# Modeling Design Object Relationships in PEGASUS

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

## *Abstract*

One of the most important problems of database management systems for CAD applications is modeling design objects and the complex relationships among them. Design objects have versions and participate in the construction of other more complex design objects. In this paper we describe the preliminary data model design of a database system for CAD applications, named Pegasus. Our model is based on previously published work on extensible and object-oriented database systems. Novel ideas of our model that are presented in this paper, include relationship objects as a modeling construct to represent relationships of any degree, replication, and the reconciliation of two subtyping 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 parts and assemblies (with their versions, components, etc.) can be modeled in the world of mechanical assembly design.

## 1. Introduction

Extensible and object-oriented database systems are emerging as a very promising solution to handle complex CAD applications because they greatly surpass pure relational systems in modeling power and performance. A comprehensive list of why object-oriented systems can deliver the performance that conventional systems do not is given in [Maie89], and a collection of papers on object-orientation can be found in a recently published book, [Kim89].

A CAD product is an aggregation of design objects and associated documents. A design object may have many *representations* each having many *versions*, [Katz87, Ditt88]. 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*; 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*, and the constituent objects are called *components*. A component object (generic or version) is by itself another design object and as such it may have its own components. A *component occurrence* describes how the component object participates in a composite object. (It is called *component instance* in [Bato85].) References to generic objects are called *generic references*; they 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.

Specific approaches to support complex objects and/or versions have been proposed [Katz87, Kim87, Ditt87, Beec88, Nara88, Ditt88, 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, Ditt87, Beec88, Nara88], and for composite objects described in [Kim87, Ditt87, Wilk89] are integrated into their data model. Considerable work in this area has also been reported in [Zdon86, Kemp87, Hard88].

This paper presents our own research in this area, and it is a summary of [Bili89a]. We present the basic elements of the PEGASUS data model in **Section 2.** Subtyping is discussed in **Section 3.** In **Section 4** we discuss relationships and relationship objects. Replication of attribute values of objects participating in a relationship is introduced in **Section 5.** The modeling elements introduced in the above sections are explained with the help of examples that deal with employees and departments. **Section 6** provides an example of modeling mechanical CAD objects. Finally, **Section 7** presents our conclusions.

## 2. Basics of the Data Model

PEGASUS starts with the modeling elements provided in the EXTRA data model, [Care88]. We will try to keep the discussion brief on modeling elements that are similar to EXTRA, and elaborate on those that present some difference.

The database is a collection of 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 that are common to all of its potential instances, the operations that can be performed on these instances, and the constraints that attribute values of these instances must satisfy. The attributes, operations and constraints of an object are collectively called the *properties* of the 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 1 defines the types Person, Employee and Department and creates two sets of Employee objects, Emps1 and Emps2, and two sets of Department objects, Depts1 and Depts2.

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 function Employee.manager is implemented as 'this.worksFor.manager', where *this* is a special variable implicitly bound to the object to which the function is applied. Additional operations for types that have already registered in the database can be defined as in [Care88].

Each attribute specification defines the type of the attribute value. Primitive types (integers, characters, etc.), user defined abstract data types (such as Date in Person type), type constructors (tuples, arrays, and sets) and the three kinds of attribute value

