AN OBJECT-CENTERED DATA MODEL FOR ENGINEERING DESIGN DATABASES

Huibin Zhao
Boston University
Boston, MA 02215
zhaohb@cs.bu.edu

Alexandros Biliris
AT&T Bell Laboratories
Murray Hill, NJ 07974
biliris@research.att.com

ABSTRACT: Entities in engineering design databases need to evolve in both structure and behavior. Such a need is not well addressed by object-oriented data models based on the class concept. On the contrary, models that are based on the prototype concept allow dynamic object evolution but they do not have the abilities for object classification and strong typing supported by models that have classes. In this paper we formally define an object-oriented model which properly consolidates those two modeling approaches and can provide dynamic object evolution, flexible object classification, and strong typing.

1. INTRODUCTION

In the past few years, the advancement of database technologies has led the research and development of database systems to more advanced engineering design applications encompassing computer-aided design (CAD), computer-aided manufacturing (CAM), computer-aided engineering (CAE). and computer-aided software engineering (CASE). Unlike entities in traditional business applications, design entities usually are complex-structured and related by complicated relationships. Moreover, design entities reveal an evolutionary characteristic; from time to time, they may need to change their structure and behavior due to the iterative and exploratory nature of the engineering design process.

With the rapid development and acceptance of the object-oriented methodology in different fields of computer science, there is an ever-growing consensus in the database community that the object-oriented methodology provides a suitable paradigm for the design of advanced engineering design database systems.[1)] A major reason for this is that object-oriented data models provide suitable modeling constructs for direct representation and efficient manipulation of design entities. Despite their large number and variety, existing object-oriented data models can be classified into two broad categories: class-based and prototype-based.

Class-based models focus heavily on classes.[2, 6)] Classes that represent the structure and behavior of entity sets in the real world are defined first; instances that represent individual entities are instantiated from classes. Instances are permanently bound to their instantiating class from which they inherit both structure (attribute) and behavior (method) definitions. Consequently, all instances of a class must have a uniform structure and behavior that cannot be changed independently of the class. Although this property allows efficient representation and access of instances of a class, it inhibits instances from changing their structure or behavior on individual basis.[7, 8)]

Prototype-based models focus on objects instead.[5, 9)] In prototype-based models, classes do not exist at all; each object exists by itself and defines its own structure and behavior. An object can also have some other objects as its *prototypes* from which it inherits either definitions or states. Clearly, all forms of structural and behavioral evolution of objects can be achieved in this approach. However, by totally discarding classes, this approach loses both the object classification and strong typing capabilities supported by the class concept.

In this paper we present a new object-oriented data model, called *object-centered*, that consolidates the above two conventional approaches. As in prototype-based models, objects in our model exist by themselves and carry their own structural definitions. However, our model introduces a novel object classification mechanism which defines classes as structural constraints enforced for class membership, instead of as structural template of their instances as in class-based models. Objects can be dynamically associated with classes to inherit their behavioral definitions. Such a loose and non-permanent binding between objects and classes allows objects to change their structure and behavior dynamically in their lifetime. Still, by enforcing all its instances to have a mandatory set of attributes, a class provides a uniform and strongly-typed view of its instances in spite of the dissimilarities in their structures and behaviors. As a result, our model achieves dynamic object evolutions, flexible object classification, and strong typing, of all the benefits in either the class-based or the prototype-based modeling approach.

The rest of the paper is organized as follows. Section 2 defines the basic notions of the model, such as objects, classes, schema and database. Section 3 discusses the typing behavior of the model, in particular, the class instanceship condition and the typing semantics of classes. Section 4 discusses the object evolution mechanism of the model in details. In Section 5, we briefly describe the object versioning constructs of the model. Section 6 reviews other related work. Finally, Section 7 concludes the paper.

2. BASICS OF THE OBJECT-CENTERED DATA MODEL

2.1. Objects: Independent Entities

This subsection defines the notation of objects. All definitions are based on the existence of the following disjoint sets:

- A finite set of *basic domains* $D_1, \cdots, D_n$ for $n \geq 1$. For instance, the set of integers, the set of reals, and the set of strings.
- A finite set of *basic types* $T_1, \cdots, T_n$, one for each basic domain, i.e., the domain of T_i is D_i for $1 \leq i \leq n$.

- A countable infinite set **A** of symbols called *attribute names*.
- A countable infinite set **ID** of symbols called *object identifiers*.
- A finite set **CN** of symbols called *class names*
- A finite set **MN** of symbols called *method names*

Definition *Values*: There are four types of values:

1. Every $d \in \bigcup_{i=1}^{n} D_i$ is a *basic value*.
2. Every $id \in$ **ID** is an *object reference value*.
3. If $v_1, \cdots, v_n$ are values, then $\{v_1, \cdots, v_n\}$ is a *set value*.
4. If $a_1, \cdots, a_n$ are attribute names in **A** and $v_1, \cdots, v_n$ are values, then $[a_1: v_1, \cdots, a_n: v_n]$ is a *tuple value*.

The symbol **V** denotes the set of all values. □

An *object* in the object-centered model represents an entity in the real world. An object has a unique and immutable identifier and a collection of attributes.

