schema may occur for several reasons. One of the most common causes is due to the fact that application requirements often have an inherent redundancy and this redundancy migrates into the initial schema design. Redundancy can also be incurred when related concepts are expressed in different schemata and these schemata need to be merged such as required in the heterogeneous and distributed database environment [20]. Two types of redundancy are considered here: specialization redundancy and useless class.

Specialization Redundancy is the redundancy that exists when a specialization edge between two class vertices has the same information content as a path of the specialization edges between these two classes. We may describe an specialization path from vertex u to vertex v by a list of vertices $<u, w_1, w_2, ..., w_n, v>$ $(n>1)$, satisfying that (u,w_1), (w_1,w_2), ..., $(w_n,v) \in EA$. The axiom for *specialization minimality* guarantees that schema G has no specialization edge that can be derived from any of the others. It identifies a way of eliminating specialization duplicate by eliminating all the redundant specialization edges implied in specialization abstraction.

For any class vertex u, if it has no incoming specialization edge and $Parts(u)=\emptyset$, then it is desirable to consider vertex u as a *useless class*, because it contains no construction information at all. This style rule is refereed to as the axiom for elimination of useless specification. Most of the useless class specifications initially come from application requirements, and mostly caused by the difficulty in distinguishing between what should be modeled as classes and what are properties of a class in the logical design.

Definition 11 (Minimality Normal Form: M-NF)
Let G = (V, L, EA, EC) be a schema graph and M-NF denote a collection of specialization minimal schema graphs. A schema graph G belongs to M-NF, iff (i) G is in V-NF, and (ii) G has neither specification redundancy nore useless classes.

4 Style Rules for Eliminating Update Anomalies

In the relational data model, normalization of relations is a commonly used means to eliminate database update anomalies [3,10,22]. The higher normal form a schema belongs to, the less update anomalies the schema would have. The intended schema can be obtained by applying progressive schema transformations.

Unlike the definition of relational normal forms, the definition of a normal form for object-oriented schemata should also take into account composite properties (components), multi-valued properties and special types of relationship sets, namely, *is-a* relationship, existence dependent relationship, Union relationship (if an object class is equal to union of some other object classes), Intersect relationship (if an object class is equal to intersect of some other object classes) and Decompose relationship (if an object class can be partitioned into several other classes). In this section we analyze how the theory of functional dependencies can be adapted to the normalization of object-oriented schemata and propose a normal form for object-oriented schema graphs.

4.1 The Problem

Definition 12 (Functional Dependencies) Let C denote an object class and X, Y be two sets of single-valued properties of class C. If both X and Y are single-valued properties of C, then a **functional dependency** (FD) exists between X and Y, denoted as $X \rightarrow Y$, iff each object instance of **domain**(X, C) corresponds to precisely one object instance of **domain**(Y, C) at any time. I.e.,

$$X \rightarrow Y \Leftrightarrow \neg\exists o_1, o_2 \in C : o_1.X = o_2.X \text{ but } o_1.Y \neq o_2.Y.$$

When the right side of a FD is a set of properties, say $X \rightarrow A_1, ..., A_n$, then the original FD is equivalent to n individual FDs, i.e., $X \rightarrow A_1, ..., X \rightarrow A_n$.

A set K of properties (components) of C is said to be a **key** of C iff all properties of C are functionally dependent on K, and there exists no proper subset K' of K such that all properties of C are functionally dependent on K'. A property of C is called a *key property* if it is contained in a key of C; otherwise it is called *non-key property* of C.

Example 1. Assume we have an application, describing `Supplier` in terms of supplier's `ssn`, `date` of supply, `item-no`, `price` per unit, and `quantity` ordered for each item. We may model it as shown in Fig. 1. Suppose that we have the following constraints: properties `ssn` and `item-no` may together decide a `Supplier` object, but the `price` of one item is uniquely determined by its item number. We can describe them in terms of the following two functional dependencies:

ssn, item-no $\rightarrow$ quantity, date;   item-no $\rightarrow$ price.

```
Class Person                    Class Supplier inherit from Person
{   pname: String,              {   item-no: Integer,
    ssn: Integer,                   price: Real,
    birthday: Date  }               date: Date,
                                    quantity: Integer }
```

Fig.1 An example schema violating normalization axiom

As every object in the object-oriented model has a unique (internal) identity, all the internal (system-supplied) identities of objects functionally determine the other single-valued properties (components). Thus we have the following FDs in class `Supplier`:

oid(Person) $\rightarrow$ pname, ssn, birthday.

oid(Supplier) $\rightarrow$ pname, ssn, birthday, item-no,price, date, quantity.

