Relations as Semantic Constructs in an Object-Oriented Language

James Rumbaugh

General Electric
Corporate Research and Development
Schenectady NY

ABSTRACT

The relation as a semantic construct in an object-oriented language clearly expresses associations and constraints among objects which would otherwise be buried in implementation code. The externalization of references between objects permits a symmetric, non-redundant conceptual model which merits its own special notation and predefined operations. The object-relation model, which combines the object-oriented model with the entity-relationship model from data base theory, is particularly useful for designing and partitioning systems of interrelated objects. Relations can be implemented efficiently using hash tables. The model proposed here has been fully implemented in an object-oriented language written by the author which has been used to implement several production applications.

1. INTRODUCTION

The relation is a semantic construct supported by relational data bases [Codd] and semantic data models [Chen, Loomis, Teorey] which is not well supported in object-oriented programming, as exemplified by languages such as Smalltalk [Goldberg]. It is possible to program relations using existing object-oriented constructs, but only by writing a particular implementation in which the programmer is forced to specify details irrelevant to the logic of an application. It is not possible to separate the abstraction from the implementation with the same clarity as found in the relational data models. This paper describes how relations can be added to object-oriented languages so that they complement existing concepts yet greatly enhance expressive power.

1.1 What are relations

A relation associates objects from *n* classes. The state of a relation can be described as a set of elements, each element consisting of one object from each of the *n* classes. A relation expresses associations often represented in a programming language as pointers from one object to another, but in a more symmetric form, as a relation is not unidirectional in the way that a pointer is. Operations can be applied uniformly to a relation as a whole, rather than singling out one of the objects in the relation as the target of a method. The state of a relation can be changed by operations to add or delete elements; the state can be queried by operations to test membership of elements, to select a subset of elements whose values satisfy some condition, and to iterate over the entire set of elements. The ability to apply operations to the entire relation, rather than simply individual objects in it, allows many expressions to be written concisely.

A relation is an abstraction stating that objects from certain classes are associated in some way; the association is given a name so that it can be manipulated. It is a natural concept used in ordinary discourse. For example, the statement "Jim Jones works for Acme Products" implies that "works for" is a relation between persons and companies and that the objects "Jim Jones" and "Acme Products" satisfy this relation.

1.2 Object-oriented languages lack relations

Object-oriented languages express *classification* (the grouping of objects into classes) and *generalization* (the refinement of classes into subclasses) well, but do not contain syntax or semantics to express relations directly. Any program can implement particular relations on an ad hoc basis, but the abstraction may get lost in the implementation mechanisms. Different aspects of a relation can be implemented by methods on the participating object classes, but this distributes the information about the relation among different

classes, rather than gathering the information into a single object which can be manipulated as a unit.

A collection subclass "relation" can be implemented which has methods that implement various relational operations. An object of this class holds sets of n-tuples containing the related objects. Writing programs is easier, because the user of the class need not duplicate the implementation mechanisms hidden inside the class. Such relation objects must be instantiated by the application program at run time.

Providing a class "relation" is basically an implementation tool which does not raise relations to the same semantic level as generalization (the class-subclass hierarchy), which is supported in most object-oriented languages with built-in syntax and semantics. There is no logical necessity for providing a declarative syntax and semantics for generalization; such behavior could be implemented on an ad hoc basis and called for explicitly. Object-oriented languages have built-in constructs for generalization because it is a natural concept that people use in ordinary discourse; it allows algorithms to be written more concisely and more clearly; and it is common enough to justify building it into a language. Relations are also natural, productive, and common in abstracting applications. An object-oriented language is more expressive if relations are a primitive declarative construct, on the same footing as classes.

1.3 Why relations should be a semantic construct

It is important that relations be considered a semantic construct, and not simply an implementation construct. Object-oriented programming has become important because it provides a way of thinking about a problem that is different from previous approaches, such as functional decomposition. An object-oriented data model structures the formulation of a design from its beginning. The use of relations as a semantic construct can have a major impact on the formulation and elucidation of a design, but only if they are considered as semantic constructs of similar weight to classes and generalization.

Relations are particularly useful in the design of larger systems containing many classes that interact, because relations abstract interactions among classes in a natural way. In an existing object-oriented language, such interactions are buried in the instance variables and methods of the classes, so that the overall structure of the system is not readily apparent. Representing a system by an object model containing classes and relations among the classes abstracts the high-level static structure of the system, without having to specify a particular implementation of the classes and their methods. Such a high-level model is useful in partitioning systems into subsystems independently of the implementation of the parts. Our experience has shown that relations are more important to the design of large systems than generalization, because relations affect the partitioning of a system into its parts, while a generalization hierarchy is often confined to a single module within a system.

1.4 Relations can be implemented

Relations can be implemented efficiently. The alleged inefficiency of many relational data bases has several causes: many relational data base management systems only support relations among attribute values, rather than among objects directly; they fail to provide (or users fail to use) appropriate indexing mechanisms; they perform too many operations to the disk, rather than caching data in memory; and they are not implemented as well as they could be, given the theoretical state of the art. Relations can be implemented in an object-oriented language using hash tables for constant-time access, regardless of the size of a relation.

The author has implemented an object-oriented language, the Data Structure Manager (DSM), which adds syntactic and semantic support for relations to existing object-oriented concepts. This language has been used to write production applications at several GE sites. Applications include interactive graphics, simulations, text processing, and the DSM system itself. Our experience has shown that the addition of this concept greatly simplifies conceptualization and implementation of many applications and provides a better fit to real-world problems than the original object-oriented model.

1.5 Organization of this paper

This paper presents the *object-relation* model, which combines the concepts of objects, classes, and methods from the object-oriented model [Goldberg] with the concept of relations from the entity-relationship model [Chen]. The object-relation model is discussed on two levels: logical and implementation. Section 2 describes relations as logical constructs, independent of their implementation. Section 3 discusses the difficulties of using conceptual relations in existing object-oriented languages. Section 4 describes how to extend an object-oriented language to implement relations. Section 5 discusses some possible objections to this model. Section 6 discusses an actual implementation of these concepts in an object-oriented language. Section 7 mentions some open issues. Appendix A briefly describes the author's object-oriented language DSM, which implements the object-relation model. Appendix B summarizes a graphical notation for diagramming object models and shows a small example

of a data model. Appendix C discusses previous work in semantic data modeling which includes many of the basic concepts both of object-oriented data models and the object-relation model.

2. RELATIONS AS LOGICAL CONSTRUCTS

2.1 Definitions

A *relation* exists among an ordered list of object classes. The number of classes participating in the relation is its *degree*. The ordering of the classes in the relation is significant; in general, relations are not symmetric. Each class in the list of classes is called a *field*, identified by its position. Alternatively, each field can be assigned a unique *role name* to identify it within the relation. A class can appear more than once among the fields of a relation, in which case it is particularly important to keep the ordering straight or use role names.

A relation contains a set of *elements*, each a tuple of objects, one for each field of the relation. The class of each object must match the class of its field (it can be a subclass). The form of a relation is fixed but its contents can change over time. Since a relation is a set, each element in it is unique, but a value in a particular field can appear many times in association with different values for other fields. Note that in this paper, relations exist directly among objects, unlike Codd's model for relational data bases, in which relations exist only among attribute values and not among objects themselves. Allowing objects to appear directly in relations greatly simplifies models for complex structures.