Definition *Objects*: An *object* is a triple $o = (id, pro, A)$ where

- $id \in$ **ID** is the identifier of o.
- $pro \in$ **ID** is the identifier of the *prototype* object of o.
- $A = \{<a_i: v_i> \mid 1 \le i \le n\}$ is the set of (locally defined) *attributes* of o, each being a pair $<a_i: v_i>$ where $a_i \in$ **A** is its name and $v_i \in$ **V** is its value.

We use $o.id$, $o.pro$, and $o.A$ to denote the identifier, the prototype, and the set of attributes of object o. Also, we use symbol **O** to denote the set of all objects. □

As in prototype-based models, objects exist independently of any class and carry their own structural definitions and attribute values. In addition to changing their attribute values, objects can add or drop attributes from time to time.

An object may also have some other object as its *prototype*,[1] and itself is called an *extension* of its prototype. The association between a prototype and an extension defines an inheritance relationship as stated by the following delegation axiom:

> **Axiom of Delegation**: For every $o \in$ **O**, $a \in$ **A** and $v \in$ **V**, $<a: v>$ is *defined* on o iff:
>
> 1. $<a: v> \in o.A$; or
> 2. Case 1 fails and $<a: v>$ is defined on $o.pro$.

Namely, all attributes of a prototype will be inherited by its extensions unless they have been redefined in the extensions. The above delegation mechanism can be realized by forwarding all messages that access inherited attributes of an extension to its prototype. We use $o.EA$ to denote the set of *all* attributes defined on object o, either locally defined or inherited.

We now give some examples of objects. For clarity, all entities will be denoted directly using their identifiers or names in all examples of the paper.

[1] For simplicity of discussion, only single inheritance will be considered in this paper.

```
V6Engine =          ( V6Engine,
                      nil,
                      { num_cylinders:      4,
                        compression_ratio:  7.6,
                        min_octane_num:     89 } )
ToyotaStdTrans =    ( ToyotaStdTrans,
                      nil,
                      { num_gears:          5,
                        final_gear_ratio:   4.2 } )
Camry =             ( Camry,
                      nil,
                      { engine:             V6Engine,
                        transmission:       ToyotaStdTrans,
                        chief_designer:     [last: "Major", first: "John"]} )
WhiteCamry =        ( WhiteCamry,
                      Camry,
                      { chief_designer:     [last: "Smith", first: "Mark"],
                        color:              "White" } )
```

In object *Camry*, the values of attributes *engine* and *transmission* are object references to object *V6Engine* and *ToyotaStdTrans* respectively. The value of *chief_designer* is a tuple attribute consisting of two sub-attributes *last* and *first*. Object *WhiteCamry* inherits *engine* and *transmission* attributes from its prototype *Camry*, redefines the *chief_designer* attribute, and adds a new *color* attribute.

2.2. Classes: Classification by Constraints

In this subsection, we define the notion of classes.

Definition *Types*: A *type* can be one of the following:

1. A basic type T_i.
2. A class name $cn \in$ **CN** called a *class type*.
3. If t is a type, then $\{t\}$ is a type called a *set type*.
4. If $a_1, \cdots, a_n$ are attribute names in **A**, and $t_1, \cdots, t_n$ are types, then $[a_1: t_1, \cdots, a_n: t_n]$ is a type called a *tuple type*.

We denote by **T** the set of all types. □

Definition *Method Specifications*: A *method specification*, or simply a *method*, is a pair $m = (mn, s)$ where

- $mn \in$ **MN** is the name of m.
- $s = t_1 \times t_2 \times \cdots \times t_n \rightarrow t$, where $t_1, t_2, \cdots, t_n$ and t are types in **T**, is the *signature* of m.

M is used to denote the set of all methods. □

Classes model the roles that entities play in the real world. A class definition consists of a set of attribute specifications and a set of method specifications.

Definition *Classes*: A *class* is a tuple $c = (cn, sup, AS, MS)$ where

- $cn \in$ **CN** is the name of the class.
- $sup \in$ **CN** is the name of its (immediate) superclass.
- $AS = \{<a_i: t_i> \mid 1 \le i \le n\}$ is the set of (locally specified) *attribute specifications* of c, each being a pair $<a_i: t_i>$ where $a_i \in$ **A** is its name and $t_i \in$ **T** is its type.

- $MS = \{m_i \mid 1 \le i \le k\}$ is the set of (locally specified) method specifications of c.

We use **C** to denote the set of all classes. □

A class also has a set of member objects called its *instances* associated with it. All instances of a class inherit the behavior specifications – but not the structure specifications – of the class. Classes can be related by the *specialization* or *subclassing* relationship, denoted as *IS-A*(c, c'). This *IS-A* relationship between a *subclass* c and its *superclass* c' is also an inheritance relationship, as stated by the following inheritance axiom:

> **Axiom of Inheritance**: For every $c \in \mathbf{C}$, $a \in \mathbf{A}$, $t \in \mathbf{T}$ and $m \in \mathbf{M}$, $<a : t>$ (or m) is *specified* on c iff:
>
> 1. $<a : t> \in c.AS$ (or $m \in c.MS$); or
> 2. Case 1 fails and $<a : t>$ (or m) is specified on $c.sup$.