Given a class *C,* we call object identity the **primary key** of class *C* no matter whether it has more than one key or not. All the functional dependencies that correspond to object identities are referred to as "identity-based FDs".

Generally speaking, the identity-based dependencies do not incur problems. The other dependencies that exist in a class of objects, however, are often the cause of information redundancy and update anomalies, even in the context of object-oriented models. For instance, consider the example schema in Fig. 1, the functional dependencies below

ssn, item-no $\rightarrow$ quantity, date;   item-no $\rightarrow$ price

are actually the source of the following three anomalies.

(i) *Insertion anomaly*:

We cannot indicate the `price` of an item unless it has a supplier.

(ii) *Deletion anomaly*:

When we delete the last supplier object for an item in the class `Supplier`, we also lose the information about its `price`.

(iii) *Modification anomaly* (*potential inconsistency*):

If the price of an item is changed, all suppliers referring to this item also need to change. Otherwise, we would not have a unique price for each item as we feel intuitively we should. Thus, the update operation has to be propagated to several `Supplier` objects. Similarly, if we need to add a new property regarding to item, e.g., `item-description`, we have to add it to the class `Supplier`. As a consequence, for each item, all the suppliers who supply the same item have to store its `item-description` repeatedly. Any change to this `item-description` would require to propagate the same change to all the `Supplier` instances in order to preserve the information consistency.

These anomalies are all related to the presence of an undesired functional dependency in the class type `Supplier`, namely, `item-no` $\longrightarrow$ `price`. In this example, this FD is the actual cause for the problem of accidental loss of information, the difficulty of representing given facts, and the cause of some unnecessary work because the same changes must be applied to a number of object or data instances. Elimination of such insertion, deletion and update anomalies encourages the schema design to preserve the logical structure of objects in a clean and robust "normal form". This can be achieved by stepwise and dependency-preserving schema decomposition.

Recall the example in Fig. 1, if we decompose the class `Supplier` into two classes `Supplier` and `Item` (see Fig. 2), then all the above problems go away. The reason is simply because that the class `Item` in Fig. 2 presents the `price` of each item exactly once. Hence, there is no redundancy. Moreover, we can enter a `price` for an item even if it is currently no `Supplier` objects. We can also add new properties for each Item object without incurring any modification anomalies. It is interesting to note that this decomposition not only eliminates certain types of anomalies, but meanwhile also preserves all the original dependencies.

```
Class Person            Class Supplier            Class Item
{ pname: String,         { inherit from Person   {item-no: Integer,
  ssn: Integer,            item: Item,              price: Real }
  birthday: Date  }        date: Date,
                           quantity: Integer }
```

Fig.2 Eliminating update anomalies of the example schema in Fig.1

Yet some questions remain. How do we recognize undesired functional dependencies for a given class type? Does the decomposition work in all the situations? How do we determine if the above replacement is beneficial? How do we find a better replacement for a "bad" object-oriented schema. We may answer these questions by introducing the concept of normalized object-oriented schema.

4.2 Normality of an Object-Oriented Schema

The process of normalization is a progressive detection and elimination of undesired functional dependencies. The main diffidence in extending the treatment of normalization from the relation realm to the object-oriented model arises from the different means of expressing identification. In this section we define a normal form for each class type in an object-oriented schema and the semantics of normalization of an object-oriented schema. The objectives for defining schema normality as such are mainly the following. First, it helps to capture and to preserve all the semantics of the real world which can be expressed in terms of functional dependencies in a database. Second, it ensures that all the relationships represented in the schema are non-redundant. Thirdly, it provides all the schemata with better adaptiveness towards future modification both at schema level and at database level.

Definition 13 (base functional dependencies)
Let C be a class and K denote a primary key (typically object identity) of instances of C. The set of **base dependencies** of C, denoted as BFD(C), is defined as follows:

(i) For each *one to one* property p of C, $K \rightarrow p$, $p \rightarrow K$ are FDs in BFD(C).
(ii) For each *many-to-one* or *many-to-many* property p of C, $K \rightarrow p$ is a FD in BFD(C).
(iii) For each one-to-many property p of C, $p \rightarrow K$ is a FD in BFD(C).
(iv) For each key K_1 of C which is not the primary key of C, $K \rightarrow K_1$, $K_1 \rightarrow K$ are FDs in BFD(C).
(v) No other FDs are in BFD(C).