CH2840-7/0000/0228$01.00 © 1990 IEEE

```
define type Person: (
      ssn:        int4;
      name:       (fname: char[20]; lname: char[20]);
      dob:        Date
      address:    char[30];
);
operations:
      age         returns int4;
end Person;

define type Employee refines Person: (
      salary:     int4;
      worksFor:   ref Department inverseof emps;
);
operations:
      manager     returns Employee;
end Employee;

define type Department: (
      name:       char[20];
      manager:    ref Employee;
      emps:       {ref Employee} inverseof worksFor;
);
end Department;

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

Figure 1. A simple database schema. The character pairs ( ), { }, and [ ] denote tuple-, set-, and array-objects, respectively.

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; the value of the attribute is either null or the oid of the referenced object. Finally, an **own ref** attribute is a reference to an object with the semantics that the referenced object is both: (1) *dependent* on the referencing object, i.e., it is deleted when the referencing object is deleted; (2) *exclusively* referenced by the referencing object, i.e., the referenced object can not be referenced through an **own ref** attribute by more than one object.

We further refine **own ref** attributes into two kinds of reference semantics, **exclusive ref** and **dependent ref**, each satisfying *only* one of the above constraints. **own ref** is treated here as a synonym for **exclusive dependent ref** attribute. An **exclusive ref** implies that the referenced object can not be referenced by another object through an **exclusive ref** or **dependent ref** attribute. However, the referenced object is not deleted when the referencing object is deleted. For example, we could extend our Department type with an additional field to keep track of the cars given to each department by adding the attribute 'car: **exclusive ref** Vehicle'. This implies that two departments may not reference the same car. However, when a department object is deleted, the referenced car does not disappear automatically from the database; it exists and it may later on be referenced by another department. A **dependent ref** implies a shared ownership of the referenced object, i.e., the referenced object is deleted when all objects that reference it through a **dependent ref** attribute have been deleted. For example, we could extend our Employee type with an additional field to keep track of employee's kids by adding the attribute 'kids: {**dependent ref** Person}'. This implies that one or more employees may reference the same Person object through their kids attribute. When an employee is deleted, his/her kids are deleted only if no other employee references them.

Other modeling elements shown in the schema of Figure 1, such as **refines** and **inverseof** are discussed in subsequent sections.

## 3. Inheritance

Types in PEGASUS, as in all object-oriented systems that we are aware of, are related with the supertype/subtype relationship in a type hierarchy [Card85]. If $X$ is above $Y$ in this hierarchy we say that $X$ is a supertype of $Y$, and $Y$ a subtype of $X$. PEGASUS provides two subtyping mechanisms, namely *refinement* and *extension*. Before we explain the difference we first give the interpretation of subtyping:

> If $Y$ is a subtype (refinement or extension) of $X$, it implies that an instance of $Y$ can be used every where an instance of $X$ is expected.

Thus, $Y$ supports all properties defined in $X$ and it may have additional properties; operations defined in $X$ may be re-implemented in $Y$, and attributes may be redefined so that their permitted values are more restricted. In the following we discuss the two kinds of subtyping.

### Refinement (Schema Inheritance)

When a type $Y$ refines a type $X$, an instance of $Y$ maintains its own storage for the values of all attributes defined in $Y$ and $X$. 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 2(a). Instances of $X$ and $Y$ are independent of each other, e.g., updating an instance of $X$ does not affect any instance of $Y$, and vice versa. An example of refinement is shown in the schema of Figure 1 where the Employee type is defined as a refinement of Person. Each Employee object holds the values of the three attributes defined in Person and the two attributes defined in Employee. Inheritance through refinement is exactly the kind of inheritance provided in some extensible and object-oriented systems such as Postgres [Rowe87a], ORION [Bane87], GemStone [Maie86a], and EXTRA [Care88].

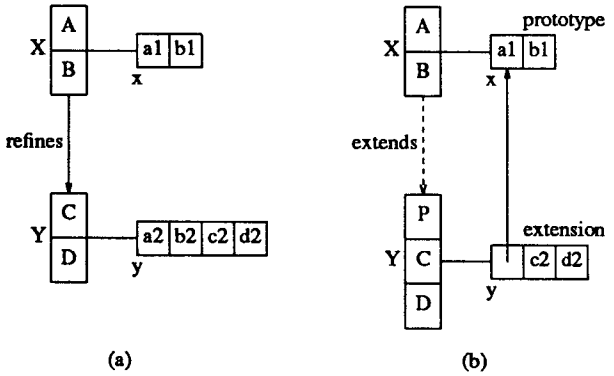



Figure 2. (a) Refinement: instances of $Y$ maintain their own storage for the attributes defined in $Y$ and $X$; instances of $X$ and $Y$ are independent of each other. (b) Extension: all attribute values of $x$ are shared by $y$. (Undirected lines indicate the type of an object.)

### Extension (Value Inheritance)

When a type $Y$ extends $X$, the following is implied: for every instance $y$ of $Y$ there exists an instance $x$ of $X$ (or any subtype of $X$) whose attribute values are inherited (*shared*) by $y$. The object $x$ whose values are inherited by $y$ is called the *prototype* of $y$, and $y$ is called an *extension* of its prototype, see Figure 2(b). The prototype of an object is referenced through a new kind of **ref** attribute, called **proto ref** attribute[1]. The above existential

[1] In [Bili89c], it was called **extends ref**.

constraint implies that an object referenced by one or more objects through **proto ref** attributes can not be deleted, and the value of a **proto ref** attribute of an extension object can not be null. Otherwise, a **proto ref** attribute is treated as any other **ref** attribute; it's value can be explicitly queried or updated by a user. The latter implies that an extension object may have different prototypes over time.

Value inheritance works as follows. Assume $P$ to be a **proto ref** attribute of an object $y$ of type $Y$. A request to access (read or write) an attribute $y.A$ that has not been defined in $Y$ is forwarded to $y.P$ and recursively to $y.P$'s prototype until we find an object $x$ for which $A$ has been defined, and the access is performed on $x$. Likewise, when an operation $O$ is applied on $y$ and $O$ has not been defined in $Y$, the prototypes of $y$ are recursively visited until such operation is found, then that $O$ is applied on $y$. The terms *prototype* and *extension* appear in object-oriented languages presented in [Lieb86, Unga87]. The process of forwarding a message that an extension object can not respond to to its prototype is called *delegation*. In these languages, however, no types are associated with objects. Any object can serve as a prototype; an extension is created by defining a list of its prototypes as well additional properties idiosyncratic to the object itself.

Figure 3 shows types Employee and Student defined as extensions of type Person (Person is defined in Figure 1). Each Employee object $e$ will store the values of the three attributes defined in Employee only, and it will share the attribute values of the object that is referenced in $e$.person. Similarly, a Student object $s$ will hold the values of person and gpa attributes, and it will share the attribute values of $s$.person. For example, assume that $s$.person references an instance of Person. A request to get $s$.ssn is resolved by the system by retrieving $e$.person.ssn, $s$.dob is resolved to $s$.person.dob, etc. Similarly, the 'age' function, defined for Person objects to return 'today - this.dob', can be applied on $s$, i.e., $s$.age, and the system will return 'today - this.person.dob'.

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

define type Student: (
        person:     proto ref Person;
        gpa:        float4;
); end Student;
```

Figure 3. Employee and Student as extensions of Person.

In the above example, $s$.person may actually reference an object of any type that is a subtype (refinement or extension) of Person. In particular, $s$.person may reference an Employee object. We may see how easy it is to model a student $s$ who also works for a particular department, without even introducing a new type for these kinds of objects; $s$ will inherit the attribute values of $s$.person (an Employee object) and recursively the ones of $s$.person.person (a Person object).

### Selective Attribute Value Sharing

Our interpretation of attribute value sharing treats read and write requests the same way. An attempt to update $y.A$ of the object $y$ shown in Figure 2(b) results on an update of $y.P.A$, i.e., $x.A$ is updated. If there are other extensions of $x$ in the database, the new $x.A$ value is automatically shared by them. In many cases this situation is exactly what we want from the semantics of attribute value sharing (as the name implies).

It is also possible, however, that we may want extensions to have their own values for some of the attributes defined in a prototype. Consider the schema of Figure 3. A Student may have two addresses, one in which she regularly leaves, and another one while she is in College. Assume that when we do not know her on campus address (e.g., during the summer) we want to use her permanent address. One way to model such situations in PEGASUS is through the **local** (*attribute-list*) clause attached to a **proto ref** attribute definition. It specifies two things: (a) updates on attributes in *attribute-list* are performed locally on the extension, and (b) when no value is set for one of these attributes in the extensions, the corresponding value of the prototype is taken as default. Extension objects that localize some attributes of their prototypes keep a flag, called *local-flag*, for each such attribute to indicate whether the attribute value has been updated on the extension object. The system defined operation 'default(*attribute-list*)' can be applied on an extension object to unset the local-flag associated with the attributes in *attribute-list* after an update on that attribute has been performed. For the above example, the Student type would have been defined as follows:

```
define type Student: (
        person:     proto ref Person local (address);
        gpa:        float4;
); end Student;
```

A Student $s$ will now hold the values of attributes person, dob, and address, and it will keep a local-flag for the address attribute. The rules of accessing $s$.address are as follows.

