A Data Model for Engineering Design Objects

Alexandros Biliris
Computer Science Department
Boston University
Boston, MA 02215
Email: biliris@bu-cs.bu.edu

ACM, IEEE, International Conference on Data and Knowledge Systems for Maunfacturing and Engineering, Gaithersburg, Maryland, October 1989, pp.49-56

Abstract

Design objects in CAD applications have versions and participate in the construction of other more complex design objects. In this paper we describe data model aspects of an experimental database system for CAD applications, called Pegasus. Our model is based on previously published work on extensible and object-oriented database systems. The novel idea of Pegasus that is presented in this paper is the reconciliation of two subtyping (inheritance) mechanisms: the first, called refinement, is based on the usual semantics of schema copying and the second, called extension, is based on the inheritance semantics between prototypes and their extensions. We use these modeling elements to show how generic and version objects as well as component occurrences of (generic or version) components can be modeled.

1. Introduction

A CAD product is an aggregation of design objects and associated documents. For example, a product in a mechanical application is associated with a variety of drawings such as detail and assembly drawings, exploded assembly drawings, etc. These design objects are frequently called *representations*, [Katz87]. A design object may have many *versions*. Each version represents a particular description of the design object as it has been defined by a user at some point in time. The object that represents all versions of the (semantically) same object is called *generic*, [Ditt88]; it keeps data about its versions, their relationships, and their common properties. A version object is associated with exactly one generic object. Versions of a single generic object x form what is frequently called the *version set* of x.

A design object may reference (consist of) a number of other objects that in turn may be the constituents of other objects. Such objects that are composed of other objects are called *complex*, [Lori83], or *composite*, [Bane87], the hierarchical composition of a complex object is referred to as the *configuration hierarchy* of the object, and the constituent objects are called *components*. For example, the design item Car (a composite object) consists of a frame, wheels, doors, seats, etc., which are the component objects of Car. Of course, a component object (generic or version) is by itself another design object and as such it may have its own components.

Each reference by a complex object to one of its components corresponds to a *component occurrence*[1] of the referenced object. For example, each of the four wheels that are components of the design item Car, correspond to four different component occurrences of the same single design object Wheel. Figure 1 shows two generic design objects, Wheel and Frame; currently, there are three versions of Wheel and two versions of Frame available to engineers to design a car. The figure shows four component occurrences of Wheel and one of Frame. Each component occurrence describes how the component object participates in the composite object. In this example, attributes of component occurrences of wheels may include a name (e.g., left front wheel), and a transformation matrix that describes where the component is located relative to the composite object. As we may see from the figure, the designer of the car has chosen version v1 of Frame to be used in his design. On the other hand, component occurrences of Wheel reference the generic object. A *generic reference* leaves the exact version unspecified. Generic references provide the means to an engineer to postpone specific decisions about non essential details for a later time. They also provide the means to reference an object x that is itself under development, perhaps by another group of engineers, without prematurely binding this reference to a specific version of x that happens to be currently available (if there is one available at all). This is important because in large design projects different users edit different components concurrently.

Briefly, the minimum requirements of a database management system for design applications include the following:

[1] In [Bato85] a component occurrence is called *component instance*. We avoided the word *instance* since it is used in a different context in object-oriented languages.

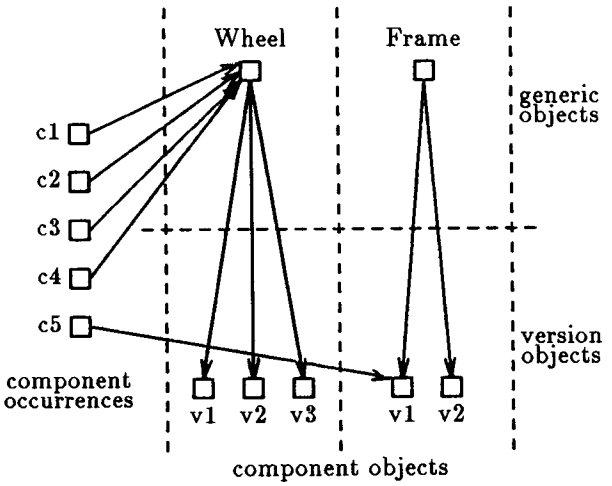

Figure 1. Generic and version objects of design objects Wheel and Frame. Each component occurrence describes specific information related to the participation of the above design objects in the composite object Car (not shown).

- It should support the design process. This includes support for design project decomposition, synchronization and documentation. The first is needed because large projects are always broken down to smaller pieces and then assigned to different (group of) designers. Since the design proceeds in parallel by different users in different subprojects, there is a need to synchronize this activity. Finally, the whole design process must be well documented so that a user can go back in time and see how a particular design was evolved.
- It should provide data modeling elements so that design objects and their relationships can be naturally mapped into the database. This includes support for objects of arbitrary complexity and size. It should allow declaration and manipulation of generic and version objects in a way that is appropriate for the application; thus, the mechanisms that implement generic or version objects should not be somehow build-in in the system. The mapping of generic references to a specific version should be under user control (when the mapping should be done and to which version). Sharing and re-use of components stored in a design library or produced by other designers should be allowed and encouraged.
- It should provide a computing environment consisting of workstations and server(s) in which the performance of the whole system is acceptable by the designers.