Axiom (elimination of undesired anomalies) Let G = (V, L, EA, EC) be an object-oriented schema graph . A class $C \in V$ is in a Class Normal Form (C-NF), iff all its functional dependencies that involve only properties of C, can be derived from the set of base FDs of C by using the Armstrong's axioms for functional dependencies [6, 22].

Interesting to note is that the FDs of class C that are not derivable from BFD(C) are the actual cause of update anomalies. If a class is not in C-NF form, then it must have such undesired FDs. Thus, the process of eliminating those undesirable FDs of C should, at the same time, automatically transform the class C to the C-NF form.

Example 2. Recall the schema graph in Fig. 1. Class `Supplier` has the following keys: (`ssn`, `item-no`), oid(`Supplier`) and two FDs in addition to its identity-based FDs. Note that `SSN` is a one-to-one attribute of `Supplier` and a key of `Person` but not a key of `Supplier`**.** The set of base dependencies for class Supplier is as follows:

BFD(Supplier) = { `oid(Supplier)`$\rightarrow$ `pname, ssn, birthday, item-no, price, date, quantity;`
`(SSN, item-no)`$\rightarrow$`oid(Supplier)` }.

Obviously, the FD "`item-no`$\rightarrow$ `price`" is not logically implied by BFD(`Supplier`). According to Axiom 8, it is considered as a undesired FD in class `Supplier`. Therefore, class `Supplier` is not in C-NF form.

In the current style rule for normality, we do not consider *multi-valued* dependencies. However it should not be difficult to add them into the Axiom.

Definition 14 (Normality Normal Form: N-NF) Let G = (*V, L, EA, EC*) be a schema graph, **FD**(G) be the set of FDs of G, and **BD**(G) be the set of **base dependencies** of G, which is the union of the sets of dependencies of all class vertices of G, i.e., **BD**(G) = $\cup_{u\in V}$ BFD(u). We say the schema G belongs to a collection of normalized normal form (N-NF) schemata, if it satisfies the following conditions:

(i) G is in V-NF form, i.e., all property names are distinct and of different semantics.
(ii) Every class in the schema graph G is in C-NF.
(iii) All the functional dependencies are implied by the set of base dependencies of G.

$$\forall \mathrm{FD}_i \in \mathbf{FD}(\mathrm{G}),\ \exists \mathrm{FD}_j \in \mathbf{BD}(\mathrm{G}) \colon \mathrm{FD}_j \Rightarrow \mathrm{FD}_i.$$

Recall the `Supplier` example in Fig.1. Since the FD for `Supplier` class, namely `item-no`$\longrightarrow$ `price`, can not be derived (implied) by the set of base dependencies of `Supplier`. The class Supplier is not in a C-NF. The schema in Fig.1 is thus not a normalized normal form schema (N-NF) as well. A number of problems regarding update anomalies may occur as discussed in Section 6.2. In order to transform this schema to a N-NF schema, we decompose the `Supplier` class into the two classes as shown in Fig. 2.

5 Style Rules for Minimizing Existence Dependencies

An existence dependency between two classes depends on the existence dependency between their instance objects. A class C_1 exists dependently on a class C_2, if and only if every instance of class C_1 must have referenced to at least one instance of class C_2 . It means that there is a total constraint on the relationship between their instance objects.

The main problem with existence dependencies is that a set of existence dependencies together may form a dependency cycle among a given set of classes. We call it a *construction dependency loop* of the schema. Such a loop can become a very dangerous cause for update inconsistency in an object-oriented database, and may often create difficulty and inefficiency during deletion, insertion and population of a database. In order to provide some control to certain types of existence dependencies, we develop a normal form for eliminating undesirable existence dependency loops.

Axiom (minimizing construction dependency loops)
Rule 1: The total number of edges involved in a construction dependency loop should be minimized.
Rule 2: The total number of construction dependency loops in a schema graph should be reduced.