Write: always performed locally on $s$.address (the prototype $s$.person remains unaffected), and the local-flag of $s$.address is set.

Read: If the local-flag of $s$.address is set, the value of $s$.address is returned; otherwise, the request is forwarded to $s$.person.

Note the difference between the above approach with the one in which 'address' is defined in Student type as any other attribute. In the second approach, $s$.address is independent from $s$.person.address. Assume that when a student goes home during the summer period we set the value of $s$.address to whatever is the current value of $s$.person.address; this could not work if after this update $s$.person.address is updated, i.e., the student moves to another permanent address. Note also the implications of defining a 'campus-address' attribute in Student type of Figure 3. Now we can no longer apply on $s$ those operations of Person that access the address value of an object, if the value that we want to be accessed is $s$.campus-address. These operations have to be re-implemented for Student objects and the benefits of inheritance are partially lost.

Although, the above problems are perhaps minor when we try to model students and their addresses, they are very important in CAD applications as we will see in the example in the last section.

### Multiple Inheritance

The type hierarchy is a DAG; it is possible for a type to inherit properties from several supertypes (*multiple inheritance*). The problem with multiple inheritance is the naming conflict arising when identically named properties have been defined in the supertypes. Such conflicts that may occur when a type T is defined are resolved as follows. Let us call $SR$ and $SE$ the sets of supertypes of which T is a refinement and an extension, respectively. We first check for conflicts among types in $SR$ only, with the algorithm described in [Care88]. If the conflicting names in the supertypes originate (i.e., they have been inherited) from a

common ancestor in the type hierarchy the conflict is resolved automatically. Otherwise, conflicting names should be renamed in $T$ by the user; i.e., no attempt is made by the system to resolve them automatically. Then we check for conflicts among types in $SE$ only, exactly the same way. After this step is successfully finished and there are conflicting names between types in $SR$ and $SE$ we give priority to the names defined in $SR$ over those defined in $SE$. (Of course, the definer of $T$ may alter this behavior by renaming.)

Figure 4 shows a type $W$ that refines $X$ and $Y$, and it extends $Z$ through the **proto ref** attribute $P$. The net result of this definition is that an instance $w$ of $W$ will hold the values of attributes $A, B, C, D$ (whose definition is inherited by $X$ and $Y$), the values of attributes $F, G, P$ (defined in $W$), and it will share the value of attribute $E$ of the object referenced in $w.P$.

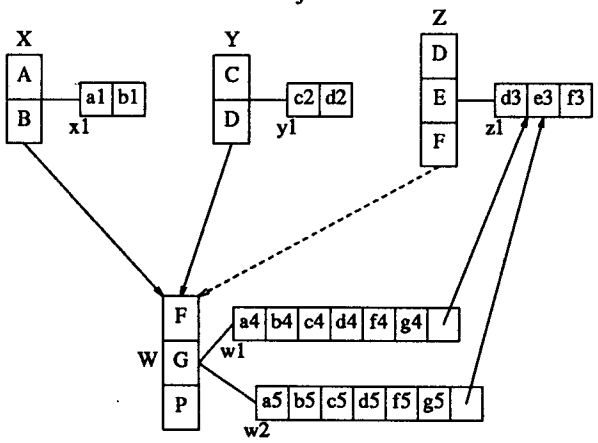



Figure 4. Type W has been defined as: **define type** W **refines** (X, Y): (F: ...; G: ...; P: **proto ref** Z;); **end** W;. That is, W refines X and Y, and it extends Z.

## 4. Relationships

The object-oriented paradigm of modeling has been criticized for lack of support for relationships and referential integrity constraints [Rumb87]; we believe that these are very important modeling elements that should be explicitly supported by an O-O DBMS. In this section we propose mechanisms that would allow the user to define binary relationships as well as relationships of degree three or more.

Assume types $X$ and $Y$ (not necessarily distinct), and attributes $X.a$ and $Y.b$ (not necessarily distinct). When $X.a$ and $Y.b$ are defined to be references to instances of $Y$ and $X$, respectively, and inverse of each other, a binary relationship is established between instances of these two types for which PEGASUS actively enforces the following *cross-referential integrity constraint*:

> for every $X$ object $x$, the value of $x.a$ is either null or a reference to an object $y$ whose $y.b$ value is $x$, **and**
> for every $Y$ object $y$, the value of $y.b$ is either null or a reference to an object $x$ whose $x.a$ value is $y$.

The attributes through which two objects participate in a relationship are indicated in the type definition by matching **inverseof** clauses. The mapping cardinality of a relationship, which expresses how many instances of one type can be associated with how many instances of another type, is simply determined by the number of references that each **inverseof** attribute can accept as values.

Here are some examples. The worksFor and emps attribute definitions of Figure 1 establish a many-one relationship from Employee to Department objects and the following constraint is enforced: for every Employee $e$, Department $d$, the value of $e$.worksFor is $d$, if and only if $e$ is a member of $d$.emps. PEGASUS ensures that the emps attribute of a department is updated properly whenever the worksFor attribute of an employee is updated, or an employee is fired or hired. If employees were working in many departments, worksFor would have to be defined as 'worksFor: {**ref** Department} **inverseof** emps' which establishes a many-many relationship between Employee and Department objects. Similarly, a one-one relationship can be established if both of the **inverseof** attributes have been defined to have single reference values. For instance, assume that at most two employees may be assigned the same office. The attribute definition 'officemate: **ref** Employee **inverseof** officemate' in Employee type defines a one-one relationship between Employee instances, and the following constraint is enforced: for every Employee objects $e1$ and $e2$, the value of $e1$.officemate is $e2$ if and only if the value of $e2$.officemate is $e1$.