1.1. Related Work

Several approaches have been proposed to handle the above requirements. Early work includes extensions of relational systems to support complex objects and the design process, [Hask82, Lori83, Bato85]. However, relational systems have been criticized, for a long period of time, for lack of modeling elements that can efficiently capture the complex relationships in a CAD application, [Atwo85, Land86]. Object-oriented and extensible database systems are emerging as a very promising solution to handle CAD applications because they greatly surpass conventional record-based systems in modeling power and performance. Included among the new modeling elements are support for arbitrarily complex objects with no constraints on structure or size, object reference, type inheritance, derived attributes (operations), and others. Since, the most common operations of CAD tools are navigation among object relationships and simple retrieval of design objects, the first two modeling elements, in particular, can substantially improve performance; the reason is that costly, value-based, joins that are necessary in relational systems to pull out the pieces of the design of interest can be mostly eliminated. A survey of the above modeling constructs can be found in [Hull87], and a collection of papers in object orientedness can be found in a recently published book, [Kim89]. Finally, a comprehensive list of why object-oriented systems can deliver the performance that conventional systems did not is given in [Maie89].

Specific approaches to support complex objects and/or versions have also been proposed [Depp86, Katz87, Kim87, Rehm88, Beec88, Nara88, Wilk89]. In [Katz87] a version server is described which explicitly supports configuration, version derivation, and equivalence relationships; it runs on top of a file system in which design objects are stored. On the contrary, the schemes for versioning described in [Kim87, Rehm88, Beec88, Nara88], and for composite objects described in [Depp86, Kim87, Rehm88, Wilk89] are integrated into their database model.

The notion of *context* has been proposed to resolve references to generic objects, [Ditt88, Chou88, Beec88]. A context is a function that maps *generic* references to an object x into *specific* versions of x. The set of contexts currently active in a particular workspace defines the *environment* in which designs are materialized in this workspace. A user can establish a number of environments in his workspace and, by switching from one environment to another, can see various alternative configurations of the design version. Alternatively, generic references to components can be resolved based on a default version supplied by either the referenced generic object itself or the referencing composite object.

Mechanisms for design synchronization are presented in [Lori83, Kim84, Klah85, Banc85, Eckl87, Horn87, Rehm88, Pu88, Bili89a]. Some of these proposals introduce new transaction models [Kim84, Klah85, Banc85, Horn87, Pu88], while others are complimentary to existing transaction systems [Lori83, Rehm88, Bili89a]. Issues related to the interaction of *time* versions and *design* versions with respect to the configuration management of design objects are discussed in [Bili89c].

1.2. Outline of our Work

The data model design of Pegasus is presented in **Section 2**. Subtyping, discussed in **section 2.1**, can be defined either as a *refinement* or as *extension*. The first, is based on the usual semantics of schema copying and the second is based on the inheritance semantics between prototypes and their extensions, i.e., sharing of object behavior and state. **Section 2.2** compares the inheritance mechanism of other data models with ours. For a discussion of other modeling elements of Pegasus, such as support for relationships and replication of relationship attribute values in the participant objects, the reader is referred to [Bili89b].

Section 3 describes how the above modeling elements can be used by the user to define generic and version objects as well as component occurrences. Versions of generic objects are modeled as extensions of their generic object and component occurrences are modeled as extensions of the object (generic or version) that participates in the composite object. Finally, **Section 4** presents our conclusions.

2. Data Model

Pegasus starts with the modeling elements provided in EXTRA data model, [Care88]. We selected this model, as our starting point, because of its clarity in all of its basic modeling concepts. We will try to keep the discussion brief on modeling elements that are similar to EXTRA, and elaborate on those issues that present some difference, mainly in the subsequent section where we discuss the type hierarchy.

The database is a collection of (database) objects, each identified by a system generated id, called *oid*. Every object is of a given type; it is said to be an instance of that type. The type provides the attribute specifications shared by all of its instances, the operations that can be performed on these instances, and the constraints that attribute values of these instances must satisfy. The state of an object is given by the values of the attributes at any point in time. The attributes, operations and constraints of an object are collectively called the *properties* of the object. As an example, the types Employee and Department are defined in Figure 2[2].

```
define type Employee:
(
      ssn:        int4;
      name:       (fname: char[20]; lname: char[20]);
      dob:        Date;
      salary:     int4;
      worksIn:    ref Department, inverseof emps;
);
operations:
      age         returns int4;
      manager     returns Employee;
end Employee;

define type Department:
(
      name:       char[20];
      manager:    ref Employee;
      numemps:    int4;
      emps:       {ref Employee}, inverseof worksIn;
);
constraints:
      c1:   emps.count >= 1;
end Department;

create Emps1, Emps2:      { own ref Employee};
create Depts1, Depts2:    { own ref Department};
```

Figure 2. A simple database schema.