Namely, all attribute and method specifications of a superclass will be inherited by its subclasses, unless they have been re-specified in the subclasses. In the later case, the re-specified attributes or methods in a subclass must be a refinement of their corresponding ones in the superclass. Similar to the delegation mechanism, the inheritance mechanism can be realized by dynamically forwarding messages that cannot be responded locally by a subclass to its superclass. Alternatively, it can be realized by copying all inherited attribute and method specifications to the subclass to speed up schema access. Likewise, we use *c.EAS* and *c.EMS* to denote the set of *all* attribute specifications and the set of *all* method specifications of class c, either locally specified or inherited.

Following are some examples of classes:

```
VehicleEngine = ( VehicleEngine,
                  Class,
                  { num_cylinders:      int,
                    compression_ratio:  real }
                  { horse_power:        VehicleEngine → int } )
GasolineEngine = ( GasolineEngine,
                   VehicleEngine,
                   { min_octane_num:    int }
                   { ···                    } )
VehicleTransmission = ( VehicleTransmission,
                        Class,
                        { num_gears:    int },
                        { ···                } )
Vehicle =         ( Vehicle,
                    Class,
                    { engine:           VehicleEngine,
                      transmission:     VehicleTransmission }
                    { ···
                      max_speed:        Vehicle → real } )
```

The *engine* attribute specification in the *Vehicle* class specifies that all instances of *Vehicle* must have an *engine* attribute with its value being a reference to some instance of the *VehicleEngine* class; similarly for its *transmission* attribute specification. The *GasolineEngine* class is defined as a subclass of *VehicleEngine*, from which it inherits *num_cylinders* and *compression_ratio* attributes and *horse_power* method specifications. Other three classes are defined as subclasses of the system-defined root class *Class*.

2.3. Schema and Database

A database in our model can be considered as consisting of two layers: an object layer and a classification layer, see Figure 1. The *object layer* maintains the set of objects in the database and controls their creation and manipulation. The *classification layer* maintains the set of classes in the database that are used to classify objects in the object layer. The classification layer provides a uniform and strongly-typed view of objects in the database.

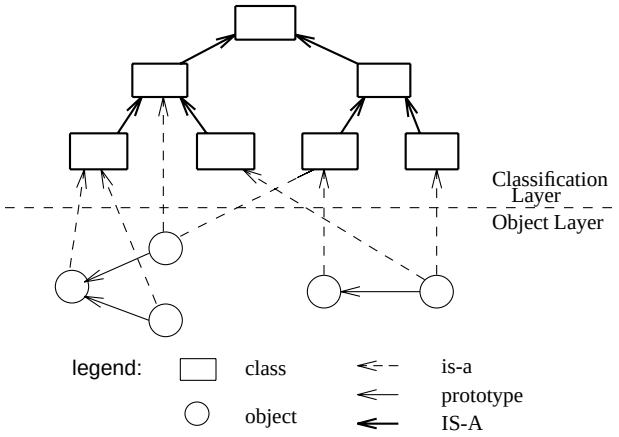

Figure 1. Two-layered database structure

Classes in the classification layer and their specialization relationships form a hierarchical structure called *class hierarchy*. The class hierarchy defines the schema of the database:

Definition *Schema*: A *schema* is a set of classes C satisfying the following constraints:

1. For all $c, c' \in C$, $c.cn \neq c'.cn$. (No two classes in the schema have the same name.)
2. For every $c \in C$, $ref(c) \subseteq C$ where $ref(c)$ denotes the set of classes referenced by class c in its definition. (All classes referenced by c are also defined in the schema.)
3. For all $c \in C$ other than *Class*, $c.sup \neq$ nil. (Every class other than *Class* has a superclass.) □

Thus a class hierarchy must be a connected tree with the system-defined class *Class* as the root.

Definition *Database*: A *database* is a pair $DB = (C, O)$ where C is a schema and O is a set of objects satisfying the following constraints:

1. For all $o, o' \in O$, $o.id \neq o'.id$.
2. For every $o \in O$, $ref(o) \subseteq O$.

Objects in O may be assigned to none, one or many classes in C at any moment. □

An imaginary vehicle design database can be defined as

```
VehicleDB = ( O = { V6Engine, ToyotaStdTrans, Camry, WhiteCamry },
              C = { VehicleEngine, GasolineEngine,
                    VehicleTransmission, Vehicle } )
```

where the objects and classes of *VehicleDB* were defined in the previous subsections.

3. CLASS INSTANCESHIPS AND TYPING RULES

This section discusses the typing behavior of our model. We first present the type inference rules of the model. Then we formally define the condition for an object to become an instance of a class. We also show that classes provide a uniform and strongly-typed view of their instances.

In our model, a type represents a set of values. The semantic assertion "value v having a type t", denoted as $v:t$, is then interpreted as v being a member of the set defined by t, and assertion "t_1 *being a subtype of* t_2", denoted as $t_1 \leq t_2$, corresponds to the mathematical set inclusion condition $t_1 \subseteq t_2$.