Binary relationships can be defined for individual objects only. Consider the schema of Figure 1. Suppose we want to enforce cross-referential constraints for members of Emps2 and Depts2 only, and not for any instance of Employee and Department. Then, we should not had included the **inverseof** clauses with the types. Instead, we can define this relationship for members of the above sets only, as follows:

**inverseof** Emps2.worksFor, Depts2.emps

This means that for every $e$ in Emps2, $d$ in Depts2, the value of $e$.worksFor is $d$ if and only if $e$ is in $d$.emps. An employee in Emps2 can no longer work for a department in Depts1, and a department in Depts2 can no longer have employees that are in Emps1.

### Relationship Types

The relationships we discussed above do not introduce new database objects and do not, in any way, change the structure of the participating objects; they are binary relationships only and have no attributes associated with the relationship itself. However, relationships of degree higher than two may have to be defined. A ternary (or n-ary, in general) relationship can be broken down into three (or more, respectively) binary relationships. It is well known, however, that in general the set of the resulting binary relationships is not always equivalent to the original n-ary relationship (see a database textbook, e.g., [Elma89], page 58.) In addition, and regardless of the degree of the relationship, one may want to associate some attributes with the relationship to describe how two or more objects are related to each other. Such situations can be modeled in PEGASUS through *relationship types*; instances of relationship types are called *relationship instances* or *relationship objects*.

A relationship type $R$ of degree $n$, $n \geq 2$, will have $n$ **inverseof** attributes, each defined as a *single* reference to an instance of the participant types, and zero or more attributes that provide additional information about $R$. Relationship objects represent relationships among existing database objects; thus, their **inverseof** attributes can not have null references.

As an example, assume that employees are assigned by different departments to work on a number of different projects and different departments may assign employees to work on the same project. We are interested in knowing the number of hours that each employee works on a given project and the department that assigned this job. Figure 5a shows how this ternary relationship can be established by the definition of the Assignment relationship type (the **mode** clause appearing in Employee is explained shortly). Each 4-tuple Assignment object $(e, d, p, h)$ implies that an employee $e$ has been assigned by department $d$ to work on project $p$ for $h$ hours per week. The matching attribute

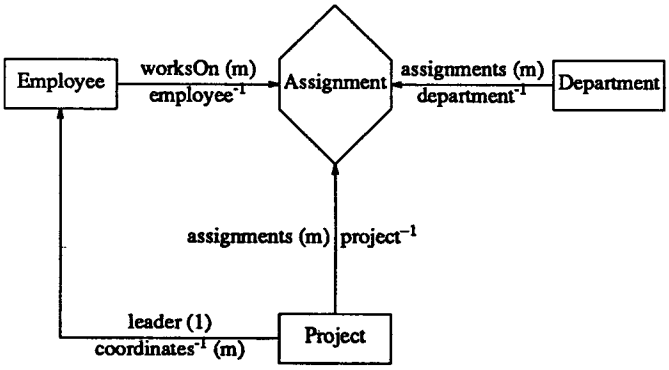
```
define relationship Assignment: (
        employee:   ref Employee inverseof worksOn;
        department: ref Department inverseof assignments;
        project:    ref Project inverseof assignments;
        hours:      int1;
); end Assignment;


define type Employee: (
        worksOn:    {ref Assignment} inverseof employee mode (r);
        coordinates: {ref Project} inverseof leader;
        ssn: int4; name: char[20]; salary: int4;
); end Employee;


define type Department: (
        assignments:    {ref Assignment} inverseof department;
        name: char[20]; manager: ref Employee;
); end Department;


define type Project: (
        name:           char[30];
        leader:         ref Employee inverseof coordinates;
        startDate:      Date;
        assignments:    {ref Assignment} inverseof project mode (r);
); end Project;
```

(a)




(b)

**Figure 5.** (a) A database schema. (b) Object-Relationship diagram for the schema defined in (a); only inverseof attributes are shown. Rectangles represent ordinary types and diamonds represent relationship types.

pairs (Employee.worksOn, Assignment.employee), (Department.assignments, Assignment.department), and (Project.assignments, Assignment.project) collectively specify the following:

For every Employee $e$, Department $d$, Project $p$, Assignment $a$, $a$ is a member of $e$.worksOn and $d$.assignments and $p$.assignments if and only if the value of $a$.employee is $e$ and the value of $a$.department is $d$ and the value of $a$.project is $p$.

In addition, instances of types Employee and Project participate in a binary relationship through their coordinates and leader attributes, respectively. The diagram of Figure 5b, similar to Entity-Relationship diagram, captures all the relationships between the above objects. The numbers inside the parenthesis attached to some attributes designate the mapping cardinalities of the relationships. **inverseof** attributes of a relationship are always single references and thus no number is attached to them.

The **mode** clause specifies whether application programs are permitted to read, write or initialize an **inverseof** attribute value (this is identical to the *allows* clause of [Onto87]). If no **mode** is specified all operations are permitted. The system can always update an **inverseof** attribute to compensate other actions performed by users. The schema of Figure 5 specifies that no user can update the worksFor attribute of an Employee object $e$, and the assignments attributes of a Project object $p$. These attributes will be updated by the system only when an assignment object that relates $e$ or $p$ is deleted, created or modified.