Each attribute specification defines the type of the attribute value. Primitive types (integers, characters, etc.), user defined abstract data types (such as Date in the Employee definition), type constructors (tuples, arrays, and sets) and the three kinds of attribute value semantics (**own**, **ref**, and **own ref**) were taken directly from EXTRA, [Care88]. An **own** attribute, which is the default, is simply a value; it lacks object identity [Khos86]. A **ref** attribute is a reference to another object in the database. Finally, an **own ref** is a reference to an object with the additional semantics of that the referenced object is deleted when the referencing object is deleted. Thus an object can not be referenced through an **own ref** attribute by more than one object. Instances of a given type are classified into user-maintained sets; i.e., the definition of a type and the classification of the instances of this type are separated from each other. For example, the schema of Figure 2 creates two sets of Employee objects, Emps1 and Emps2, and two sets of Department objects, Depts1 and Depts2.

[2] The pairs "(" ")", "{" "}", and "[" "]" denote tuple-, set-, and array-objects, respectively.

Attribute values and operations defined for an object are accessed using the dot notation (regardless of the type of the accessed attribute, see [Care88]). For example, the 'manager' function of Employee is implemented as "this.worksIn.manager", where *this* is a special variable implicitly bound to the object to which the function is applied. Inclusion of operation and constraint definitions with the types in Figure 2 is rather cosmetic, since operations and constraints can be defined after the definition of the types, as in [Care88] for operations.

Constraints defined in a type T, specify a predicate (and optionally a name) that should be satisfied for all instances of T in the database. For example, c1, in Figure 2, specifies that for every instance d of Department, d.emps should have at least one member. A constraint can also be defined on individual objects to strengthen the type level constraints (if they exist at all). For example, the following constraint specifies that members of Depts1 should be large departments only (with more than 100 employees).

constraint c1: Depts1.emps.count > 100

The **inverseof** clause that appears in the example schema defines a many-one binary relationship between instances of Employee and Department objects. Pegasus will enforce the following *referential integrity* constraint:

> for each Employee e, Department d, d is the value of e.worksIn, if and only if e is a member of d.emps.

Binary as well as higher level relationships and relationship types, roles, and replication are discussed in length in [Bili89b].

2.1. Type Hierarchy

Types in Pegasus, as in all object-oriented systems, are related with the supertype/subtype relationship. This relationship among types form the type hierarchy. where, an edge from a type T1 to T2 indicates that T2 is a subtype of T1 (and T1 a supertype of T2). Although, all object-oriented systems that we are aware of maintain some kind of type hierarchy, the semantics of inheritance implied by this hierarchy is not the same. We discuss the two inheritance mechanisms provided by Pegasus, namely *refinement* and *extension*, and at the end of this section we compare them with inheritance mechanisms of other systems.

When T2 **refines** T1, the property definitions of T1 are applicable to T2 and a T2 object maintains its own storage for all values of all attributes defined in T2 and T1. In this kind of subtyping, a real world object is mapped to exactly one database object that is an instance of the object's most refined type. This is shown in Figure 3; undirected lines indicate the type of an object. Instances of T1 and T2 are independent from each other, e.g., updating an instance of T1 does not affect any instance of T2, and vice versa.

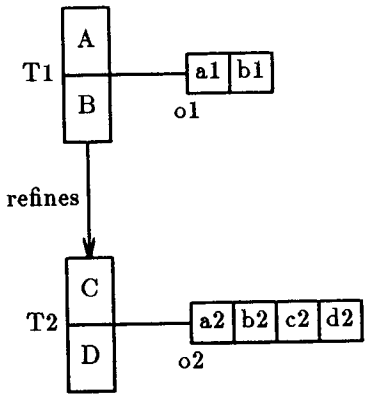

Figure 3. T2 refines T1; instances of T1 and T2 are independent from each other.

When a type T2 **extends** T1 then the following existence constraint is implied: for every instance o2 of T2 there exists an instance o1 of T1 whose attribute values are inherited by o2. The object o1 whose values are inherited by o2 is called the *prototype* of o2, and the dependent instance o2 is called an *extension* of its prototype. Subtyping through extension is illustrated in Figure 4. There is always a special link from an extension object to its prototype, called *proto*-link. In general, inheritance works as follows: if x.P is requested from an object x of type T and P is not defined in T, searching for P will be forwarded to the object referenced in x.proto and recursively to x.proto.proto..., until we find an object for which P is defined. The prototype of an extension object should be specified when the extension is created in the database.

In both cases, a subtype T2 may add more properties and it may also strengthen the properties defined in its supertype. Thus, an instance of T2 can be used anywhere an instance of T2's supertype is used.