It is indispensable to develop strategies and algorithms in order to break down the loops, because it is usually not very obvious how the minimization or reduction could be suggested or carried out to achieve a better compromise between the efficiency expected and the semantic richness required.

One way to destroy a construction dependency loop is to replace a mandatory construction edge in the loop by an optional one. Of course, consulting with users about the possibility and necessity of doing so is desirable and helpful. Another interesting issue related with the breaking down of construction dependency loops is then the choice of which construction edge could be replaced with less impact on the semantics of the original schema. We below propose a number of policies which can be used as a default choice or advice to the designer.

Policy 1: If there is only one edge involved in more than one construction dependency loop, then it is appropriate to relax the mandatory constraint on this edge by using an optional construction edge to replace the mandatory one.
Policy 2: If several edges are involved in more than one construction dependency loop, then the edge that is involved in the most of the construction dependency loops should be chosen first and be relaxed through replacing its mandatory constraint by an optional one.
Policy 3: Otherwise, chose any edge contained in a given construction dependency loop and relax it by using the optional constraint instead of the mandatory one.

Due to the space restriction, we here only present the cycle-free dependency normal form definition. Readers may refer to [16] for formal definitions of concepts and examples.

Definition 15 (cycle-free dependency normal form: D-NF)
Let G = (*V, L, EA, EC*) be a schema graph and D-NF denote a collection of specialization minimal schema graphs. A schema graph G belongs to D-NF, iff (i) G is in V-NF, and (ii) G has minimal existence dependency loops.

6 Discussion

We have presented a framework for axiomatic characterizations of object-oriented schemata, aiming at enhancing the adaptiveness and robustness of object-oriented schema design. This framework consists of a selection of normal forms, i.e., valid NF, inheritance NF, minimal NF, normalized NF and cycle-free dependency NF (D-NF). We regard the valid NF as the baseline for every object-oriented schema design since it exhibits the basic assumption of object-oriented modeling.

It is interesting to note that a schema which belongs to, for example, inheritance-NF, may not be necessarily in the other normal forms (e.g., minimal NF, normalized NF, or cycle-free dependency NF). Fig. 3 shows the basic relationships among these five basic normal forms. By using this framework, many desired properties of a schema design, such as validity, extensibility, minimality, normality, etc., can be well understood under the context of object-oriented models. For example, a I-NF schema promotes maximum information localization so that the impact of schema changes can largely be reduced, a M-NF schema encourages that every aspect of the requirements should only appear once in the schema in order to enhance the maintainability of the schema, while a N-NF schema avoids update anomalies and therefore when a change happens to the database, the cost for database consistency maintenance is minimized. We use the D-NF to advocate cycle-free existence dependency so that any change to the population of the database can be performed more effectively. When a schema belongs to both I-NF and M-NF, it is certainly more adaptive and robust when changes occur to either specialization or construction structure of the schema. We evaluate those schemata that satisfy all the normal forms to be of mostly higher quality regarding adaptiveness and robustness of a schema in responding to future requirement changes at (schema level and database level).

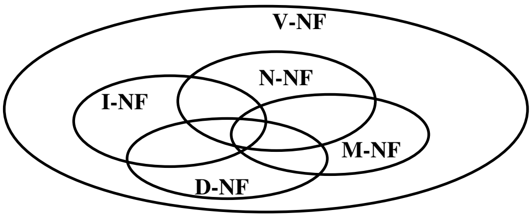

Fig. 3 Possible relationships among the five basic normal forms object-oriented schemata

9 Concluding Remarks

The increased complexity of object-oriented models necessitates the enhancement of adaptiveness of an object-oriented schema design towards changing requirements. The understanding of what properties are critical for improving schema adaptiveness becomes increasingly important in schema design and evaluation. In this paper, we have formally presented a selection of style rules for building adaptive schema in an object-oriented data and knowledge base system. Many of them can also be regarded as basic mechanisms used during design validation stage for automatic transformation of a schema to be of better quality with respect to adaptiveness and extensibility. We believe that the framework developed in this paper presents an interesting step towards overcoming the potential problems hidden in many object-oriented schema designs.