## Roles

Objects may participate in a relationship in different *roles*. We illustrate roles with the help of an example that models the mother-father-child relationship among persons, shown in Figure 6. The different roles that an object may play in a relationship are indicated by the attribute names through which the object participates in the relationship. Attribute asParent of Person has been defined as **inverseof** (mother, father) which is interpreted as being the inverse of mother or father. That is, for every Person object $p$, MFC object $r$, $r$ is in $p$.asParent if and only if the value of $r$.mother is $p$ or the value of $r$.father is $p$. Thus, a Person object may participate in a MFC relationship either as a child or as a parent, and furthermore a Person object that plays the role of a parent may play the role either of a mother or a father. MFC is a many-one relationship from (Person (as mother), Person (as father)) to Person (as child).

```
define type Person: (
        asParent:   {ref MFC} inverseof (mother, father);
        asChild:    ref MFC inverseof child;
); end Person;


define relationship MFC: (
        mother:     ref Person inverseof asParent;
        father:     ref Person inverseof asParent;
        child:      ref Person inverseof asChild;
); end MFC;
```

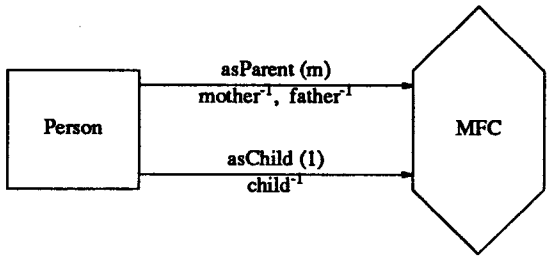



**Figure 6.** The ternary MFC (Mother-Father-Child) relationship among Person objects.

The reader may wonder why we did not define in type Person three attributes (asMother, asFather, and asChild) each indicating the very distinct roles that a person may play in MFC relationship. Certainly, for this particular example, we could. We have chosen the approach shown in Figure 6 just to show how object relationships can be modeled when the roles that objects may play in the relationship can not be distinguished in a meaningful way. This is the case of all relationships that are *symmetric*. Consider the case of a relationship $R$ (part1, part2) which represents connections between two parts. If p1 is connected to p2, then p2 is connected to p1. What would be the meaning of saying that p1 plays the role of part1 and p2 plays the role of part2 (or vice versa)? We are simply interested in whether p1 and p2 participate in the relationship; their role is unimportant. Thus, a type definition for parts would include an attribute of type '{ref R} **inverseof** (part1, part2)' simply to indicate the participation of a part in R and not its particular role.

### Semantics of Update Operations

So far, we have assumed that **inverseof** attributes are not arrays. The reason is that arrays imply an ordering among its elements that only the application knows. For example, when a reverse reference which happens to be the $i$-th element of an array $a$ has to be deleted, the system could not know whether to set $a[i]$ to null or shrink the array. Similarly, the system could not know where in the array a reverse reference should be inserted. For these reasons **inverseof array** attributes are permitted only if the matching **inverseof** attribute is not an array and its **mode** is read-only. In that case, the user manipulates the array and the system updates the matching attribute.

Deleting (or creating) an object $x$ will require a participant object $y$ to update the value of its attribute $a$ through which it supposed to reference $x$. The functions used by PEGASUS to unset and set a reference of $y.a$ to $x$ work as follows:

*unsetReverseRef* $(y, a, x)$
if $y.a$ is a set of references then delete $x$ from this set, else (i.e., $y.a$ is defined as a single reference) set the value of $y.a$ to null.

*setReverseRef* $(y, a, x)$
if $y.a$ is a set of references then insert $x$ into this set, else (i.e., $y.a$ is defined to be a single reference) set the value of $y.a$ to $x$

Below are the steps followed by PEGASUS when objects participating in relationships are delete from the database.

D1. Delete a relationship object $r$.
For each object $p$ referenced by one of $r$'s **inverseof** attributes, *unsetReverseRef* $(p, a, r)$ where $a$ is the matching **inverseof** attribute of $p$. Then, delete $r$ itself from the database.

D2. Delete a non-relationship object $p$.
For each object $y$ referenced by an **inverseof** attribute of $p$, if $y$ is a relationship object then delete $y$ as in D1, else *unsetReverseRef* $(y, a, p)$, where $a$ is the matching **inverseof** attribute of $y$. Then, delete $p$ itself from the database

As an example, consider the schema of Figure 5; when a a project $p$ is deleted, the following actions are taken by PEGASUS:

```
for each Assignment object a referenced in p.assignments {
      delete a from a.department.assignments
      delete a from a.employee.worksOn
      delete a itself from the database }
delete p from p.leader.coordinates.
delete p itself from the database.
```

Symmetric actions are taken when objects participating in relationships are inserted in the database or they are being updated; see [Bili89a].

### Discussion

Perhaps, an interesting question is whether the introduction of relationship objects is necessary, e.g., can we model the Assignment relationship type shown in Figure 5 as any other (not relationship) type that includes exactly the same **inverseof** clauses? There are many reasons why we believe relationships should be treated in a different way. Below we summarize the differences between ordinary and relationship objects. First, while the value of an **inverseof** attribute of an object might be null, the value of an **inverseof** attribute of a relationship object can not. Second, the **inverseof** attributes of a relationship type should be defined as *single* references (not sets or arrays) to objects of the participant types. Although, there is nothing fundamentally wrong in having set **inverseof** attributes in a relationship object, the strategies that should be employed to enforce cross-referential constraints will be significantly more complex than those we have described in previous sections. Third, and perhaps most important, is the issue of update semantics. When a relationship object is deleted, the participant objects are updated (not deleted). On the other hand, when a participant object is deleted all the relationship objects in which it participates are deleted too. This can not be modeled with some kind of **exclusive ref** or **dependent ref** or both (i.e., **own ref**), each for obvious reasons.

Note that user defined constraints can not be used to enforce cross-referential integrity constraints between participant objects. Constraints defined by users are *passively* enforced by the system, i.e., an operation that violates such a constraint is simply rejected. Cross-referential integrity constraints are *actively* enforced by the system, i.e., when such a constraint is going to be violated the system performs certain additional compensating operations in order to satisfy the constraint. Other proposals to handle cross-referential constraints include the use of rules in an active database system [Daya88], and exceptions [Onto87]. Although, there is no question that these mechanisms are valuable in a database system for advanced applications, the use of these general (and complex) mechanisms to implement such fundamental and frequently used modeling construct *is* questionable.

Finally, we note that relationship objects and ordinary objects are distinguished only at the data definition level. The language for queries and updates may treat both kinds of objects exactly the same way.