The type hierarchy is a DAG and therefore it is possible for a type to inherit properties from several supertypes (*multiple inheritance*). Figure 5 shows a type T4 that refines T1 and T2, and it extends T3. It also shows two objects of T4, o4 and o5, to share the values of the same T3 object. Checking for conflicting definitions in the supertypes is done in the same way for refinement and extension, and no attempt is made to automatically resolve them. If conflicting definitions are found, the system will

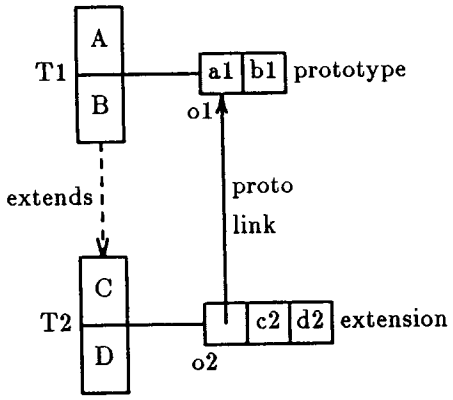

Figure 4. T2 extends T1; for every instance o2 of T2 there exists an instance o1 of T1 whose attribute values are inherited by o2.

reject the operation. Notice that the proto-link of o2, in Figure 5, may actually reference any object whose type, say T3, is a subtype of T1, and an inheritance conflict may arise. For example, attribute C may have also been defined in T3. This kind of conflict, however, does not create any problem because searching for an attribute value always starts from the current object, in this case o2.

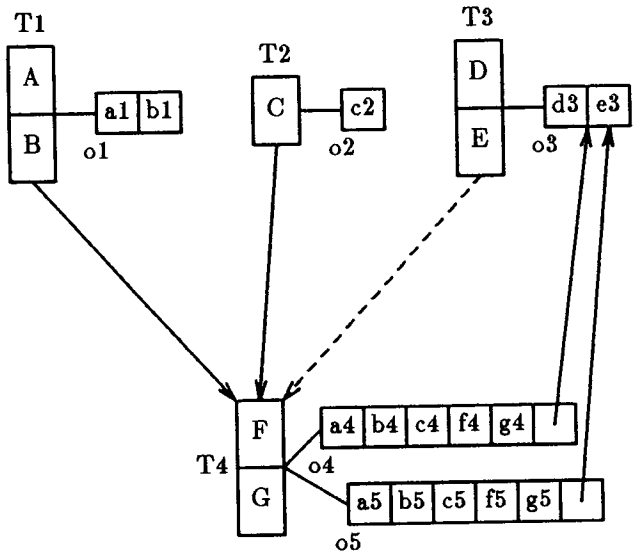

Figure 5. T4 refines T1 and T2, and it extends T3

From the application's point of view there is no difference whether subtyping is realized through refinement or extension. In both cases, if T2 is a subtype (refinement or extension) of T1, and T1 includes the attribute definition A, the application can request the value of A from an instance of T2. The difference in the semantics between the two kinds of inheritance is simply in the implementation of inheritance and it is the one between schema *copying* and value *sharing*.

As an example, the following schema defines the Employee type as a refinement of Person:

```
define type Person:
(
      ssn:        int4;
      name:       (fname: char[20]; lname: char[20]);
      dob:        Date;
);
operations:
      age         returns int4;
end Person;

define type Employee refines Person:
(
      salary:     int4;
      worksIn:    ref Department;
);
operations:
      manager     returns Employee;
end Employee;
```

Each Employee object will store the values of the three attributes defined in Person and the two attributes defined in Employee. The function 'age' applied to an Employee object e will find the 'dob' value of e in e itself.

To denote subtyping through extension, we introduce an additional kind of attribute value semantics, called **extends**. An **extends** attribute is like a **ref** attribute which carries the additional value inheritance semantics discussed previously. The definition of Employee as an extension of Person is as follows[3]:

```
define type Employee:
(
      person:     extends Person;
      salary:     int4;
      worksIn:    ref Department;
);
operations:
      manager     returns Employee;
end Employee;
```

Each Employee object will store the values of the three attributes defined in Employee only, and it will

[3] An alternative type definition syntax for **extends** would be like the one we used for **refines**; i.e., "**define type** T2 **extends** T1: (...); **end** T2;", and have the system add a "proto: **extends** T1;" attribute definition in T2. We have chosen the first approach because we may want to associate additional information with the proto-link, such as a possible participation in a relationship.

share the attribute values of the object that is referenced in 'person'. In this case, to evaluate the 'age' function, we must visit e.person. Notice, that e.person may actually reference an object of any type that is a subtype of Person, e.g., it may reference another Employee object.

A question that is raising because of the existence of the two mechanisms is when someone should use refinement and when extension. If we look at this question from a performance point of view, the answer is refinement because the entire state of an object is packed together in one place. With extension, once the object of interest is brought in main memory, we may need additional disk accesses to bring the prototype, its prototype, etc. Clustering the prototype object with the extension, as a general policy, does not make sense because the prototype may have many extensions. The drawback of refinement is when we want to formulate queries on all objects that logically belong to the same type T, because we must first union all sets that have instances of T and all of T's subtypes; e.g., to find the names of all persons we must first find all Person objects, then all Employee objects, etc. In the examples that we present here and in [Bili89b], we use extension only when it is needed, this is the case when we want to model objects whose state *must* be shared by many other objects. In any other case, we use refinement.

2.2. Discussion

The terms *prototype* and *extension* appear in actor languages, [Lieb86, Unga87], where an (extension) object is created by specifying a list of its prototypes as well as local behavior idiosyncratic to the object itself. Inheritance is performed by a process called *delegation*; when an extension object receives a message that can not respond to, it forwards (*delegates*) the message to one of its prototypes. However, the similarities with our system stop here. In these languages no types are associated with objects. There is no way to determine whether an object x can respond to a message unless the message itself is delegated to its prototype (and recursively to the prototype' prototype, etc.). Data models based on the idea of prototypes include [Nara88] and [Caru87] where attributes can be added to an object after its creation. On the contrary, in our model the type of an object x determines its fixed structure and whether x is an extension object as well as the type of its prototype. Thus, some error checking and optimization techniques can be applied at compile time, and storage clustering becomes feasible.