2.2 Syntax

The concept of a relation is a natural one, and corresponds to real-world concepts. Relations represent information about the associations among different objects, rather than information about objects in isolation. For example, consider the relation between persons and the companies they work for. Such a relation could be written:

```
RELATION Works_for
    (employee: Person, employer: Company)
```

A diagram of this relation is shown in Figure 1a. Object classes are shown as boxes containing the name of the class. Relations are shown as lines connecting two classes, with the name of the relation near the line. The black dot indicates that each company may be associated with many persons.

The value of a relation is a set of object tuples which represents part of the state of the world at a particular moment. For example, the "Works for"

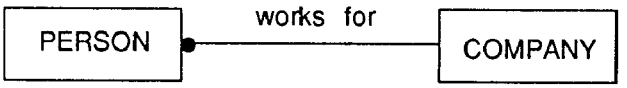

Figure 1a. Data Model of a Relation

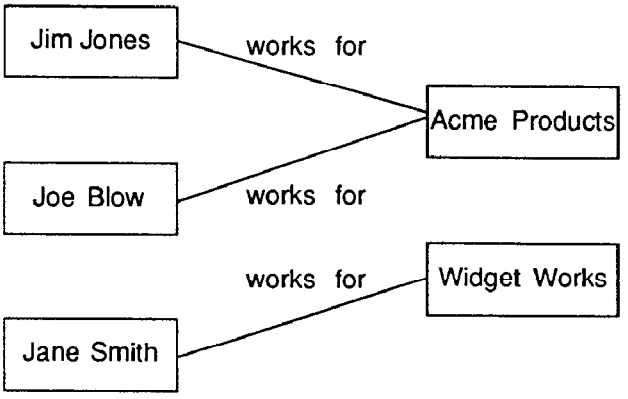

Figure 1b.
Object Instances and Elements of the Relation

relation might have elements ("Jim Jones", "Acme Products"), ("Joe Blow", "Acme Products"), and ("Jane Smith", "Widget Works"). This situation is diagrammed in Figure 1b; object instances are drawn as boxes connecting to related objects by lines. Note that the relation exists between the objects themselves, not their attributes, although in giving examples we identify objects by their unique attributes (such as their names).

2.3 Update operations

The value of a relation can be changed by adding elements to it or deleting elements from it; these operations correspond to changes in the real-world situation abstracted by the relation. Each operation requires one value for each field. For example, the following sequence of operations (written in a C++ style syntax) would produce the state of the "Works for" relation shown in the figure:

```
Works_for.add (Jim Jones, Acme Products)
Works_for.add (Joe Blow, Acme Products)
Works_for.add (Jane Smith, Widget Works)
```

The value of a relation is a set, so adding an element that already exists does not change its value. Similarly, deleting an element not present in the relation does not change its value. In such cases, a particular implementation might choose to raise an exception or silently ignore it.

2.4 Query operations

The value of a relation can be queried with operations to test membership of an element, select all the elements that match certain fields, or scan all the elements.

2.4.1 Testing membership

A membership test requires one value for each field, and returns a boolean value. Here are some membership tests of the "Works for" relation:

```
Works_for.test_member (Jim Jones,
    Widget Works) returns False
Works_for.test_member (Jim Jones,
    Acme Products) returns True
Works_for.test_member (Acme Products,
    Jim Jones) returns False
```

The order of the objects within an element is significant, as shown by the last example.

2.4.2 Indexing by fields

Relational data bases support general queries which select from a relation all the elements whose fields satisfy an arbitrary boolean expression. Such generality may not need to be a primitive in an object-oriented language, but at the least it is necessary to be able to select all the elements in which one or more designated fields match specified values. The *index* operation is defined on some subset of the fields of a relation, called the *index set*. A value is required for each field in the index set; the index operation returns the set of elements whose fields match the index values. Since the index values are already known, it is convenient to ignore them in the returned values, and consider the operation as returning a set of elements of reduced degree. Indexing a binary relation by one field returns a set of values from the other field. For example, to find all employees of a company:

```
Works_for.index_2 (Acme Products)
    returns {Jim Jones, Joe Blow}
Works_for.index_2 (Widget Works)
    returns {Jane Smith}
Works_for.index_2 (Marvelous Manufacturing)
    returns { }
```

The suffix "_2" indicates that the index set is the second field. Either field in a binary relation can serve as an index. To find a person's employer:

```
Works_for.index_1 (Jim Jones)
    returns {Acme Products}
```

There is one index operation for each subset of fields. For a relation of degree n, any subset k of the n fields can serve as an index, returning a set of elements drawn from the remaining n-k fields. An implementation would not necessarily need to support all $n!$ modes of indexing.

2.4.3 Scanning elements

Since a relation is a set, an operation to scan all its elements is necessary. The scan operation applies a specified operation to each element of the relation in turn. The order of scan depends on the implementation; a program should assume that the elements are unordered. General queries can be constructed using the scan operation, but the index operation is more direct when it can be used.

Given these basic operations, various composite operations can be constructed, such as deleting all occurrences of a value in a given field. A particular implementation might define some of these composite operations as primitives for efficiency.

2.5 Cardinality

It is often convenient to constrain a relation by specifying the *cardinality* of one or more of its fields. The cardinality is the number of different values from the specified field that can be associated with a given set of values for the other fields. Cardinality can be specified as an integer interval. For example one car is associated with four tires and from four to eight spark plugs. Most of the time, it is sufficient to distinguish scalar fields (cardinality zero or one, "one" fields) from set fields (cardinality zero to unbounded, "many" fields). For example, if each person works for a single company, then the "Works for" relation is many-to-one from persons to companies, that is, many persons may be associated with each company, but one company is associated with each person. In general, every object of a class need not appear in the relation. An unemployed person would not appear in the "works for" relation. In some cases, it is useful to indicate that the cardinality of a relation must be nonzero. For example, if we replace "Person" with "Employee" in the "Works for" relation, every instance of class "Employee" must be associated with an instance of class "Company"; class "Employee" is said to be *existence dependent* on class "Company".

The cardinality of each field can be declared as follows:

```
RELATION Works_for (employee: Person|*,
    employer: Company|0–1)
```

where the cardinality is shown as an interval range and the star shows an unbounded cardinality. In a diagram, a black dot shows a set of values, a "many" value. A simple line shows a single required value, a "one" val-

ue. An open circle shows a single value that is optional, a "zero or one" value.

Cardinality constraints represent an important aspect of the real-world situation being abstracted which is absent in many models. The constraints cannot be validated from within the model, but must be determined by the real-world situation and its relevance to a particular application. For example, the relation "married-to" would be one-to-one in the United States, but one-to-many in Saudi Arabia. On the other hand, if the "married-to" relation represents marriage partners over time, then it would be many-to-many even in the United States. Choosing cardinality constraints forces the designer to confront assumptions early which are frequently buried in the code. They also force the designer to decide how special cases and exception conditions will be handled.

The cardinality of the set returned by an indexing operation is necessarily consistent with the cardinality of the relation. For example, because "Works for" is a many-to-one relation, indexing it by "Person" yields a set containing a single company object. An implementation can take advantage of this constraint to store an associated value as a scalar rather than a set. An implementation of the operations must guarantee that cardinality constraints always remain valid and that duplicate elements do not appear in the relation, by rejecting operations that would violate the constraints or defining side effects that preserve the constraint.