## 5. Replication

The presence of relationship objects requires that access from one participant object to another be done through the relationship object. This might be awkward in expressing queries and it incurs performance penalties when the participant objects are accessed frequently from each other. Let us take as an example the schema of Figure 5. To find the names of all employees to whom a particular department $d$ has assigned a job, we should retrieve for each $a$ in $d$.assignments, $a$.employee.name. This might involve two disk accesses for each $a$ in $d$.assignments; one to retrieve $a$ itself, and then one more to retrieve $a$.employee. If the employees working for a department are accessed frequently, this performance penalty may be unacceptable.

For the above reasons, PEGASUS provides a mechanism to replicate attribute values of objects of one participant type and place the replica with objects of another participant type. Attributes holding replicated values can not be updated by the user, i.e., they are read-only attributes. The following definition makes each instance $d$ of the Department type, shown in Figure 5, to hold in $d$.emps the value of $a$.employee of each $a$ in $d$.assignments:

```
define type Department: (
      assignments:      {ref Assignment} inverseof department;
      emps:             replicates assignments (employee);
      ...
); end Department;
```

The overhead associated with keeping consistent replicated values in objects participating in a relationship is minimal. In the above example, $d$.assignments is **inverseof** $a$.department; thus, when $a$ is updated we know precisely the object $d$ (it is the value of $a$.department) whose emps attribute has to be updated. Similarly, when a new Assignment object $a$ is inserted in the database, we simply insert $a$.employee in $a$.department.emps. When $a$ is deleted we simply remove $a$.employee from $a$.department.emps.

The above example is a special case of the **replicates** clause. In general, an attribute may be defined to replicate the values of more than one attribute of a participant object (provided all attribute values are of the same type). In the following we

present the general form of a **replicates** clause and subsequently we provide an example.

Assume the following $X$ and $Y$ types:

type $X$
has attributes A = $(a_1, a_2, \cdots, a_k, a_{k+1}, \cdots a_m)$, with
$a_1, a_2, \cdots, a_k$: **ref** $Y$ **inverseof** $b$;

type $Y$
$b$: {**ref** $X$} **inverseof** $(a_1, a_2, \cdots, a_k)$;
$c$: **replicates** $b$ $A$ 1;
$d$: **replicates** $b$ (A1 **inverseof** I1, A2 **inverseof** I2, ...);

Where A1, A2, ..., are subsets of A, and I1, I2, ... are subsets of $(a_1, a_2, \cdots, a_k)$. The attribute definitions $Y.c$ and $Y.d$ imply the following, for every instance $y$ of type $Y$:

$Y.c$: $y.c$ replicates each $x.A$ 1 value of every $x$ in $y.b$.

$Y.d$: $y.d$ replicates all $x$.A1 values if at least one of $x$.I1 values is $y$, and all $x$.A2 values if at least one of $x$.I2 values is $y$, and so on and so forth.

Or equivalently, for every instance $x$ of $X$:

$Y.c$: each $x.A$ 1 value is replicated in $x.a_1.c$, and $x.a_2.c$, $\cdots$, and $x.a_k.c$.

$Y.d$: each $x.A$ 1 attribute value is replicated in $x.a_i.d$ (for all $a_i$ in I1), and each $x.A$ 2 attribute value is replicated in $x.a_j.d$ (for all $a_j$ in I2), and so on and so forth.

As an example, we take the Person type that is shown in Figure 6, and we redefine it so that it includes some replicated values from MFC relationship. The new Person type is shown in Figure 7.

```
define type Person: (
        asParent:    {ref MFC} inverseof (mother, father);
        asChild:     ref MFC inverseof child;
        kids:        replicates asParent (child);
        parents:     replicates asChild (mother, father);
        hasKidsWith:
                     replicates asParent
                     (mother inverseof father, father inverseof mother);
); end Person;
```

Figure 7. Possible replication of attribute values of MFC objects in Person objects.

Assume a Person object $p$. $p$.kids holds references to all children of $p$; $p$.parents replicates $p$.asChild.mother and $p$.asChild.father Finally, $p$.hasKidsWith replicates from each MFC object $m$ in $p$.asParent, the $m$.mother value if $m$.father is $p$, and the $m$.father value if $m$.mother is $p$. Note that if we had defined hasKidsWith as 'replicates asParent (mother, father)' we would have different results because $p$ would have been included in $p$.hasKidsWith. In other words, when a MFC object $m$ is inserted in the database the following actions are taken:

1. insert $m$. child in $m$.mother.kids and $m$.father.kids
2. insert $m$.mother and $m$.father in $m$.child.parents
3. insert $m$.mother in $m$.father.hasKidsWith, and $m$.father in $m$.mother.hasKidsWith.

Replication, as we described it this section, is performed through attributes that define a direct relationship between two object, i.e., the objects are directly crossed-referenced through these attributes. If this constraint is not satisfied we have no way of knowing the objects that have copies of an attribute value of another object, unless reverse indices are maintained by the system. [Maie86b] and very recently [Shek89] describe the design of such reverse indices in the Gemstone and EXODUS system, respectively.

## 6. Mechanical CAD Objects, Example

In this section we present an outline of how generic objects, version objects, and their component occurrences can be represented in modeling mechanical design objects that represent parts (hereafter referred to as *designs*, *design parts* or simply *parts*)[2]. The following assumptions are made about the problem. A part may be simple (non-composite) or it may have two or more component parts which in turn may be simple or composite. A design part may be a component in many higher level designs (e.g., it's a library component). Each design has many versions each being a version of a single design. Parts can be connected to each other at some specific points that we call *handles*. We assume that each handle is a hole and that to connect two parts we use a bolt to tighten them together, a nut, and a lock washer that prevents nuts from becoming loose under vibration.

Generic and version objects are treated as any other ordinary object and as such they are of a given type. The properties of a generic object $g$ represent the common characteristics of all versions of $g$. They can be though of as forming the interface of $g$'s versions. Version properties represent deviations in the defined characteristics of $g$. Although, determination of generic and version properties is application specific we can say that, in general, the latter are those properties that do not change the form, fit, and function of the (generic) design object (i.e., all versions of a design are interchangeable within a higher level assembly).