Inheritance through refinement is exactly the kind of inheritance provided in some extensible and object-oriented systems such as Postgres [Rowe87a], ORION [Bane87], GemStone [Maie86], and EXTRA [Care88]. On the contrary, in other systems such as TAXIS [Mylo80], GALILEO [Alba85], and Iris [Lyng86] a real world object is mapped to a number of database objects each being an instance of the type that this object (logically) possesses. For instance assume that both Student and Employee are subtypes of Person. Then a given Student x is mapped to two database objects: s, an instance of Student, and p, an instance of Person, each representing the two different roles of the same object x. If later on, this student becomes a TA a third database object e will be created as an instance of type Employee to represent the role of x as an employee. Database objects s and e will share the values of p; e.g., if the value of the 'name' attribute in p is changed, the new value is inherited by both s and e. Although, this mechanism does provide a notion of sharing, instances can not be individually addressed by the user. In other words, sharing is permitted only among database objects that represent different roles of the same real world object.

Therefore, in both of the two classes of systems we described above, it would be impossible to represent a situation where many instances of the same type inherit the values of a single object. In the next section, we will see that this is precisely the relationship that we want to capture between a generic object and its version as well as between component occurrences and (generic or version) components.

3. Modeling Design Objects

Version and generic objects are treated as any other ordinary object and as such they are of a given type. The type of the generic object is not the same as the type of its versions. A user can independently define the attributes that a generic object must have and the attributes of the versions of this generic object. The attribute values of a generic object are shared by all of its versions. For example, if an AdderChip (a generic object) has four pins, then all its versions have four pins too. This suggests that version objects should be defined as extensions of their generic object. The system provides two types: Generic and Version to capture the common behavior of generic and version instances, respectively. The mechanism of using these types to define new user types is discussed shortly. We first provide the definition of type Generic shown in Figure 6.

```
define type Generic :
(
    defaultVersion:   ref Version;
    nextVid:          int4;
    vcount:           int4;
    vd:               array[] of own ref VDescriptor;
);
operations:
    lastVersion              returns Version;
    lastVersions             returns {Version};
    versionSet               returns {Version};
    version(vid: int4)       returns Version;
    creationDate(vid: int4)  returns Date;
    home(vid: int4)          returns Database;
    status(vid: int4)        returns VersionStatus;
    successors(vid: int4)    returns {Version};
    successorsAll(vid: int4) returns {Version};
    predecessors(vid: int4)  returns Version;
    /* etc */
end Generic;
```

Figure 6. Definition of type Generic.

The 'defaultVersion' determines the version that should be chosen when a generic reference (a reference to an instance of type Generic) is mapped to a specific version. The 'nextVid', which is zero initially, is the version id that will be assigned to the next version of the object that will be created, and 'vcount' is the number of versions of the generic object. Finally, 'vd' is an array of version descriptors, one for each version that has been created. The definition of VDescriptor follows.

```
define type VDescriptor :
(
    version:   own ref Version;
    created:   Date;
    home:      ref Database;
    status:    VersionStatus;
    succ:      {ref VDescriptor}, inverseof pred;
    pred:      ref VDescriptor, inverseof succ;
);
end VDescriptor;
```

Attribute 'version' references the actual version object, and 'created', 'home', and 'status' indicate, respectively, the version creation date, the current home database in which the version resides and its status which can be one of the following: in-progress, stable, frozen, or released, [Bili89c]. Finally, 'succ' is a set of references to the descriptors of those versions that have been derived by this one and 'pred' is a reference to the one from which this version was derived from.

Operations defined in Generic provide data associated with versions if the access is done through a generic object (similar operations are defined for version objects). For example, the function home(vid: int4) returns "this.vd[vid].home", and the function versionSet is implemented as "**range of** descriptor **is** this.vd **retrieve** descriptor.version".

Types Generic and VDescriptor are not built-in, in the sense that they can be enriched to fit the application needs at a particular installation. One can add attributes in VDescriptor to show whether the version has been archived and when, the date that the version was installed in the shared system, the creator of the version, a list of approvals, etc. Similarly, users at a particular installation may need the capability to choose a subtree of the version derivation hierarchy to be the *active* derivation subtree. This means that nodes of this subtree would be the only ones from which new versions can be derived thus, effectively, blocking out all other versions in the hierarchy. If such need exists, we can define an attribute 'activeRoot' in Generic to indicate the root version of the active subtree. Once, however, these types have been fixed all applications running in that installation use the same type definitions.

Figure 7 shows the definition of the type Version with 'generic' being the proto-link to a version's generic object.

```
define type Version:
(
    generic:   extends Generic;
    vid:       int4;
);
operations:
    creationDate     returns Date;
    home             returns Database;
    status           returns VersionStatus;
    successors       returns {Version};
    successorsAll    returns {Version};
    Predecessors     returns Version;
    ...
end Version;
```

Figure 7. Definition of type Version.