The set of style rules we have introduced is not an exhaustive list. Many issues are still pending better solutions. For instance, it is interesting to study the interactions among the given style rules and the reasoning possibility by means of the proposed axioms. Our work on the present framework continues along with several lines. On the side of practical applicability, a prototype for the proposed framework is undergoing. It initially consists of three modules: object-oriented schema style evaluator, schema optimizer, and C++ code generator based on the Demeter system™[15]. On the side of theoretical justification, in addition to the ongoing research on studying the interactions among the given style rules and the reasoning possibility by means of the proposed axioms, we are also interested in exploring the basic set of primitive operations for both object-preserving and object-extending transformations. For example, we observe that a sequence of object-preserving transformations is also an object-preserving transformation. Using this property, we may formally prove that the schema transformations to inheritance-NF and minimal-NF are both object-preserving, whereas the transformation to Normality-NF is object-extending.

The work presented in this paper proposes a design and evaluation framework and not a new model or system. It is not be expected that the proposed framework be incorporated in a given system in its entirety. Instead, the designer may select a useful subset of the proposed style rules, taking into account a particular application domain and data modeling requirement. Furthermore, as is the case with most schema design and conceptual modeling concepts, the proposed framework is based on pragmatic ground and hence no rigorous proofs of its completeness can be given. Rather, its usefulness is demonstrated by concrete examples of situations that could not be handled adequately within other existing formalisms for object-oriented schema design and evaluation.

Acknowledgment The author would like to thank Roberto Zicari for the encouragement and Karl Lieberherr for the discussions on the Demeter system.

Reference

1. Batini C. and Battista G.Di, A methodology for conceptual documentation and maintenance. Information Systems 13 (3), pp297-320 (1988)
2. Banerjee J. Kim W. Kim H.J. and Korth H., Semantics and Implementation of Schema Evolution in Object-oriented databases. Proc. of ACM SIGMOD 1987
3. Batini C., Ceri S. and Navathe S., Conceptual Database Design: an entity-relationship approach. Benjamin/Cummings Publishing Company Inc. (1992).
4. Bergstein P., Object-preserving Class Transformations. Proc. of OOPSLA 1991.
5. Bersoff E.H. and Davis A.M., Impacts of life cycle models on software configuration management. CACM 34 (8) 1991.
6. Berri C., Fagin R. and Howard J.H., Complete Axiomatization for functional multivalued dependencies in database relations. in: ACM SIGMOD 1977.
7. Cardelli L., A semantics of multiple inheritance, in: Journal of Information and Computation 76 (1988).
8. Card L. and Wegner P., On understanding types, data abstractions and polymorphism, in: ACM Computing Surveys, 17 (4), 1985.
9. Grant J., Constraint preserving and lossless database transformations. Information Systems 9 (2) pp139-146 (1984).
10. Hull R. Relative information capacity of simple relational database schemata. SIAM J. Comput. Vol.15, No.3 (1986) pp857-886.
11. Jajodia S., Ng P. and Springsteel F., The problem of equivalence for Entity-Relationship Diagrams. IEEE Transactions on Software Engineering (Sept. 1983).
12. Kim W., A model of queries for object-oriented databases. Proc. VLDB 1989.
13. Lieberherr K., Bernstein P. and Silva-Lepe I., From Objects to Classes: Algorithms for object-oriented design. Journal of Software Engineering, 6(4) 1991.
14. Lerner B.S. and Habermann A.N., Beyond Schema Evolution to Database Reorganization. Proc. of OOPSLA 1990.
15. Lieberherr K., The Art of Growing Adaptive Software: Beyond Object-oriented Software. 1994
16. Liu L., Adaptive Schmea Design and Evaluation in an Object-oriented Information System. Technical Report TR-21-95, University of Alberta.
17. Liu L. and Meersman L. Extensibility in Object-oriented database systems. SPRITE - An integrated system for technical multimedia documentation, (eds.) J. Hoppe (Springer Verlag, 1992).
18. Liu L., Zicari R., Lieberherr K. and Hürsch W., The Role of Polymorphic Reuse Mechanisms in Schema Evolution in Object-oriented Database Systems. To appear in IEEE Transactions on Knowledge and Data Engineering.
19. Penney J. and Stein J. Class Modification in the GemStone Object-oriented DBMS. Proc. of OOPLSA 1987.
20. Sheth A. and Larson J.L., Federated Databases: Architecture and Integration. ACM Computing Survey (1990).
21. Skarra A. and Zdonik S., The management of changing types in an object-oriented database. Proc. of OOPLSA 1986.
22. Ullman J.D., Principles of database systems, CS press (1980).