A component occurrence $c$ of a (generic or version) design object $x$ indicates a particular usage of $x$ in another more complex object; it is distinct from other component occurrences of $x$, and each shares the same $x$ properties. For example, assume that a designer of a bike makes use of two already designed wheels for her design; each of the two wheels that are components of bike corresponds to two different component occurrences of the same wheel design. Each component occurrence describes how the component object participates in the composite object; it may have such attributes as a name (e.g., front wheel), and a transformation matrix that describes where the component is located relative to the composite object.

Figure 8 shows the type Part of generic part objects, and the type PartVersion of their versions. We have chosen the attributes of a generic part object $g$ to be the name and a description of $g$, an array of handles showing how $g$ can be connected to other parts, and a drawing that provides a sketchy picture of the part and its handles. Attribute occurrences holds all occurrences of $g$ in other more complex parts that use $g$. Each Part object also holds a set of references to its versions.

A PartVersion object $v$ shares the attribute values of its generic object referenced in $v$.generic, except for the occurrences attribute which has been defined in PartVersion too. Thus, if $v$ is a version of $g$, $v$.occurrences will give us the composite parts that explicitly use $v$, while the same operation applied on $g$ will give us those objects that use $g$ in a generic way. A version $v$ may have its own sketchy drawing or it may share the one specified in $v$.generic. In addition, each version $v$ includes a version id, various drawings and a set of references to the part occurrences of the parts being used in $v$; for a non composite part version the subparts attribute value will be the empty set. Attributes of type Image have been defined as **dependent ref**; this is because we may want to permit two or more different parts to have the same appearance for certain kinds of drawings, e.g, two parts whose only difference is the material from which they are made.

[2] We would like to emphasize that in this example we model *design* objects that represent some *physical* parts; we do not model *physical* parts.

```
define type Part: (
        versions:       {own ref PartVersion} inverseof generic;
        name:           char[20];
        description:    char[];
        handle:         array[] of Handle;
        drawing:        dependent ref Image;
        occurrences:    {ref PartOccurrence}
                        inverseof occurrenceOf mode (r);
); end Part;

define type PartOccurrence: (
        occurrenceOf:   proto ref Part local (name) inverseof occurrences;
        orientation:    Orientation;
        partOf:         ref PartVersion inverseof subparts mode (r);
        configurations: {ref Configuration} inverseof (p1, p2);
        connectedWith:  replicates configurations
                        (p1 inverseof p2, p2 inverseof p1);
); end PartOccurrence;

define type PartVersion: (
        generic:        proto ref Part local (drawing) inverseof versions;
        occurrences:    {ref PartOccurrence} inverseof occurrenceOf;
        vid:            int4;
        detailDrng, assemblyDrng, explodedDrng: dependent ref Image;
        subparts:       {own ref PartOccurrence} inverseof partOf;
); end PartVersion;

define relationship Configuration: (
        p1:             ref PartOccurrence inverseof configurations;
        p2:             ref PartOccurrence inverseof configurations;
        h1, h2:         int4;
        bolt:           ref Bolt;
        nut:            ref Nut;
        lkWasher:       ref LkWasher;
); end Configuration;
```

Figure 8. Type definitions of generic design parts (Part), their versions (PartVersion), and their component occurrences in more complex parts (PartOccurrence). The Configuration relationship indicates how two part occurrences are connected together in a higher level design.

Figure 8 shows also the type definition of PartOccurrence as an extension of Part. Each component occurrence $c$ shares the properties of the design referenced in $c$.occurrenceOf. The prototype of $c$ may be a generic object (an instance of Part), or a version object (an instance of PartVersion). A component occurrence $c$ may set its own name, or otherwise the name of $c$.occurrenceOf will be used. Attribute $c$.partOf references the design version in which $c$ is being used and $c$.orientation provides the orientation of $c$ relative to the complex object. $c$.configurations references all Configuration relationships (discussed shortly) that describe how $c$ is connected with other component occurrences, and $c$.connectedWith replicates from $c$.configurations these other component occurrences that $c$ is connected with. Note that no operations defined for Part or PartVersion objects need to be redefined in this type. For example, a program that draws a Part or PartVersion object can be applied on $c$. The only difference is that the name of $c$ will appear on the screen instead of the name of $c$.occurrenceOf, and the design will appear the way specified in $c$.orientation. From the user point of view a component occurrence is seen as a copy of the component part.

Instances of type Configuration are binary relationship objects between two part occurrences, p1 and p2, that are connected together in a higher level part. Each such relationship also describes which handles are being used for the connection (h1 and h2 are indices of the handle array of p1 and p2), and the kinds of bolts, nuts and lock washers. Figure 9 summarizes the relationships defined in our schema.

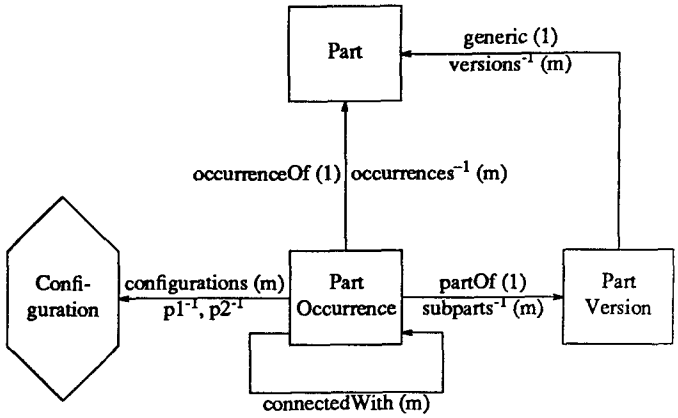



Figure 9. Relationships in which design objects participate.

As an example, Figure 10 shows the subcomponents of object o11 which is a version of o10; o11 uses the object o3, a version of part o1, and it uses twice part o4 in a generic way, i.e., no decision has been made yet as to which versions of o4 will be finally used. Note that when design version o11 is deleted from the database, so are the part occurrences o7, o8 and o9, of other parts used in this design (because o11.subparts is an **own ref** attribute). However, the actual parts of the just deleted part occurrences, i.e., o4 and o3, remain intact.