2.6 Similarities between classes and relations

A class and a relation can both be thought of as objects with fixed descriptions and variable states which reference other objects. The description part of a class includes its superclasses and subclasses, instance variables, and methods; the state part is the set of instances of the class. The description part of a relation includes its degree, cardinality, and list of fields; the state part is the set of elements of the relation. Specification of an object-relation model for an application requires that the descriptions of the relevant classes and relations be given. Execution of an algorithm based on the model generates successive states of the objects.

2.7 Qualified Relations

So far only binary relations have been discussed in detail, although the semantics of relations have been described for relations with any number of fields. A special kind of ternary relation, called a *qualified relation*, arises frequently enough to merit special treatment. They come about as follows: Often a one-to-many relation exists between two classes: call them the source class and the target class. Each object in the source class is related to a set of objects in the target class. To distinguish among the set of target objects, they are given names unique within the set. A set of names is associated with each source object, naming the target objects associated with it. Each set of names is local to the source object it qualifies. Another source object might share all, some, or none of the names. A name qualifies the source object to identify a unique target object. A source object and a name, taken as a pair, are associated with a unique target object. There is one-to-one relation between the (source object, name) pair and the target object. Equivalently, there is a ternary relation among the (source object, name, target object) classes. Such a relation is called a qualified relation.

Qualified relations occur when there is a set of names, or some other set of qualifiers, that serves to distinguish the target elements in a one-to-many or many-to-many relation. For example, a directory in a file system contains many files; a file name unique within the directory distinguishes them. A qualified relation could be declared as:

```
RELATION File_system
    (Directory[filename:Name]|1, File|1)
```

to show that each (directory, name) pair identifies a unique file and each file identifies a unique (directory, name) pair. The brackets indicate that the name qualifies the directory, similar to an array index which qualifies an array. The cardinality applies between the (directory, name) pair and the file. If links are supported, each file could have many names, so the declaration would be:

```
RELATION Link_file_system
    (Directory[filename:Name]|*, File|1)
```

to show that each (directory, name) pair identifies a unique file but each file is associated with a set of (directory, name) pairs. Figure 2 shows a diagram for this many-to-one qualified relation.

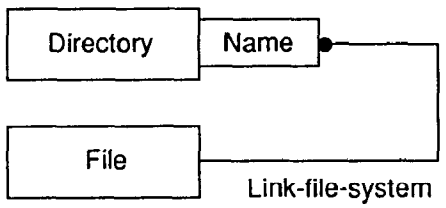

Figure 2. Qualified Relation

Examples of qualified relations come to mind whenever a set of names or other identifiers is associated with an object. For example, the name of a state iden-

tifies a particular state associated with a country; the name of a city identifies a particular city associated with a state; the name of a street identifies a particular street associated with a city:

```
RELATION World_country
    (World[Name]|1, Country|1)
RELATION Country_state
    (Country[Name]|1, State|1)
RELATION State_city (State[Name]|1, City|1)
RELATION City_street (City[Name]|1, Street|1)
```

The first relation is included for uniformity, although most applications would contain only a single instance of "World". Representation of the information as a qualified relation makes its bidirectional status clear. Given the names of a country, state, city, and street, a unique street object can be found. Conversely, given a street object, unique names can be found for the country, state, city, and street.

The qualifier need not be a name, as long as it distinguishes target objects. The following declarations show some more examples:

```
RELATION Gate_pins
    (Gate[pin_number:Integer]|1, Pin|1)
RELATION State_machine
    (initial:State[input:Token]|*, final:State|1)
RELATION Officers
    (Company[Office]|*,officer:Person|1)
```

The first relation associates gates with pins on the gate, using the pin number. Each pin on a gate has a unique pin number. The second relation describes a state machine. An initial state and an input token produce a final state. A given final state may have been produced by more than one (initial state, input token) combination. The third relation describes the officers of a company. In this example, "Office" is an object class that describes the office, rather than a name. Each office is held by one person; one person can hold several offices, in the same or different companies.

Operations on a qualified relation are simply special cases of the n-relation. To add or delete an element or test membership, the values of all three fields must be given. There are 3! indexing sequences possible on three fields, but it is convenient to require that the qualifier field not be given without the source field, i.e. to disallow searches that start with the index field. There is no mathematical justification for this restriction, but it permits the qualified relation to be manipulated as an extended binary relation, which simplifies both the notation and the implementation. (For those situations where this restriction is unacceptable, a full ternary relation can be used.) A qualified relation can be indexed by the first field and qualifier field to yield values from the second field. It can be indexed by the second field to yield pairs of values from the first field and the qualifier field. This can be factored on the first field to yield a table of qualifier values indexed by values from the first field. Finally, it can be indexed by the first field alone to yield a table of values from the second field indexed by values from the qualifier field. These can be written:

```
Link_file_system.index_1q (Directory, Name)
    returns File
Link_file_system.index_2 (File)
    returns Table [File] of Name
Link_file_system.index_1 (Directory)
    returns Table [Name] of File
```

A qualified relation is appropriate wherever a name of local scope is used to discriminate among a set of related objects. In general, a name is a qualifier on a set of objects. Whenever a name is felt to be global or unique, the model can usually be recast in a more general form; on deeper inspection unique names are found to be unique with respect to some other object, such as a catalog, organization, and so on. Programs written with the assumption of unique names often have to be rewritten later; it is better to represent names using qualified relations from the beginning. In most applications, we have used qualified relations heavily; as many as half the relations in certain applications are qualified relations.

2.8 Other Relations

Other variations on ternary relations can be defined, but they are of lesser utility than qualified relations. We have developed a general formulation for relations of any degree, in which cardinality constraints are expressed by listing all the subsets of fields that form candidate keys of the relation. A set of fields could be an index set, provided the possible index sequences are specified in advance and used by the system to maintain indexing tables. This approach will be described in detail in a future paper. We have found that relations with more than three fields are rarely if ever needed in practice.

3. RELATIONS IN STANDARD OBJECT ORIENTED LANGUAGES

A standard object-oriented language (such as Smalltalk) has two kinds of abstraction structuring mechanisms which relate different objects: *instantiation* ("an instance of") and *generalization* ("a kind of", the class hierarchy). Instantiation is a relation between class descriptor objects and instances of the class. Generalization is a relation between pairs of class descriptor objects, the superclass and the sub-

class. These special relations are built into the semantics of the language and are supported by special syntax; they cannot be accessed as discrete objects. Relations between ordinary object instances are not supported by syntax or semantics. They can be simulated by the use of instance variables that refer to other objects. If the relation must be traversed in more than one direction, an instance variable is required in each participating object.

3.1 Implementing relations using instance variables

Representing a relation between two objects as an instance variable in each object fails to capture the semantics of the relation. The information about the relation is distributed among two classes, rather than being specified in one place, making it harder to understand and maintain. The constraint that related objects must mutually reference each other cannot be explicitly expressed. A programmer cannot represent a relation between two classes without choosing an implementation, including choice of instance variables and methods, and exposing much of the implementation. The conceptual and implementation levels cannot be kept distinct in representing relations, because there is no semantic support for relations in languages such as Smalltalk.