In the following, the *type inference rules* of the model are presented. Those rules are in the form of $\frac{X}{Y}$ where the horizontal line denotes a logical implication, i.e., if we can infer X, then we can infer Y. In those rules, a denotes an attribute name variable, v a value variable, t a type variable, o an object variable, and c a class variable. The following five rules are used to deduce the subtyping relationships among types:

(Basic) $$\frac{D_i \subseteq D_j}{T_i \leq T_j}$$

This rule states that the subtyping relationships on basic types are determined by the set inclusion relationships on their corresponding domains.

(Set) $$\frac{t_1 \leq t_2}{\{t_1\} \leq \{t_2\}}$$

This rule says that the subtyping relationships on set types are determined by the subtyping relationships on their element types.

(Tuple) $$\frac{t_1 \leq t'_1, ..., t_n \leq t'_n}{[a_1{:}t_1, \ldots, a_n{:}t_n, a_{n+1}{:}t_{n+1}, ..., a_{n+k}{:}t_{n+k}] \leq [a_1{:}t'_1, ..., a_n{:}t'_n]}$$

This rule states that a tuple type is a subtype of another only when it has at least those attributes of the supertype and with more restrictive domains.

(Class) $$\frac{IS\text{-}A(c_1, c_2)}{c_1.cn \leq c_2.cn}$$

This rule states that a subclass is also a subtype of its superclass.

(Transition) $$\frac{t_1 \leq t_2, t_2 \leq t_3}{t_1 \leq t_3}$$

This rule says that the subtyping relationship is transitive.

The next five rules are used to deduce the type(s) that a value has:

(Basic) $$\frac{v \in D_i}{v:T_i}$$

This rule states that all values in the domain of a basic type have this basic type.

(Set) $$\frac{v_1:t, ..., v_n:t}{\{v_1, ..., v_n\}:\{t\}}$$

This rule states that a set value has a set type $\{t\}$ if all its elements have type t.

(Tuple) $$\frac{v_1:t_1, ..., v_n:t_n}{[a_1{:}v_1, ..., a_n{:}v_n]:[a_1{:}t_1, ..., a_n{:}t_n]}$$

This rule states that a value of a tuple type must have at least of those attributes in the tuple type with values of corresponding types.

(Object) $$\frac{is\text{-}a(o, c)}{o.id:c.cn}$$

Here $is\text{-}a(o, c)$ denotes that object o is an instance of class c. This rules states that all instances of a class have this class type.

(Subtype) $$\frac{v: t_1, t_1 \leq t_2}{v: t_2}$$

This rule says that all values of a subtype is also values of its supertypes.

In our model, classification of objects is achieved by associating objects with certain classes. A *class assignment* operation is provided for assigning an object o to a class c. The operation will check o to see whether it satisfies the instanceship condition of c according to the following axiom:

> **Axiom of is-a**: An instance $o = (id, pro, A)$ of a class $c = (cn, sup, AS, MS)$ must satisfy the *instanceship condition* of c, formally defined as:
>
> For every $<a: t> \in c.EAS$, there exists a $<a: v> \in o.EA$ such that $v:t$.

Namely, for o to become an instance of c, it must have at least of those attributes specified in c with values of required types. Otherwise, the assignment is rejected. In *VehicleDB*, objects *V6Engine* and *ToyotaStdTrans* can be assigned to classes *GasolineEngine* and *VehicleTransmission* respectively. Afterwards, objects *Camry* and *WhiteCamry* can be assigned to class *Vehicle*.

As we can see, instances are allowed to have more attributes than those required by their class. For instance, object *Camry* has an additional *chief_designer* attribute than those required by its *Vehicle* class. Therefore, the attribute specifications of a class actually define the minimum structural requirements or constraints that instances of the class must satisfy, in contrast to serving as the structural template of its instances in class-based models. Classes thus can be used to classify objects of heterogeneous structures. Such a classification mechanism also has two other benefits.

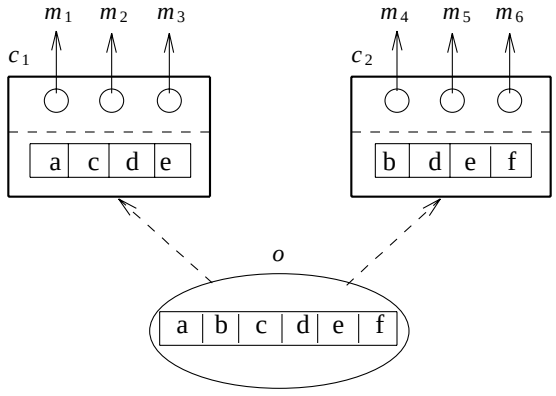

Figure 2. Multiple views of object

One benefit is that an object can be assigned to multiple classes concurrently when it satisfies instanceship conditions of all those classes. This essentially gives the capability to support multiple views of the same entity observed from different perspectives, as shown in Figure 2. In the figure, object o is assigned to both class c_1 and c_2, each defining a different view of the object: in the view defined by c_1, attributes a, c, d and e of o are visible and methods m_1, m_2 and m_3 are applicable; in the other view defined by c_2, attributes b, d, e and f of o are visible and other three different methods m_4, m_5 and m_6 are applicable instead. As in the relational system, such a view mechanism can be used to support multiple levels of abstraction, authorization, and interoperability in heterogeneous environment.