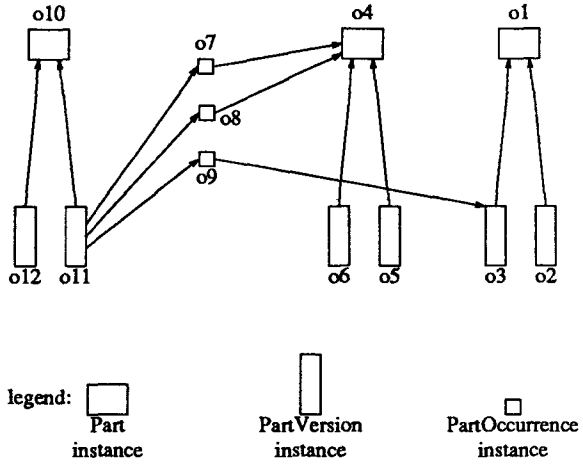



Figure 10. A PartVersion instance o11 (a version of the generic object o10) and its components.

Additional information associated with generic and version objects, such as derivation relationships, the status and creation date of a version, ownership and access control, etc., are discussed in [Bili89b, Bili89c].

## 7. Conclusions

In this paper we have presented a preliminary design of the PEGASUS data model. The target is to support the complex relationships among design objects in a Computer Aided Design environment. After a brief review of the EXTRA data model that PEGASUS is based upon, we presented the two kinds of subtyping (refinement and extension) and we have shown that these mechanisms can co-exists in a database system. Then, we introduced relationship types as a modeling construct to capture the relationships of any degree among objects. Automatic enforcement of cross-referential integrity constraints among objects participating in a relationship, whenever applications specify to be

enforced, is very useful for such complex applications as CAD. Finally, we presented replication of attribute values between objects participating in a relationship, and we have shown that the expected performance penalty because of replication is minimal. In general, a key factor whether relationships or replication should be used is the importance of referential integrity (for relationships) and the relative frequency of retrievals over updates (for replication).

We presented an example that shows how applications may use the modeling elements provided in PEGASUS to handle mechanical CAD objects. In our solution it is not required to have some kinds of built-in structures to represent generic and version design objects as well as their component occurrences in higher level designs. Instead, these kinds of objects are treated as any other object and as such the user has to define their types. The reader should be aware that the definitions of Figure 8 are not necessarily the most appropriate for all kinds of CAD systems. This is exactly our point; it is impossible to define properties for generic and version objects that are appropriate for all kinds of CAD applications. Thus, users working on different applications are allowed to express their own way of seeing their design objects.

**Acknowledgments.** Thanks are due to the other members of the PEGASUS project who contributed to ideas expressed in this paper: S. Braoudakis, R. Dermer, H. Sinha, and H. Zhao. I am also grateful to Jack Orensrein who suggested many improvements to earlier versions of this paper.

## References

[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 Conf. on Data Engineering*, February 1988, pp. 14-22.

[Bili89a] Biliris, A., "Management of Objects in Engineering Design Applications," BU, Comp. Sc., TR 89-005, April 1989.

[Bili89b] Biliris, A., "Database Support for Evolving Design Objects," *ACM, IEEE 26th Design Automation Conference*, Las Vegas, Nevada, June 1989, pp. 258-263.

[Bili89c] Biliris, A., "A Data Model for Engineering Design Objects," *ACM, IEEE 2nd Int. Conf. on Data and Knoweledge Systems for Manufacturing and Engineering*, Gaithersburg, Maryland, October 1989, pp. 49-58.

[Card85] Cardelli, A., and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM, Computing Surveys*, Vol. 1, No. 4, Dec. 1985, pp. 471-522.

[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.

[Daya88] Dayal, U., A. P. Buchmann, and D.R. McCarthy, "Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System," *2nd Int. Workshop on Object-Oriented Database Systems*, Springer-Verlag, LNCS 334, 1988, pp. 129-143.

[Ditt87] Dittrich, K., W. Gotthard, and P. C. Lockemann, "DAMOCLES - the Database System for the UNIBASE Software Engineering Environment," *IEEE, Data Engineering*, Vol. 10, No 1, March 1987, pp. 37-47.

[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.

[Elma89] Elmasri, R., and S.B. Navathe, *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1989.

[Hard88] Hardwick, M., and D.L. Spponer, "Rose: An Object-Oriented Database System for Interactive Computer Graphics Applications," *2nd Int. Workshop on Object-Oriented Database Systems*, Springer-Verlag, LNCS 334, 1988, pp. 340-345.

[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.

[Kemp87] Kemper, A., P.C. Lockemann, and M. Wallrath, "An Object-Oriented Database System for Engineering Applications," *ACM SIGMOD, Int. Conf. on Management of Data*, San Francisco, May 1987, pp. 299-310.

[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.

[Lieb86] Lieberman, H., "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems," *Proc. ACM, OOPSLA*, 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.

[Maie86a] Maier, D, Stein J., Otis A., and Purdy A., "Development of an Object Oriented DBMS," *ACM, Proc. OOPSLA*, September 1986, pp. 472-482.

[Maie86b] Maier, D., and Stein J., "Indexing in Object-Oriented DBMS," *ACM, IEEE Proc. Int. Workshop on Object-Oriented Database Systems*, September 1986, pp 171-182.

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

[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.

[Onto87] Ontologic, *Vbase, Integrated Object System, Technical Notes*, Billerica, MA, 1987.

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

[Rumb87] Rumbaugh, J., "Relations as Semantic Constructs in an Object-Oriented Language," *Proc. ACM, OOPSLA*, October 1987, pp. 466-481.

[Shek89] Shekita, E.J., and M.J. Carey, "Performance Enhancement Through Replication in an Object-Oriented DBMS," *ACM SIGMOD, Int. Conf. on the Management of Data*, Portland, Oregon, June 1989, pp. 325-336.

[Unga87] Ungar, D., and Smith B., "Self: The Power of Simplicity," *Proc. ACM, OOPSLA*, 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.

[Zdon86] Zdonik, S.B., "Version Management in an Object-Oriented Database," Proc. *IFIP, Int. Workshop On Adv. Progr. Environments*, 1986, pp. 397-416.