The value of 'vid' provides the version number of the version. Some of the operations in Version are implemented by performing the corresponding operations on the version's generic object. For example, the function 'home' may be implemented as "this.generic.home(this.vid)". Notice that all operations in Version are overloaded because Version inherits the operations defined in Generic. Thus, if 'home' is applied to a version instance v with no arguments, i.e., v.home, the above piece of code will be executed and the home database of v will be returned. If, however, the function is called on v with a single integer argument, e.g., v.home(5), the system will delegate this operation to the version prototype, i.e., its generic object. Thus,

"this.generic.home(5)" will be executed which returns the home of version 5 of v's generic object.

3.1. Versionable Types

Now, let us turn our attention on how a user can proceed to define a versionable type T. Assume that G-properties are the properties that we want to associate with generic objects and V-properties the properties that we want to associate with versions of these generic objects. The G-properties of a generic object x represent the common characteristics of all versions of x. They can be though of as forming the interface of x's versions. The V-properties represent deviations in the defined characteristics of x. The variation that each version possesses is considered to be small enough to constitute valid membership in the version set of a generic object. Although, determination of G-properties and V-properties is application specific we can say that, in general, the V-properties are those properties that do not change the form, fit, and function of the generic object (i.e., all versions are interchangeable within a higher level assembly). For example, assume a generic object 2bitAdder and that all versions of this item must have a certain number of pins. Then, this information should be associated with 2bitAdder itself; i.e., versions of 2bitAdder must have this specific number of pins or otherwise, they can not be considered to be versions of 2bitAdder. The definition of a type T whose instances are generic objects looks like this:

```
define type T refines Generic:
    G-properties
end T;
```

Each (generic) instance of T maintains its own state for the attributes defined in Generic and T, not shared by any other instance of T. Any type that refines Generic is called *versionable*. For any versionable type T there should be a type, TVersion, that defines versions of T objects. The definition of TVersion is going to look like this:

```
define type TVersion refines Version:
    generic:    extends T;
    V-properties
end TVersion;
```

A TVersion instance inherits the 'vid' attribute from Version. However, the 'generic' proto-link definition is strengthened to be a reference to an instance of T rather than just a reference to an instance of Generic. Thus instances of TVersion will share the properties of one of T's instances.

As we described in the introduction, a component occurrence indicates how a subordinate object participates in another more complex object. A component occurrence of an object x (generic or version) is distinct from other component occurrences of the same object x, and each shares the same x properties. They may also have attributes that are not inherited such as a name and a location. Type TOccurrence will look like the following definition, where O-properties are the component occurrence specific properties.

```
define type TOccurrence:
    occurrenceOf:    extends T;
    O-properties;
end TOccurrence;
```

The resulting type hierarchy after the definition of the above types and an example of instances of these types are shown in Figure 8a,b. Type TOccurrence **extends** T through the 'occurrenceOf' reference. Thus, a particular component occurrence can reference and share the values of a generic object instance of T, or one of its versions of type TVersion. Note that from a user point of view a component occurrence is seen as a copy of the referenced component and the user can request any property defined for that component. If the 'occurrenceOf' attribute value did not carry the semantics of value inheritance, it would be an ordinary reference and all functions defined in TVersion (including the ones inherited from T and Version) would have to be explicitly redefined for TOccurrence objects.

A specific mechanical design example that involves simple (non assembled) parts as well as assemblies is presented in [Bili89b].

4. Conclusion

In this paper we have presented the dual subtyping mechanism of Pegasus, an object-oriented model targeted to support CAD applications. We have shown that this simple modeling element is powerful enough to enable users to define generic and version objects in a way that fits the particular needs of their application. This is unlike other version schemes where generic objects are somehow build-in in the system, [Chou88].

The modeling of a component occurrence of a (generic or version) component as extension of the component object itself, allows CAD applications to see this occurrence as a whole copy of the referenced component. This is true even if multiple designs are using the same component. In contrast, in some systems component objects are considered to be exclusively owned by the composite object to which they are components, [Bane87, Lori83]. This introduces a number of problems. Specifically, it does not