Another benefit is that the association between an object and a class can be changed dynamically. A *class de-assignment* operation is provided for removing an object from a class. Thus it is possible for an object to be assigned to a class at time $t1$, then removed from the class at $t2$, and finally assigned to a different class at $t3$. The dynamic nature of the *is-a* relationship allows an object to play a different role at different time or play multiple roles at the same time. As a result, objects can also change their behavior dynamically.

Besides for classification of objects, classes are also used for typing of objects. According to the *object* typing rule, a class c is a type with its domain being the set of all its instances. The significance of a class type is that, as speculated by the next two typing rules, it provides a uniform view of its instances which may actually have different structures and behaviors:

(Attr. Sele.) $$\frac{is\text{-}a(o, c),\ <a : t> \in c.EAS}{o.a : t}$$

This rule states that for an instance o of a class c, if c has an attribute specification named as a and typed as t, then the value of attribute a of object o must be of type t.

(Meth. Appl.)

$$\frac{is\text{-}a(o,c),\ v_2{:}t_2,...,v_n{:}t_n,\ (mn, c \times t_2 \times \cdots \times t_n \rightarrow t) \in c.EMS}{mn(o, v_2, ..., v_n){:}\ t}$$

This rule states that for an instance o of a class c, if c has a method specification with its name being mn and the types of arguments being t_2, $\cdots$, t_n and the result type being t, then an invocation of method mn on object o with arguments of required types must return a result of type t. The above two rules provide the basis for strong typing of objects as in class-based models.

4. OBJECT EVOLUTION

4.1. Object Evolution Invariants

As mentioned in previous sections, the object-centered model allows object to evolve in both their structure and behavior. In addition to changing their attribute values, objects can also add or drop attributes from time to time, or change their classes to obtain different behavior. However, if not controlled properly, such dynamic evolution of objects may result in unexpected invalidation of their instanceships. For example, for the *ToyotaStdTrans* object in the *VehicleDB* database, if it drops its *num_gears* attribute, then its previous instanceship to the *VehicleTransmission* class will be invalidated as a result of the update because *VehicleTransmission* requires all its instances to have a *num_gears* attribute. More seriously, an update to an object not only may invalidate instanceships of this updated object itself, but it may further trigger the invalidation of the instanceships of its referencing objects. Continuing the above example, for the *Camry* complex object in *VehicleDB* which refers to the *ToyotaStdTrans* object, once the instanceship of *ToyotaStdTrans* to class *VehicleTransmission* is invalidated after the update, the previous instanceship of *Camry* to the *Vehicle* class will also be invalidated consequently because the *Vehicle* class requires all its instances to have a *transmission* attribute of *VehicleTransmission* type.

The above unexpected invalidation of objects' instanceships caused by dynamic object evolution is problematic for the following two reasons. First, to maintain the type consistency of class types according to the *Axiom of is-a*, all objects whose instanceships have been invalidated must be promptly removed from corresponding classes. However, determining those invalidated objects may require substantial amount of execution time due to the cascading effect of instanceship invalidation. Second, cascading invalidation of instanceships creates a severe difficulty for static type checking of objects, because it makes practically impossible to determine the types of objects at compile time. Those reasons suggest that appropriate control should be applied to dynamic evolution of objects so that unexpected instanceship invalidation can be totally prevented.

Below we identify two *object evolution invariants* that define the necessary and sufficient condition for preventing any unexpected instanceship invalidation. The invariants are presented in the form of "$\{ P \} u_o \{ Q \}$" which states that if a precondition P holds before an update operation u is applied to an object o then a postcondition Q must become true after the update:

is-a Object Evolution Invariant:

For any object o, any class c, and any object update operation u other than the class de-assignment operation:

$$\{ is\text{-}a(o, c) \}\, u_o \{ is\text{-}a(o, c) \}$$

This invariant says that all object update operations except the class de-assignment must not invalidate any instanceship of the object they update.

as-a Object Evolution Invariant:

For any object o, any class c, and any object update operation u:

$$\{ as\text{-}a(o, c) \}\, u_o \{ as\text{-}a(o, c) \}$$

This invariant says that all object update operations, including the class de-assignment operation, must not invalidate any *as-a* relationship of the object they update. Here relation $as\text{-}a(o, c)$ denotes that object o is referenced by some other object(s) *as* an instance of class c, formally defined as (see Figure 3): there exist at least a complex object o_x and a complex class c_x such that: (1) c_x has an attribute specification, say $<a : c> \in c_x.EAS$, whose type is a reference to class c; and (2) o_x has a same named attri-

bute $<a : o> \in o_x.EA$ whose value is a reference to object o; and (3) o_x is an instance of class c_x (this implies that o must be an instance of c).