For example, consider the implementation of the "Works for" relation shown in Figure 3. Each person has a pointer to the employer, while each company has a pointer to a set of pointers to employees.The following class declaration fragment implements the "Works for" relation using pointers:

```
CLASS Person
INSTANCE VARIABLES
  employer : Company
METHODS
  put_employer (Company)
CLASS Company
INSTANCE VARIABLES
  employee : Set of Person
METHODS
  add_employee (Person)
  delete_employee (Person)
```

The instance variables Person.employer and Company.employee are related, but there is no way to express this constraint in an object schema declaration. Public update methods Person.put_employer, Company.add_employee, and Company.delete_employee must maintain the constraint as elements of the relation change by updating instance variables employer and employee. A method such as Person.put_employer must have access to the instance variables of

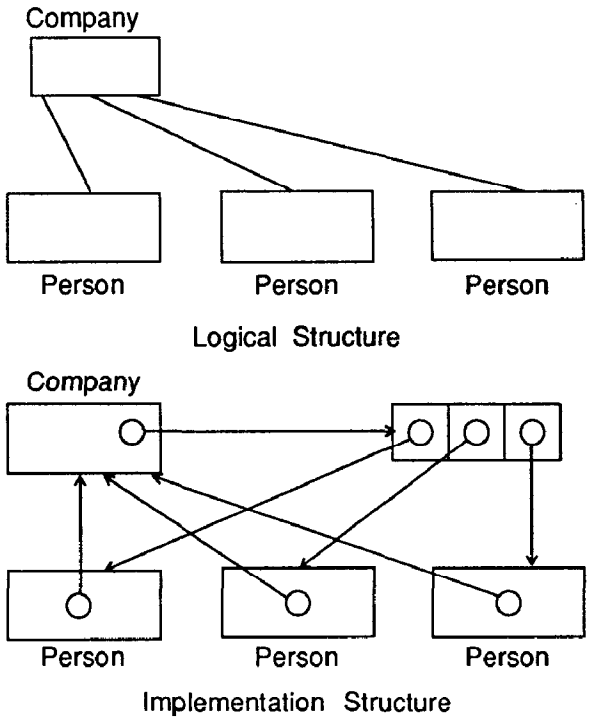

Figure 3. Implementation of "Works for" relation using pointers

"Company" (perhaps using the "friend" mechanism of C++). The alternative is another layer of private methods, such as Company.raw_add_employee, which allow the public methods, such as Person.put_employer, to cause the instance variable to be updated. (Calling Company.add_employee within Person.put_employer would cause an infinite loop.) If the language lacks the "friend" mechanism, it may be impossible to hide the raw update methods yet still allow the public methods to be written, without exposing the internal structure to public view and the danger of inconsistency. In any case, each update method involves modifications to several instance variables.

A simple update requires several lines of code to implement. This leads to opaque code and introduces the danger of inconsistencies in the data structures if one pointer is updated independently. Another disadvantage of representing relations between objects as pointers is that an instance variable must be reserved in each object instance for each relation that an object of a given class can participate in. This is no problem for a dense relation, in which most or all instances of a class will participate in the relation, but it tends to discourage the use of sparse relations, since the cost must be paid by every instance of the class, even if few instances participate in the relation.

3.2 Duality of relations and instance variables

Relations and instance variables can be mapped into one another. It is easy to represent a binary relation as a pair of instance variables on the respective classes, each of which holds a set of values from the other class. The two instance variables are mutually dependent; this constraint must be expressed in the method code rather than the object schema. This mapping increases the mechanism baggage that the designer must deal with; relations provide more semantic information in a more transparent form. On the other hand, an attribute of a class can be represented by a many-to-one relation between the class and the class of the attribute. Attributes and many-to-one relations are both logically equivalent to discrete partial functions. The replacement of an attribute by a relation does not increase the complexity of the representation for the designer, because the same amount of information needs to be specified. Use of a relation can reduce the complexity of the algorithm, because a relation can inherently be traversed in either direction, while an instance variable can be followed only in the "forward" direction. We have found it profitable to represent all attributes as relations in the initial design of an object model, and to consider instance variables as simply implementation optimizations for cases where traversal in the "reverse" direction is not needed.

3.3 Why adding a relation class is not enough

In a standard object-oriented language it is possible to define a collection class "relation" whose instances represent the values of particular relations and whose methods implement the operations proposed for relations. This is useful in simplifying the implementation of relations, but it fails to separate the relation as a logical construct from the relation object as an implementation tool. Individual relation objects must be instantiated at run time as part of the application code. It is desirable to build syntactic and semantic support for relations in the language, similar to the support for classes, for the following reasons:

- There may be more than one possible implementation of a logical relation. A programmer should be able to choose the implementation using an option flag on the declaration, without changing the code that uses the declaration or even most of the declaration itself.
- The compiler can implicitly instantiate and initialize relations at the beginning of program execution, just as object classes are instantiated and initialized.
- The language can provide special syntax to simplify operations on relations, just as special syntax is provided for method application.
- The compiler can automatically generate methods on the participating object classes to access and update the relations.
- Object classes will have a list of relations they participate in, represented in a uniform way. This information can be used in writing generic methods to destroy objects and clean up relations they participate in, to copy objects and objects they are related to, and to pretty-print objects along with objects they are related to.
- Most importantly, treating relations as important built-in semantic constructs changes the way programmers abstract and formulate problems. Thinking in terms of objects and generalization hierarchies is generally unfamiliar at first, but eventually changes the way a programmer thinks about a problem. We have found from experience that making relations a first-class semantic construct affects a programmer's way of thinking about a problem from the design stage all the way through to the coding. This new way of thinking is particularly useful for formulating and partitioning designs.

4. IMPLEMENTATION OF RELATIONS IN AN OBJECT-ORIENTED LANGUAGE

4.1 Syntax

The syntax of declaring relations should be parallel to the syntax for declaring classes, in accord with their joint status as first-class semantic constructs. An object class schema declaration consists of both class definitions and relation definitions, neither subordinate to the other. The definition of a relation requires the information described under the logical model, namely its name, degree, cardinality constraints, and a list of object classes for the fields. Each field can be given an optional name for convenience. The compiler automatically instantiates and initializes each declared relation at program initiation. When a program begins execution, each relation contains no elements, just as each class contains no instances. The names of relations have global scope, as do the names of classes.

4.2 Methods

The methods applicable to relations are attached to class "Relation". There are subclasses for special cases, such as "Binary relation" or "Qualified relation". The methods are the ones described under the logical model, namely "add element", "delete element", "index" (in several varieties, according to the index fields supplied), "test membership", and "scan." The methods would be invoked on the relation object

itself. This is similar to invoking a class method on a class, such as "new", in that the name of the object in both cases is a proper name known to the compiler, and designates a predefined object initialized implicitly as a consequence of making a declaration. It is not necessary for users to define new methods on particular relations, because they are not classes and do not describe instances. (It is possible to define new relation subclasses corresponding to individual relations or groups of relations for the purpose of overriding the predefined methods. This would define a relation hierarchy, similar to the class hierarchy, and permit new methods to be defined on some relations. We have not explored this concept yet.)

The methods that update a relation must guarantee that the cardinality constraints are never violated. An implementation may define whether an update that would violate a cardinality constraint is rejected with an error status, or whether the conflicting elements are deleted from the relation as an implicit side-effect of the operation, as long as the resulting state is valid.