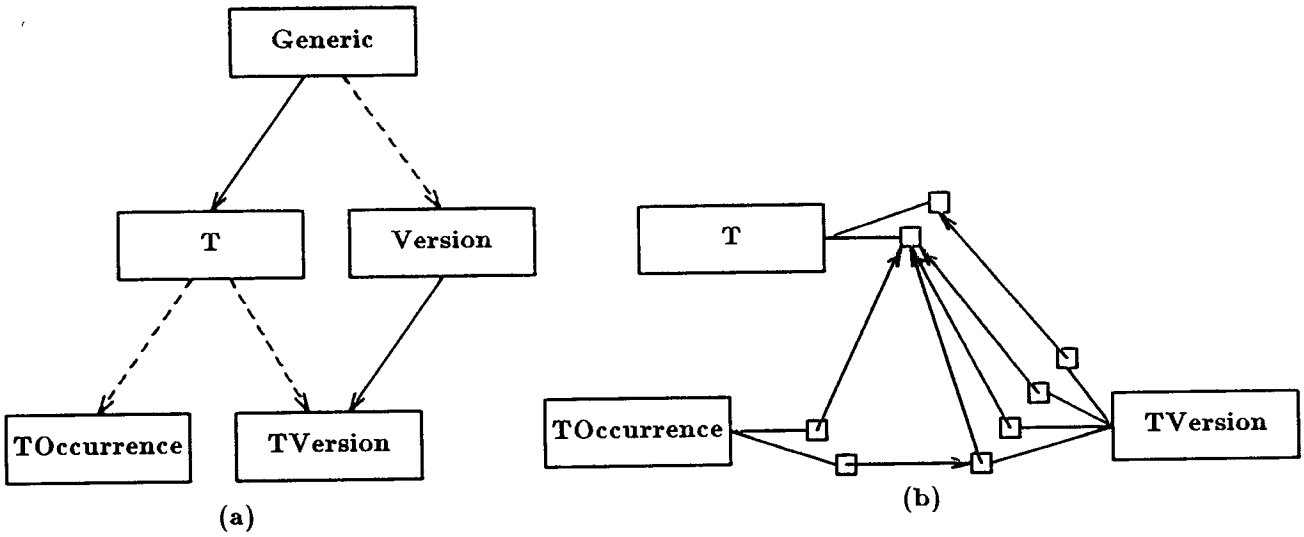

Figure 8. (a) schema definition for T, TVersion and TOccurrence. (b) an example of instances of these types; all arrows indicate the proto-links of the objects to their prototypes.

permit two or more designs to share the same components, and it can not model composite objects that are being designed as assemblies of other already existing components. Furthermore, when the complex design is deleted from the database, its components have to be deleted, and when a version is copied to derive a new version, new versions for all components should be created. Our view is that each component is an independent design object by itself and whether it should be deleted or new versions of it should be created has nothing to do with where this component is used.

Acknowledgments

I would like to thank Effie Biliris and Himanshu Shina for making fruitful comments on issues discussed in this paper.

References

[**Alba85**] Albano, A., Cardelli L., and Orsini R., "Galileo: A Strongly-Typed, Interactive Conceptual Language," *ACM, Tran. on Database Systems*, Vol. 10, No. 2, June 1985, pp. 230-260.

[**Atwo85**] Atwood, M. T., "An Object-Oriented DBMS for Design Support Applications," *IEEE, Proc. Computer Aided Technologies*, 1985, pp. 299-307.

[**Banc85**] Banchilhon, F., Kim, W. and Korth, H. F., "A Model of CAD Transactions," *Proc. Int. Conf. on Very Large Data Bases*, Stockholm, Sweden, 1985, pp. 25-31.

[**Bane87**] Banerjee, J., Chou H., Garza J. F., Kim W., Woelk D., Ballou N., and Kim H., "Data Model Issues for Object-Oriented Applications," *ACM Trans. on Office Inf. Systems*, Vol. 5, No. 1, January 1987, pp. 3-26.

[**Bato85**] Batory, D. S., and Kim, W., "Modeling Concepts for VLSI CAD Objects," *ACM Trans. on Database Systems*, Vol. 10, No. 3, Sept. 1985, pp. 322-346.

[**Beec88**] Beech, D., and Mahbod B., "Generalized Version Control in an Object Oriented Database," *Proc. IEEE Data Engineering Conference*, February 1988, pp. 14-22.

[**Bili89a**] Biliris, A., and H. Zhao, "Design Versions in a Distributed CAD Environment," *IEEE, Int. Conf. on Computers and Communications*, Phoenix, Arizona, March 1989, pp. 354-359.

[**Bili89b**] Biliris, A., "Management of Objects in Engineering Design Applications," BU, Comp. Sc., TR 89-005, April 1989.

[**Bili89c**] Biliris, A., "Database Support for Evolving Design Objects," *ACM, IEEE 26th Design Automation Conference*, Las Vegas, Nevada, June 1989, pp. 258-263.

[**Care88**] Carey, M.J., DeWitt, D.J. and S.L. Vandenberg, "A Data Model and Query Language for EXODUS," *ACM SIGMOD Int. Conf. on Management of Data*, Chicago, June 1988, pp. 413-423.

[**Caru87**] Caruso, M., and E. Sciore, "The VISION Object-Oriented Database System", *Proc. International Workshop on Database programming Languages*, Roscoff France, 1987

[**Chou88**] Chou, H., and W. Kim, "Versions and Change Notification in an Object-Oriented Database System," *ACM, IEEE, 25th Design Automation Conference*, 1988, pp. 275-281.

[**Cope84**] Copeland, G. P., and Maier, D., "Making Smalltalk a Database System," *Proc. ACM SIGMOD Int. Conference on Management of Data*, June 1984.

[**Depp86**] Deppisch, U., Paul H.-B., and Schek H.-J., "A Storage System for Complex Objects," *ACM, IEEE Proc. Int. Workshop on Object-Oriented Database Systems*, September 1986, pp 183-195.

[**Ditt88**] Dittrich, K. R., and Lorie, R., "Version Support for Engineering Database Systems," *IEEE Trans. on Software Engineering*, Vol. 14, No. 4, April 1988, pp. 429-437.