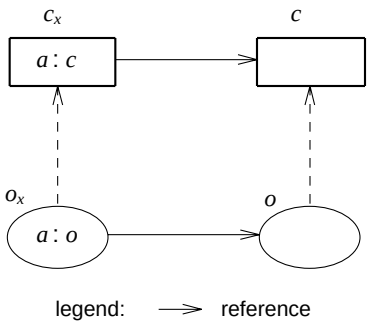

Figure 3. *as-a* (o, c) **relation**

The *is-a* invariant assures that, from the time when an object o is explicitly assigned to a class c and until it has been explicitly de-assigned from c with a class de-assignment operation, all object update operations applied to o during this period will not affect the instanceship of o to c. Furthermore, the *as-a* invariant assures that, from the time when the first reference to object o as an instance of c is made and until all such references have been removed, object o cannot be de-assigned from class c. Consequently, the instanceships of objects that refer to o will also be unaffected at any time by updates to o. Together the above two invariants guarantee that under no circumstance will the instanceships of objects be affected by object evolution operations.

4.2. Object Evolution Rules

In the previous subsection, we identified two invariants that must be preserved by object evolution operations in order to avoid unexpected instanceship invalidation. In this subsection, we present a set of *object evolution rules* that represent the strategies adopted by the model in maintaining those object evolution invariants.

Class Assignment Rule:

If an object o belongs to more than one class, then for any two classes c and c' that o belongs to, i.e., *is-a* (o, c) and *is-a* (o, c'), if $min\_type(o, a, c) \neq$ nil and $min\_type(o, a, c') \neq$ nil then $min\_type(o, a, c) = min\_type(o, a, c')$ must hold.

Here function $min\_type(o, a, c)$ denotes the minimum type that the value of attribute a of object o has to obtain in order to preserve its instanceship to class c, formally defined as:

$$min\_type(o, a, c) = \begin{cases} t & \text{if } is\text{-}a(o, c) \text{ and } <a : t> \in c.EAS \\ \text{nil} & \text{otherwise} \end{cases}$$

This rule says that for all classes which an object belongs to and require an attribute a, they must require the same type for a. This rule ensures that objects can be updated independently by each of those classes that they belong to without affecting their instanceships to other classes.

Class De-Assignment Rule:

For an object o and a class c to which o belongs, if *as-a* (o, c), then o cannot be de-assigned from c.

This rule ensures that the removal of an object's instanceship will not affect the instanceships of its referencing objects.

Attribute Deletion Rule:

For an object o and an attribute a, if there exists a class c such that $min\_type(o, a, c) \neq$ nil, then a cannot be deleted from o.

This rule says that if an attribute of an object is still required by some class to which the object belongs, then the attribute cannot be dropped from the object.

Attribute Update Rule:

For any new value v to be assigned to an attribute a of an object o, if there exists a class c such that $min\_type(o, a, c) \neq$ nil, then $v : min\_type(o, a, c)$ must hold.

This rule says that all attribute values assigned to an object must fall into the minimum types required by those classes that the object belongs to.

It is not hard to see intuitively that above object evolution rules maintain the two object evolution invariants of the last subsection. By enforcing those rules, the model allows objects to evolve without sacrificing its strong-typing capability.

5. OBJECT VERSIONING

In design systems, changes to design entities usually need to be versioned for various purposes such as documenting their evolution process or representing different design alternatives and revisions. Current object-oriented version models use *generic objects* as an abstraction to represent the version sets of versioned entities. Generic objects also support *dynamic references* used to dynamically configure complex designs. However, the typing rules for dynamic references require that a generic object must be bound to some type t all the time. As a result, in current version models versions of a generic object can only model incremental changes on a versioned entity.[4, 10]

In this section, we describe a generalized object versioning scheme that is fully integrated with the object-centered data model. We show that our versioning scheme can capture a broader spectrum of the evolution process of versioned entities than existing version models.

5.1. Generic Objects

A version of a design entity is a semantically meaningful snapshot of the entity at a point of time. For each versioned entity, a *generic object* is defined, which represents all versions – each being an object by itself – of this versioned entity. A derivation relationship, *derived-from*, is also defined among the set of versions of a generic object to capture the evolution history of the versioned entity. A generic object does not have any user-defined attribute and is not assigned to any user-defined class (though they can be assigned to some system-defined class such as *Generic* to inherit version-related operations defined on the class).

Definition *Object Versions*: An object version is a tuple $o_v = (id, pro, A, gen)$ where id, pro, and A are the identifier, prototype identifier, and attributes of o_v, and $gen \in$ **ID** is the identifier of the generic object to which o_v belongs. □

Definition *Generic Objects*: A generic object is a triple $o_g = (id, VS, VG)$ where

- $id \in \mathbf{ID}$ is the identifier of the generic object.
- $VS \in 2^{\mathbf{O}}$ is a set of object versions called the *version set* of the generic object. For every $o_v \in VS$, $(o_v).gen = id$.
- $VG = (VS, E)$ is a DAG (Directed Acyclic Graph) called *version graph* of the generic object, in which nodes represent versions in VS and edges represent the *derived-from* derivation relationships among those versions. □

From now on, the set of all objects $\mathbf{O}$ will be partitioned into three subsets, $\mathbf{O} = \mathbf{O}_u \cup \mathbf{O}_v \cup \mathbf{O}_g$, where $\mathbf{O}_u$ denotes the set of all unversioned objects, $\mathbf{O}_v$ the set of all object versions, and $\mathbf{O}_g$ the set of all generic objects.