For purposes of information hiding it is desirable to restrict access to a relation to the classes participating in it. Methods on participating classes can access the relation freely, as with instance variables in classes, but methods on other classes have access to the relations only indirectly, through methods on the affected classes.

To closely model natural ways of thinking about relations, it is convenient to support two complementary ways of applying methods to them. The first way treats a relation as an object to which a method is applied. For example, the syntax for adding an element to the "Works for" relation might be

```
works_for.add
    (Susan_Hill, Marvelous_Manufacturing)
```

and the syntax for indexing by a value from the first field might be

```
works_for.index_1 (Susan_Hill)
```

returning "Marvelous_Manufacturing". The second way treats a relation as analogous to an attribute on one of the participating classes; the target of the method is an object from one of the classes, rather than the relation as a whole. For example, the syntax to add a worker to a company might be

```
Marvelous_Manufacturing.add_worker
    (Susan_Hill)
```

An alternate form would be

```
Susan_Hill.put_employer
    (Marvelous_Manufacturing)
```

Both forms have the same effect. A good object-oriented language preprocessor can automatically generate access methods on the participating classes using role names; an option flag on the relation declaration indicates that the generated methods are wanted.

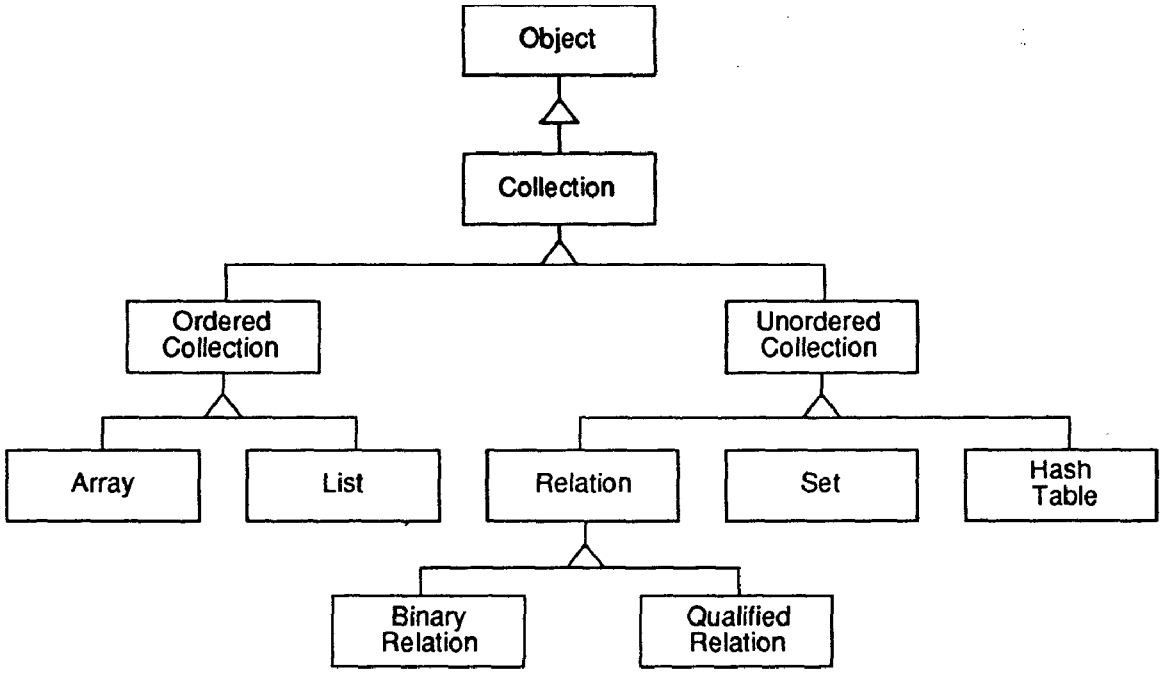

Figure 4. Collection class hierarchy

4.3 Relation objects

Relation objects can be implemented so that access is efficient. The most straightforward implementation is to have an actual relation object for each declared relation. Class "Relation" is a subclass of class "Unordered Collection" and is similar to class "Set" (Figure 4). Relation objects contain a description part and a variable-length value part. The description part contains the degree of the relation, a list of fields, and the cardinality constraint. It may be unnecessary to implement fully general cardinality constraints; in practice, it is useful to support cardinality "zero or one" and cardinality "many". The value of a relation is a set of tuples of values from the respective object classes. The value could be represented as a set object, but this would make indexing inefficient, as it would be necessary to scan the set to find field values that match the arguments.

4.3.1 Internal Structure

To permit efficient access for indexing, it is necessary to build an index table for each desired index order. For example, a binary relation can be accessed by field 1 to yield field 2, or by field 2 to yield field 1. Two index tables are required, one mapping field 1 values to field 2 values and one mapping field 2 values to field 1 values. A qualified relation can be accessed by field 1 to yield the qualifier and field 2, by field 1 and the qualifier to yield field 2, or by field 2 to yield field 1 and the qualifier. Two index tables are required, one mapping field 1 values to nested tables mapping qualifier values to field 2 values, the other mapping field 2 values to subtables mapping field 1 values to qualifier vaues. An index table is implemented using hash tables, which permit a lookup operation to be performed in constant time. (The time is larger than a simple pointer access, but does not increase with the size of the set to be searched. On the Sun-3 workstation, a call to find an integer in a hash table is more efficient than a for-loop to search for an integer in an array of 8 or more elements.) For example, the value in a binary relation would be stored in two hash tables: one table mapping field 1 values to sets of field 2 values, and another table mapping field 2 values to sets of field 1 values. Indexing by either field would be equally efficient, essentially in constant time regardless of the size of the relation. Testing membership of an element would require two steps: the field 1 value is used to index the first table, returning a set of field 2 values; then the field 2 value is tested for membership in the set. Adding an element would require that both tables be updated; for most applications accessing a relation is much more frequent that updating it, so it is desirable to optimize access time at the expense of update time (and also at the expense of space).

The cardinality constraints can be used to reduce the storage space needed for fields of cardinality "one" by storing scalar values in the appropriate hash table, rather than sets of values. In order to maintain the cardinality constraints, it is necessary to check for conflicts before adding a new value to a constrained relation; any conflicting elements must be deleted from all the hash tables before the new value is added. For convenience of the programmer, an indexing operation that returns a field of cardinality "one" can return a scalar value, rather than a set containing a single element.

Relation objects can be created at run time as well as being defined in the object schema. Such anonymous relations must be manipulated by object ID, as they have no predefined global name. A new, empty relation object is created by a class method "new" on class "Relation". Named relations can be considered as special cases implicitly created by the compiler and initialized when the program begins execution.

4.3.2 Implementation Benefits

The use of relation objects as implementation constructs has certain advantages. Using relations externalizes information, rather than internalizing it as part of object records. This makes the implementation of sparse relations more efficient. It is possible to add a new relation, without modifying the existing structure of an object record; this might be important in a system in which data types could be dynamically modified. A relation object groups information which would otherwise be distributed among several objects, and permits the application of operations to the relation as a whole, such as scanning or copying an entire relation. Such operations are particularly useful in implementing persistent storage of objects, that is, storage of objects to a permanent data base between program executions. The main theoretical problem with persistent storage of objects is in representing inter-object references in external storage, and remapping all or some of the references into a new context when the stored objects are reloaded into an existing object set. If the use of pointers to objects is avoided in implementing objects, then all inter-object references reside in relations, which contain only object IDs. The fact that all the inter-object references are in one place, in a highly symmetrical form, can greatly simplify the task of writing a persistent storage manager.

