Object Evolution and Versioning in an Object-Centered Data Model

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

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

1 Introduction

Entities in engineering design databases evolve in both their structure and behavior due to the iterative and exploratory nature of engineering design process. Changes to design entities need to be versioned to document their evolution process and to represent different design alternatives and revisions.

Existing object-oriented data models do succeed in modeling the complex structures and complicated relationships of design entities. However, they show weakness in modeling the dynamic aspects of design objects: instances that represent design entities are permanently bound to their instantiating class, from which they inherit structure and behavior definitions. Consequently, all instances of a class must have a uniform structure and behavior that can not be changed on an object-by-object basis.

Our research investigates an alternative model, called *object-centered*, to support flexible object evolution and versioning. This paper discusses the object evolution and object versioning schemes of the object-centered model. We show that the novel classification mechanism of the model based on a loose and temporary binding between objects and classes not only enables objects to evolve dynamically in both structure and behavior but also provides the basis for a generalized object versioning scheme that can capture the complete evolution process of versioned entities.

2 The Object-Centered Data Model

Objects in the model are defined as structural entities that exist independently of any class. An *object* is a pair $o = (id, A)$ where id is the *object identifier* and $A = \{< a_i : v_i > | \ 1 \leq i \leq n\}$ is a set of *attributes*. Here a_i denotes an attribute name and v_i denotes its value. Attribute values may be either of type *integer*, *real*, *boolean*, *string*, or object identifiers which are used to reference to other objects. Objects carry their own structural definition and they can add or drop attributes or change their attribute values from time to time.

Classes are defined as structural constraints enforced for instanceship instead of as structural template of their instances. A *class* is a triple $c = (cn, AS, MS)$ where cn is the class name, $AS = \{< a_i : t_i > | \ 1 \leq i \leq m\}$ is a set of *attribute specifications* and $MS = \{(mn_i, t_{i1} \times \ldots \times t_{il} \rightarrow t_i) \mid 1 \leq i \leq n\}$ is a set of *method specifications*. Here a_i and mn_i denote the attribute and method name respectively, and t_i, t_{i1}, ... and t_{il} denote either system-defined primitive types or user-defined class types. A class has a set of objects called its *instances* associated with it and they inherit all the operations (methods) of this class. The set of attribute specifications $c.AS$ of a class c defines the *instanceship condition* for an object o to become an instance of c: for every $< a : t > \in c.AS$, there exists a $< a : v > \in o.A$ such that value v has type t.

As instances of a class are allowed to have additional attributes than those required by the class, classes can be used to classify objects of heterogeneous structures. An object can also be assigned to multiple classes concurrently when it satisfies the instanceship conditions of these classes. Such a capacity can be used to support multiple views of an entity observed from different perspectives. Moreover, the association between an object and a class can be changed dynamically. It is possible for an object to be assigned to a class and then de-assigned (removed) from it sometime later. This capability allows objects to dynamically change their behaviors in their lifetime.

In sum, the class extent is a heterogeneous collection of objects. The loose and temporary binding between objects and classes enables objects to evolve dynamically in both structure and behavior. However, by enforcing all instances to have a mandatory set of attributes, a class still provides a uniform and strongly-typed interface for its instances.

*This work was done while the author was with the Boston University.

3 Object Evolution

If not controlled properly, dynamic evolution of objects may result in unexpected invalidation of instanceships of objects to classes that they previously belong to. Unexpected instanceship invalidation creates a substantial difficulty for static type checking of objects. For this reason, the following *object evolution rules* are guarantee to prevent typing problems during the object evolution process.

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, if $min\_type(o,\ a,\ c) \neq nil$ or $min\_type(o,\ a,\ c') \neq nil$, then $min\_type(o,\ a,\ c) = min\_type(o,\ a,\ c')$ must hold.

Here function $min\_type(o,\ a,\ c)$ denotes the minimum (most general) 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

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

This rule says that for all classes that an object belongs to and require an attribute a, they must require the same type for a. This rule assures that objects can be updated independently of the classes 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 o is currently being referenced by some other objects as an instance of class c, then o cannot be de-assigned from c.

This rule ensures that the removal of an object's instanceship will not affect the instanceship 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 must have type $min\_type(o, a, c)$.

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

4 Object Versioning

Current version models use *generic objects* as an abstraction to represent the version set of versioned entities. Generic objects also support *dynamic version 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 and all versions of the generic object must be of type T or a subtype of T. As a result, versions of the generic object can only model incremental changes of the versioned entity.

To overcome these modeling limitations of generic objects, our model removes the association of a generic object with some (user-defined) class and versions of a generic object are allowed to belong to different classes. As a result, all forms of changes on object versions are allowed and the complete evolution process of versioned entities can be captured by their versions.

For a dynamic reference R_o to a generic object o, a *selection condition* c, where c specifies a class (type), can be specified to restrain the versions to be selected by the reference. Unlike the ordinary dynamic reference R_o that can later be resolved into a *static* reference to any version of o, the *conditioned* dynamic reference $R_o(c)$ can be resolved into a static reference to only some version of o which is also an instance of class c.

Conditioned dynamic references bring two advantages. First, they can be used to select only qualified component design versions with desirable behavior in the configuration definition of complex designs. For instance, assume that the $V6Engine$ component design object has many design versions of different types, such as $GasolineEngine$ or $DieselEngine$ or $ElectricEngine$ types. Then for a $TurboLexus$ complex design that must use a $DieselEngine$ type of engine, a conditioned dynamic reference $R_{V6Engine}(DieselEngine)$ can be made to select only those versions of $V6Engine$ that are of $DieselEngine$ type. Second, in contrast to the ordinary dynamic reference R_o that cannot be statically type-checked because it can be mapped to any version of o which may belong to any type, the conditioned dynamic reference $R_o(c)$ is strongly-typed with type c as it can only be mapped to a version of o 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.