A new object version can be created from scratch or derived from some existing one. In the later case, a new copy of the *predecessor* object version is made as the initial state of the newly derived *successor* version and a derived-from relationship is established between the two versions. After the creation of an object version, its structure and behavior as well as its attribute values can still be changed. These general forms of changes, though not allowed in current version models, are possible in our model for two reasons. First, they are allowed by the host object-centered data model, as we have described in the previous sections. Second, they are also allowed by the object versioning mechanism, because generic objects are not associated to any class type and thus do not enforce any typing constraint on their versions. Therefore, different versions of a generic object may belong to different classes and have different structures and behaviors. As a result, versions of a generic object in our model are able to capture the complete evolution process of a versioned entity.

For example, assume that the *VehicleDB* database has evolved substantially since its creation, see Figure 4. In particular, additional subclasses have been added under *VehicleEngine*, and more versions of $V6Engine_g$ have been derived.[2] Those versions of $V6Engine_g$ belong to different classes (except $V6Engine_v^2$ which does not belong to any class) and have different structures and behaviors.

5.2. Version References

Versions of a generic object can be referenced either statically or dynamically:

Definition *Static vs. Dynamic References*: A *static reference* is an object reference value $v = o_v.id$ where $o_v \in \mathbf{O}_v$, and a *dynamic reference* is an object reference value $v = o_g.id$ where $o_g \in \mathbf{O}_g$. □

Both static and dynamic version references can be used to configure complex objects, as shown in the following example:

$Lexus = (\ Lexus,$
$\quad nil,$
$\quad \{\ \text{engine:} \quad V6Engine_g,$
$\quad\ \ \text{transmission:} \quad ToyotaStdTrans_v^k\ \}\ )$

Here the value of the *engine* attribute of complex object *Lexus* is a dynamic reference to generic object $V6Engine_g$, and the value of its *transmission* attribute is a static reference to object version $ToyotaStdTrans_v^k$.

Dynamic references to generic objects are resolved into static ones at run time by *context functions*. A context function takes a generic object as the argument and returns a version of the generic object as the result. Accordingly, all dynamic references to the argument generic object are mapped into static references to the result object version. In this way, up-to-date versions of component designs can be dynamically incorporated to configure complex designs using different context functions.

An unconventional feature of dynamic references in our model is that they can also have a *selection condition* associated with them. The selection condition restricts the versions that can be mapped to by the dynamic references.

Definition *Conditioned Dynamic References*: A *conditioned dynamic reference* is a dynamic reference argumented by a class parameter: $o_g.id(c.cn)$ where $o_g \in \mathbf{O}_g$ and $c.cn \in \mathbf{CN}$, which enforces the following constraint on any context function π:

$$\pi(o_g.id(c.cn)) \in \{\ o_v^i.id \mid o_v^i \in o_g.VS \text{ and } is\text{-}a(o_v^i, c)\}$$

The class argument *c.cn* is called the *selection condition* of the conditioned dynamic reference. □

Namely, unlike a normal dynamic reference which can select any version of the referenced generic object to map to, a conditioned dynamic reference will select only among those versions that belong to the class specified by its selection condition for subsequent mapping.

Conditioned dynamic references allow complex designs to dynamically select qualified component design versions according to their configuration requirements. For instance, assume that the previous *Lexus* complex object must use a *GasolineEngine* type of engine in its design as determined by certain design rules. Then a selection condition can be added to its dynamic reference to the $V6Engine_g$ generic object, as shown below, so that it will select only those versions of $V6Engine_g$ that are of *GasolineEngine* type:

$Lexus = (\ Lexus,$
$\quad nil,$
$\quad \{\ \text{engine:} \quad V6Engine_g(GasolineEngine),$
$\quad\ \ \text{transmission:} \quad ToyotaStdTrans_v^k\ \}\ )$

Thus only version $V6Engine_v^1$ or $V6Engine_v^5$ of $VehicleEngine_g$ can be selected in a configuration of *Lexus* (see Figure 4).

Conditioned dynamic references are also beneficial for typing of dynamic references. In contrast to an ordinary dynamic reference $o_g.id$ that cannot be statically type-checked because it can be mapped to any version of o_g which belong may belong to any type, a conditioned dynamic reference $o_g.id(c.cn)$ is strongly

[2] Here $V6Engine_g$ is used to denote the generic object for $V6Engine$ designs and $V6Engine_v^i$ is used to denote its i th version.