5. POSSIBLE OBJECTIONS

An objection could be raised that use of relations compromises information hiding, because the classes appearing in a relation must know about each other and the behavior of one in updating a joint relation can affect the other. This is true, but we do not feel that this is a drawback, but a virtue. A relation represents an inherent constraint between objects of two or more classes. This constraint is not something to be hidden, but rather to be specified abstractly, without imposing an implementation. In a Smalltalk program, the constraints would be buried within method code, hard to recognize. This kind of "information hiding" hides semantic information and exposes implementation information, exactly backwards. Use of relations need not compromise true information hiding. All accesses to a relation can be restricted to methods defined on the affected classes. Any access to an object from another class must use one of the defined methods, rather than accessing the object's instance variables or associated relations directly. For example, most classes will have a set of "get" and "put" methods to manipulate the values of instance variables. In most cases, these translate directly into reading or writing the instance variable value, but in some cases there are side effects of an update which may be hidden from the caller. A similar set of "get" and "put" methods could be defined for the relations affecting a class. For example, the operation "put_employer" applied to an instance of "Person" would update the "Works for" relation. Unlike the update of an instance variable, the update of a relation element would implicitly affect more than one class. This is not a breakdown of information hiding. A relation indicates a situation in which two (or more) classes are interlocked in some way. It simply would not make sense to update an instance variable in one class, and leave the corresponding instance variable in the related class unmodified. However, the traditional view of strict information hiding in separate classes does not allow these constraints to be specified cleanly, and as such is flawed for representing real systems.

Why bother with relations, one might say. After all, can't they be implemented as instance variables? This objection misses the point, indeed the entire point of object-oriented programming, which is to match the computer model more closely to the conceptual real-world model and to avoid introducing implementation constraints on the design. The object-oriented model itself is logically unnecessary, because any computation can be represented as a Turing machine (or at least in assembly code, which is not too much different). Instance variables are an implementation construct. Representing a binary relation as a pair of mutually interlocked instance variables loses semantic information, because the standard object-oriented model cannot represent the constraint that the two objects must point at each other. Not only do relations carry more semantic information, but they provide a concise, symmetric way of describing information that is not subordinate to any one class.

All this may be very good, but won't these ideas be too hard to implement, and won't they be too inefficient in any case? No. All the concepts discussed here, and other variations also, have been implemented in an object-oriented language written by the author. This language has been used for several applications, including interactive graphics, with very satisfactory performance.

6. AN ACTUAL IMPLEMENTATION

The author and colleagues have implemented two tools incorporating the concepts presented in this paper: the Object Modeling Technique [Loomis et al] and the Data Structure Manager [Rumbaugh].

The Object Modeling Technique is a notation for drawing object models, which includes representations for class generalization hierarchies, relations among objects, aggregation trees, and other things. We have used some of this notation for the examples in this paper. A graphical tool to draw object models and automatically produce Data Structure Manager declarations is being written.

The Data Structure Manager (DSM) is an object-oriented programming system written by the author. It is a fully-implemented, production-quality object-oriented programming system, intended to support C language programming using an arbitrary mixture of straight C language code together with extensions which add object-oriented capabilities. DSM contains syntax and built-in object classes to fully support relations, as well as the features found in other object-oriented languages. A brief description of DSM is given in the appendix.

These tools have been used to model and implement several large interactive applications with excellent results. Our experience has shown that the use of relations greatly enhances the modeling process. We have used object-relation models to design several systems, such as a chemical plant layout system, which were then implemented as relational data bases. This notation greatly facilitated communication with clients, some of whom were not computer experts; they found it intuitive and easy to learn with a few minutes' explanation. The Data Structure Manager has been used to implement several large interactive applications, including a PHIGS-like hierarchical graphics package.

The use of relations greatly simplified design and implementation of these applications. Performance has not been a problem. The DSM compiler and the DSM run-time package are entirely written using DSM objects; these programs are large and complex and have heavily exercised many of the features of the language.

Our experience has shown that the greatest advantage of object-oriented programming is greatly improved ease of modifying programs. We have found that even substantial changes to a system do not propagate very far if the system is partitioned well. For example, we added multiple inheritance to the DSM compiler and run-time package in two weeks, including debugging. None of these applications would have been nearly as easy without the availability of relations. Descriptions of the Object Modeling Tool, the Data Structure Manager, and our experiences with using these techniques on actual applications will appear in future publications.

7. FURTHER WORK

The concept of relations in an object-oriented language could be extended in a number of ways:

- Hierachies of relations. These can be implemented as suggested in section 4.2, but their semantics require clarification. For example, is "Husband-of" a subrelation of "Married-to"?
- Making elements of relations first class objects. This adds logical power, but may be difficult to implement efficiently.
- Symmetries and other constraints within relations. The transitive closure of a symmetric relation occurs often in practice and seems to call for a special implementation.
- Constraints on sets of relations. Often the elements of two different relations are not independent. Perhaps this is a case of relations between relations themselves.
- Derived relations. It is often convenient to build intermediate objects and relations that are strictly derivable from a set of independent objects and relations. If a programming system could construct and maintain such derived relations from a set of declarations, considerable simplifications of algorithms would result.
- Views of relations. This can be considered a special case of derived relations. A lot of effort has been devoted to views in the database area, with mixed success.
- Use of relations in saving and restoring object-oriented data to a permanent store or a database. Relations could be assigned different binding strengths. If an object is saved, all objects related to it by strong relations (such as "Part of") would be saved with it, but objects related to it by weak relations would be regarded as separable and could be saved and restored independently. The concept of external names and catalogs is important to the problem of restoring data in a different context than it was saved.

8. CONCLUSIONS

The use of relations as a conceptual construct in an object-oriented environment can help capture the semantics of a system more clearly than the use of object pointers. Promoting relations to an equal footing with classes permits a symmetric and compact representation of a highly-interconnected system of classes. Binary relations and qualified ternary relations seem adequate to model most applications. Specification of cardinality is useful in tightening semantics and reducing unnecessary operations. All these kinds of relations can be implemented efficiently using hash techniques. The externalization of inter-object references into relations can aid in dumping and reloading object values to databases and to external formats. We have implemented these concepts in an object-oriented language and found them to be useful, practical, and natural.

ACKNOWLEDGMENTS

I wish to thank Mike Blaha, Paul Brown, and Ashwin Shah for their valuable comments on the manuscript.

REFERENCES

Blaha, M.R., W.J. Premerlani, and J.E. Rumbaugh. Relational database design using an object-oriented methodology. Document submitted for review.

Bobrow, Daniel G., Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: merging Lisp and object-oriented programming. *OOPSLA Conf. Proc.*, Portland, 1986, 17-29.

Chen, P.P. The entity-relationship model: toward a unified view of data. *ACM Transactions on Database Systems 1*:1 (March 1976), 9-36.

Codd, E. A relational model for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377-387.

Fishman, D.H., et al. Iris: an object-oriented database management system. *ACM Trans. on Office Information Systems 5*, 1 (Jan. 1987), 48-69.

Goldberg, Adele, and David Robson. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, Reading, 1983.