[**Eckl87**] Ecklund, D. J., Ecklund E. F. Jr., Eifrig R. O., and Tonge F. M., "DVSS: A Distributed Version Storage Server for CAD Applications," *Proc. Int. Conf. on Very Large Databases*, England, 1987, pp. 443-454.

[**Hask82**] Haskin, R. L., and Lorie, R. A., "On Extending the Relational Database System," *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, 1982, pp.207-212.

[**Horn87**] Hornick, M.F., and Zdonik, S.B., "A Shared, Segmented Memory System for an Object-Oriented Database," *ACM, Trans. on Office Inf. Systems*, Vol. 5, No. 1, January 1987, pp. 70-95.

[**Hull87**] Hull, R., and R. King, "Semantic Database Modeling, Survey, Applications, and Research Issues," *ACM Computing Surveys*, Vol. 19, No 3, September 1987, pp. 201-260.

[**Katz87**] Katz, R. H., and Chang E., "Managing Changes in a Computer-Aided Database," *Proc. Int. Conf. on Very Large Databases*, Brighton, England, 1987, pp. 455-462.

[**Khos86**] Koshafian, S. N., and Copeland, G. P., "Object Identity," *Proc. ACM, Object Oriented Programming Systems Languages and Applications*, September 1986, pp. 406-416.

[**Kim84**] Kim, W., Lorie, R., McNabb, D., and Plouffe, W., "Transaction Mechanism for Engineering Databases," Proc. *Int. Conf. on Very Large Data Bases*, Singapore, August 1984, pp. 355-362.

[**Kim87**] Kim, W., Banerjee J., Chou H., Garza J., and Woelk D., "Composite Object Support in an Object-Oriented Database System," *Proc. ACM, OOPSLA*, October 1987, pp. 118-125.

[**Kim89**] *Object-Oriented Concepts, Applications and Databases*, W. Kim and F. Lochovsky, Eds., Addison-Wesley, New York, NY, 1989.

[**Klah85**] Klahold, P., G. Schlageter, R. Unland, W. Wilkes, "A Transaction Model Supporting Complex Applications in Integrated Information Systems," *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Austin, Texas, 1985, pp. 388-401.

[**Land86**] Landis, S. G., "Design Evolution and History in an Object-Oriented CAD/CAM Database," *IEEE, Proc. COMPCON*, 1986, pp. 297-303.

[**Lieb86**] Lieberman, H., "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems," *Proc. ACM, Object Oriented Programming Systems Languages and Applications*, September 1986, pp. 214-223.

[**Lori83**] Lorie, R., and Plouffe, W., "Complex Objects and Their Use in Design Transactions," *IEEE, Database Week - Engineering Design Applications*, 1983, pp.115-121.

[**Lyng86**] Lyngbaek, P., and W. Kent, "A Data Modeling Methodology for the Design and Implementation of Information Systems," *ACM, IEEE Proc. Int. Workshop on Object-Oriented Database Systems*, September 1986, pp 6-17.

[**Maie86**] Maier, D, Stein J., Otis A., and Purdy A., "Development of an Object Oriented DBMS," *ACM, Proc. Object Oriented Programming Systems Languages and Applications*, September 1986, pp. 472-482.

[**Maie89**] Maier, D., "Making Database Systems Fast Enough for CAD Applications," Oregon Graduate Center, TR CS/E-87-016. Also, in *Object-Oriented Concepts, Applications and Databases*, W. Kim and F. Lochovsky, Eds., Addison-Wesley, New York, 1989.

[**Mylo80**] Mylopoulos, J., P.A. Bernstein and H.K.T. Wong, "A Language Facility for Designing Database-Intensive Applications," *ACM Trans. on Database Systems*, Vol. 5, No. 2, 1980.

[**Nara88**] Narayanaswamy, K., and Rao, K. V., "An Incremental Mechanism for Schema Evolution in Engineering Domains," *Proc. IEEE 4th Data Engineering Conference*, February 1988, pp. 294-301.

[**Pu88**] Pu, C., G. Kaiser, and N. Hutchinson, "Split-Transactions for Open-Ended Activities," *Int. Conf. on Very Large Data Bases*, August 1988, pp. 26-37.

[**Rehm88**] Rehm, S., *et al.*, "Support for Design Processes in a Structurally Object-Oriented Database System," *2nd Int. Workshop on Object-Oriented Database Systems*, Springer-Verlag, LNCS 334, 1988, pp. 80-96.

[**Rowe87**] Rowe, L, and Stonbraker M., "The POSTGRES Data Model," *Proc. Int. Conference on Very Large Data Bases*, England, 1987, pp. 83-96

[**Unga87**] Ungar, D., and Smith B., "Self: The Power of Simplicity," *Proc. ACM, Object Oriented Programming Systems Languages and Applications*, October 1987, pp. 227-242.

[**Wilk89**] Wilkes, W., P. Klahold, and G. Schlageter, "Complex and Composite Objects in CAD/CAM Databases," Proc. *IEEE, Int. Conf. on Data Engineering*, February 1989, pp. 443-450.