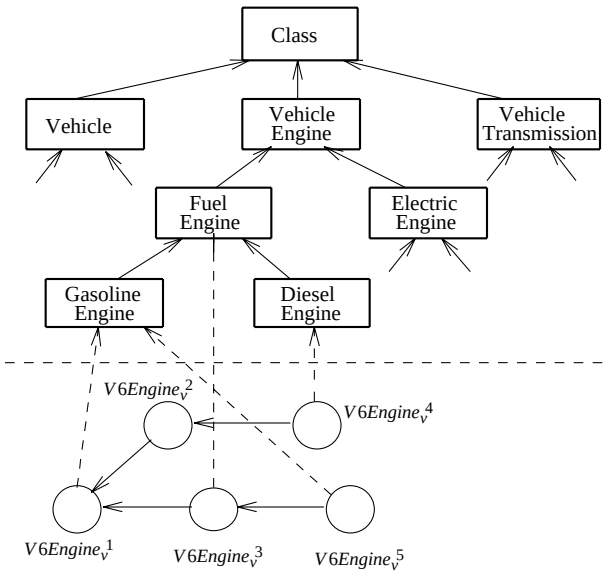

Figure 4. Versions of $V6Engine_g$

typed with type c as it can only be mapped to a version of o_g which belongs to type c. Therefore, the selection condition in conditioned dynamic references supplements generic objects with the typing information required for static type-checking of dynamic references.

6. Related Work

The object-centered model is not the first attempt on consolidating both the class-based and the prototype-based modeling approaches. There exist also other models that combine features of both types of models.

One such a model is Pegasus, an extension of the EXTRA class-based model.[3] In addition to the specialization inheritance relationship among types, a new kind of inheritance relationship, called *extension*, is added which specifies that an instance of an extension subtype must have a prototype in its extension supertype. Such an extension relationship improves the inheritance flexibility of class-based model by supporting a limited form of object-level inheritance among certain types of objects. Still, no improvement for object evolution has been achieved since all instances of a type have a uniform structure as in class-based models.

In another model Vision,[8] a real-world entity is modeled as an *object hierarchy* wherein objects represent different roles of the entity and are related by the prototyping inheritance relationships. In addition, *template hierarchies* can be defined as prototype hierarchies used for cloning instances. Although template hierarchies provide an elegant way of ensuring uniformity of newly cloned objects, their use is purely advisory and one may arbitrarily change the structure and behavior of an object after its has been cloned from a template no matter whether it is sill being referenced or not. The model, however, is not strongly-typed due to its heavy basis on the prototype-based approach.

By contrast, our model provides a more balanced combination of the two modeling approaches. Following the prototype-based approach, our model defines objects as independent structural entities that can share among them both definitions and states through the prototyping relationship. Following the class-based approach, our model uses classes to classify objects and to ensure uniformity on their instances. But by substituting the intimate and permanent binding between objects and classes with a loose and non-permanent one, our model overcomes the limitation of class-based models in allowing objects to evolve in their structure and behavior. Furthermore, by enforcing a small number of object evolution rules, our model still preserves the strong typing capability that can not be achieved in prototype-based models.

7. Conclusions

In this paper we have presented a new object-oriented data model that consolidates features of both class-based and prototype-based models. The model extends the prototype-based modeling approach with a novel classification mechanism which enables objects to be loosely and temporally associated to classes to inherit their behavioral definitions. We showed that the model not only allows object to evolve in both structure and behavior, but also supports flexible object classification and strong typing as required by advanced engineering design databases. We also showed that the model provides the basis for a generalized object versioning scheme that can capture the complete evolution process of versioned entities.

References

1) Atkinson, M., *et al*, ‘‘The object-Oriented Database System Manifesto,’’ in Proc. of 1st DOOD, 1989, pp. 40-57.

2) Banerjee, J., *et al*, ‘‘Data Model Issues for Object-Oriented Applications,’’ *ACM Trans. on Office Info. Systems*, Vol. 5, No. 1 (Jan. 1987), pp. 3-26.

3) Biliris, A., ‘‘Modeling Design Object Relationships in PEGASUS,’’ in IEEE Proc. of Int. Conf. on Data Engineering, 1990, pp. 228-237.

4) Katz, R.H, ‘‘Toward a Unified Framework for Version Modeling in Engineering Databases,’’ ACM Computing Surveys, Vol. 22, No. 4 (Dec. 1990), pp. 375-408.

5) Lieberman, H., ‘‘Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems,’’ in ACM Proc. of OOPSLA ’86, 1986, pp. 214-223.

6) Maier, D, J. Stein, A. Otis, and A. Purdy, ‘‘Development of an Object Oriented DBMS,’’ in ACM Proc. of OOPSLA ’86, 1986, pp. 472-482.

7) Richardson, J.E., and P. Schwarz, ‘‘Aspects: Extending Objects to Support Multiple, Independent Roles,’’ in ACM Proc. of SIGMOD, 1991, pp. 298-307.

8) Sciore, E., ‘‘Object Specialization,’’ ACM Transaction on Information Systems, Vol. 7, No. 2 (April 1989), pp. 103-122.

9) Stein, L.A., ‘‘Delegation is Inheritance,’’ in ACM Proc. of OOPSLA ’87, 1987, pp. 138-146.

10) Zhao, H., Biliris, A., ‘‘Integrating Object and Schema Versioning in an Object-Centered Data Model,’’ submitted for publication, 1992.