Khoshafian, Sertag N., and George P. Copeland. Object identity. *OOPSLA Conf. Proc.*, Portland, 1986, 406-416.

Korth, Henry F. Extending the scope of relational languages. *IEEE Computer 3*, 1 (Jan. 1986), 19-28.

Lieberman, Henry. Using prototypical objects to implement shared behavior in object-oriented systems. *OOPSLA Conf. Proc.*, Portland, 1986, 214-223.

Loomis, Mary E.S. *The Database Book.* Macmillan, New York, 1987.

Loomis, Mary E.S, Ashwin V. Shah, and James Rumbaugh. An object modeling technique for conceptual design. *European Conference on Object-Oriented Programming*, Paris, June 1987.

McAllester, David. Boolean classes. *OOPSLA Conf. Proc.*, Portland, 1986, 417-423.

Meyer, Bertrand. Genericity versus inheritance. *OOPSLA Conf. Proc.*, Portland, 1986, 391-405.

Pascoe, Geoffrey. Encapsulators: a new software paradigm in Smalltalk-80. *OOPSLA Conf. Proc.*, Portland, 1986, 341-346.

Rumbaugh, James. *Data Structure Manager Reference Manual.* GE Internal Document.

Snyder, Alan. Encapsulation and inheritance in object-oriented programming languages. *OOPSLA Conf. Proc.*, Portland, 1986, 38-45.

Stefik, Mark J., Daniel G. Bobrow, and Kenneth M. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Computer 3*, 1 (Jan. 1986), 10-18.

Teorey, Toby J., Dongqing Yang, and James P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys 18*, 2 (June 1986), 197-222.

Ullman, Jeffrey D. *Principles of Database Systems.* Computer Science Press, Rockville, 1982.

APPENDIX A: THE DATA STRUCTURE MANAGER

The Data Structure Manager (DSM) is a programming development system to support object-oriented programming in the C language. The DSM system implements all the concepts of a standard object-oriented system, such as Smalltalk, within the context of the C language, while adding a number of extensions, such as relations, which greatly extend its power. It comprises a set of preprocessors and a subroutine library written in C. The DSM system provides a declaration format in which a user can define an object class hierarchy, including single and multiple inheritance and relations between classes. Classes and relations are first-class objects, and a full metaclass hierarchy is supported, including class variables and class methods, similar to Smalltalk. The DSM system is written entirely using itself. All system objects are created dynamically at run time and are fully extensible. The system performs memory allocation and deallocation, but does not include a garbage collector, because garbage collection is fundamentally impossible with an open language such as C that permits arbitrary user operations. Class descriptor objects and relation descriptor objects provide a full internal description of each class and relation at run time for symbolic operations, such as pretty printing of objects or transfering objects to or from permanent storage.

DSM is intended to be used for production applications, including quasi-real-time applications such as graphics, so it was designed to be efficient. The programmer is given a choice of constructs in several situations, allowing a conscious trade-off between generality and efficiency when needed. For example, a method can be called using a run-time method lookup based on the object class, as in Smalltalk. Hashing is used for efficiency, but the programmer has the option of storing a pointer to a method function in a class variable or of calling a specific C function directly if the class of an object is known at compile time. To facilitate C-language programming, instance variables can have C types as values, so that objects need not be created to hold pure values, such as integers or structures of values; an object ID is a special case of a C data type. The class hierarchy has been replaced by a C type hierarchy, within which classes are simply one subtype. Since DSM programs are written in C, it is easy to interface to non-DSM programs in C or other languages.

The DSM system is written in normal C using the standard I/O package, and has been easily ported to several different systems, including Sun, VAX VMS, VAX Ultrix, Apollo and HP workstations, and the IBM PC. It is used to implement itself. It includes an object pretty-printer and a run-time interpreter for displaying data structures and debugging programs. It has been used to implement several large application packages, including a generic interactive graphics system (with capabilities similar to PHIGS) and a user interface system for managing program execution and file storage using interactive graphics on a workstation. It is only slightly less efficient than writing operations in straight C.

Implementation of Relations in DSM

All objects in DSM are created dynamically at run time, including class descriptors, relations, and other internal objects. The package includes a predefined set of object classes which are generically useful and which are used to implement DSM itself. These

include strings, symbols, and several dynamic aggregate data types, including arrays, sets, lookup tables (discrete functional mappings), and relations. The support given to relations in the subroutine library and the declaration notation is perhaps the most novel feature of the DSM system.

Relations are implemented in DSM as fully indexed lookup tables. A binary relation contains two tables: one maps field 1 values into field 2 values or sets of field 2 values (depending on the cardinality of the relation), and the other maps field 2 values into field 1 values or sets of field 1 values. When an update is performed, both tables are updated. When an index operation is performed, the table containing the field to be indexed by is used. No searching is needed at any time, since hashing is used, so relational update and access operations operate in constant time on average regardless of the size of the relation.

The lookup table object is a built-in DSM data class. A table is a functional mapping from one field to another. It is implemented using open-chain hashing for constant time average access. The hash table is never allowed to become full. If a hash table exceeds a certain fraction of its allocation, then its storage allocation is multiplied by a constant factor and the entire table is reallocated and rehashed. Because the table grows by a multiplicative factor, the average cost of reallocation is a constant factor of the cost of a simple update over the life of the table and does not cause a non-linear cost. This storage technique minimizes the time cost of exact-match associative updates and retrievals, at the price of a fixed multiplicative factor of unused memory necessary to permit efficient hashing. The attempt to minimize execution speed at the cost of memory usage is justified on most small to medium applications on modern virtual-memory machines. The DSM implementation consistently attempts to minimize execution time at the possible expense of storage space.

APPENDIX B: NOTATION

A good graphical notation for object classes and relations can aid in the development and use of a set of class definitions. A portion of the Object Modeling Technique [Loomis et al] has been used for examples. Object classes are shown as rectangular boxes containing the name of the class. Within the object boxes, separate sections show lists of instance variables and lists of methods. Object subclasses sprout off a triangle connected to the superclass. Relations are shown as lines between classes, with the name of the relation written on the line. The cardinality of each end of the relation can be indicated by an integer range (such as 1-4, 0+, etc.); the special case of 0 or more is shown as a black dot, of 0 or 1 is shown as an open circle, and of exactly 1 is shown as a simple line. A qualified relation is indicated by a small box containing the qualifier attached to the field 1 class box, connected by a line to the field 2 class box.

Figure 5 shows a sample data model containing some information about a company and its employees. The model is incomplete, perhaps an early stage of a design. No methods have been shown yet; they typically are added much later. Only two subclasses are shown; they are much less important than relations in an early design. Certain restrictions can be seen from

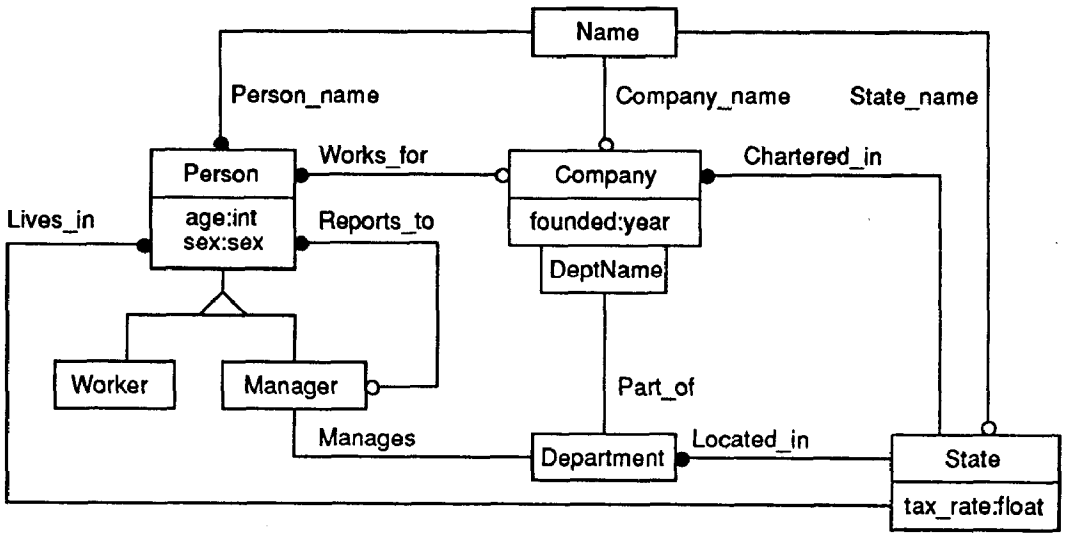

Figure 5. Sample data model

the diagram: each person can only work for a single company, each department is located in a single state, each company has a unique name, each state has a flat tax rate, several persons with the same name might exist. One can make up questions and see if the information to answer them is in the model. The following kinds of questions can be answered: who are the employees of a company, what are its departments, what are all the states in which it has departments, what is the reporting hierarchy from a worker up to the top. Some questions cannot be answered: which department does an employee work for, where does an employee work, what are the organizational entities corresponding to the reporting hierarchy. A realistic model would contain much more detail. There is no structure to the corporate organization. Perhaps "Department" should be replaced by "Organizational unit", which would be recursive to match the reporting hierarchy. "Tax table" would have to be an object of some complexity. Maybe we should distinguish "Person" from "Employee"; a person might be an employee of more than one company (perhaps over time) and we might want to group all the information about a single employment instance in a separate object, which would be related to the person object. The object model diagram is an invaluable tool to designing an object-oriented system correctly with proper regard for the overall picture.

APPENDIX C. SEMANTIC DATA MODELS.

Researchers in data base theory have explored a number of models for data structuring which depend heavily on the concept of relations. The relational data base model [Codd] is the earliest and simplest of these models, but it lacks the semantic richness of more advanced models including the object-oriented model. The entity-relationship model [Chen] introduces the concept of objects (called "entities" in the model) and relations among them, but does not contain the concept of generalization. Extensions to the entity-relationship model [Teorey] add generalization and other relations. The standard object-oriented model includes a subset of these data structuring concepts but adds the concept of binding operations to data as part of the schema ("methods").

The simplicity of the relational data base model is a drawback in dealing with highly structured problems, because the semantics of a problem cannot be totally represented in flat relations among attributes and must reside in the code. Relational data bases offer a uniform, elegant model which is well understood mathematically, but they have only one level of structure and are not well suited to representing complex objects. Relational tables themselves are not organized into any superstructure, unlike classes in an object-oriented model. A relational table contains only attribute values from a predefined set of primitive types, such as integer, real, character string (often fixed length), and date; abstract data types are not generally supported. There is no way to create a unique object, except by specifying unique values for certain fields, since there is no concept of object identity (the property of an object that distinguishes it from all others, independently of its attributes) [Khoshafian]. In practice, the data base implementor must often generate arbitrary IDs for objects and include ID fields in relational tables to ensure uniqueness. If IDs are not generated, then relations among objects must be represented by relational tables in which each object is represented by a set of unique attributes. Taken with the lack of abstract data types as field values, this implies that the program must manipulate composite keys as if they were single entities. Recursive data structures, such as trees, require the introduction of arbitrary object IDs and are difficult to manipulate in relational data bases because of the absence of transitive closure operations. Relational data bases are essentially flat, in all respects. While this makes them attractive to implement, it makes them weak in representing complex data structures. Considerable effort has been devoted to normalization of relations to eliminate redundancy. Most, if not all, of the problems of normalization are caused by the insistence that all data be represented solely by attribute values and the failure to admit the identity of objects. Relational data bases have been used extensively for commercial applications, in which information has traditionally been viewed as flat collections of records, but pure relational data bases do not appear adequate for representing highly-structured data models such as found in engineering applications, either conceptually or as efficient direct implementations. We feel that relational data bases may best be viewed as implementation vehicles, the "assembly language" of data modeling. Object-oriented systems can represent complex data structures and some of their semantics directly in the data models. However, standard object-oriented systems must resort to object pointers to represent relations among objects.

The entity-relationship model ([Chen], but more readily available in [Ullman] Chapter 1) is a more powerful model than the relational data base because it supports object identity and relations. It has some common concepts with the object-oriented model. In the entity-relationship model, objects (called "entities") belong to named classes (called "entity sets"). Objects have instance variables (called "attributes") that have pure values (they may not be

object references). Named relations (called "relationships") can exist among two or more classes. Each relation has a defined cardinality in terms of its constituent classes. For example, a binary relation associates an object from one class with a single object or a set of objects from another class, according to whether the relation is many-to-many, many-to-one, or one-to-one. A relation, considered as a single object, comprises a set of tuples containing objects drawn from the constituent classes. Classes and relations are of equal weight in this model. The major difference between the relational data base and the entity-relationship model is that objects in the entity-relationship model have identity, apart from their attributes, and relations among objects are specified directly, rather than indirectly in terms of attributes. Chen has proposed a notation for entity-relationship diagrams which clearly shows the structure of a particular schema visually. The entity-relationship model lacks a generalization hierarchy of classes, something that the object-oriented model provides.

Other data base researchers have extended the entity-relationship model by adding the concept of generalization and other structuring mechanisms (see [Loomis] and [Teorey] for a full exploration of various semantic data modeling methodologies). The semantic data models found in the data base literature seem to address data structuring only, without considering the application operations that will be applied to the data. Most of the developers of semantic data modeling techniques seem to view them as high-level notations for developing data base designs, which will eventually be implemented using conventional relational data bases. Object-oriented programming, on the other hand, includes the concept of binding methods to the class schema, which serves as the organizational structure for both data and code.

In the object-oriented community, a lot of attention has been devoted to issues of the class hierarchy: degrees of information hiding of instance variables [Snyder], various kinds of inheritance [Meyer, Lieberman], variations on instantiation [McAllester] and value propagation [Pascoe], and so on. It is possible to design elegant class hierarchies, but there is no way to represent relations among objects from different classes without describing the actual implementation of the relations, usually as instance variables pointing from one object to another.

We have proposed a synthesis of semantic data modeling and object-oriented programming which combines complementary concepts without significant drawbacks. Although the semantic data modeling techniques are widely known, their importance to object-oriented programming does not seem to have been widely recognized. We argue that relations should be given equal semantic weight in an object-oriented model with the class-subclass hierarchy, and that object-oriented languages should support relations directly with syntax and implementations. The model proposed here combines the class hierarchy and methods of the object-oriented model with relations and the other kinds of relations from the extended entity-relationship model. We call this model the *object-relation* model.