Implementing Set Operators over Class Hierarchies

John L. Pfaltz

IPC-TR-88-004
August 5, 1988

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903

This research was supported in part by JPL Contract #957721 and by DOE Grant #DE-FG05-88ER25063.

1. Introduction

In this paper we introduce the concept of an *entity database model*. This model is not intended to be a complete working database model in the sense that various semantic database models, (e.g. DAPLEX [Shi81], SDM, Semantic Data Model [HaM81], Galileo [ACO85], IFO [AbH87]) or object oriented database models, (e.g. Smalltalk [CoM84,MSe86], Orient84/K [IsT86], and [SSE87]) are. These latter models are "complete" in the sense that entire computational languages with syntax and semantics have, or could be, implemented with respect to them. Instead, we will characterize our generic entity database model by only two fundamental properties which are common to all of them. Then we will derive several necessary consequences arising from the implementation of set operators in any such language belonging to the entity model and thereby contrast such languages with the many languages adhering to the relational database model, (e.g. [CGY81,RoS87,Sho79,SWK76]). We hasten to point out that our entity database model is quite distinct from the entity-relationship model introduced by Peter Chen [Che76] and extended by several others (e.g. [CaS84,TYF86]). Their's can be viewed as a *design* model which is extremely useful for organizing the structure of real world data prior to representing it with respect to some particular language, which is often relational. Our entity database model is strictly an *implementational* model.

The key presupposition of the *entity database model* is that the database is implemented by creating uniquely identifiable entities, or objects. These entities may be transient or persistent. These representational entities may, or may not, correspond directly with what we would choose to regard as "entities" in the real world situation we are modeling.

It is customary to impose a type structure on entities, or objects, by assigning them to a *class*. All entities in the class, for example the class PERSON, will share common properties, such as *name, home_address, age,* and *social_security_number*. Most semantic and object-oriented databases support the concept of *sub-class* by means of an *IS_A* construct. Brachman

[Bra83] correctly notes that *inheritance* as defined by the *IS_A* construct is really little more than convenient syntactic shorthand for incrementally creating sub-classes, so we will ignore actual inheritance mechanisms *per se*. The important feature we abstract in the entity database model is the existence of classes and sub-classes, which can be declared by what ever syntactic formalism is convenient. For example, the class DOCTOR, with the additional properties *speciality, training,* and *office_address,* might be a sub_class of PERSON. Of course, a single class may be a super-class subsuming several sub-classes. The class PERSON may contain the sub-class PATIENT, with properties *case_history, complaint,* and *outstanding_amount_due,* as well.

In these preceding examples, classes and sub-classes have been characterized by properties which in relational terminology might be called *attributes*. For the purposes of this paper, we will assume that a class property is a singled valued function, f, of a single variable x where x denotes the unique identifier of an entity instance within the class. The image of f may be a value (e.g. printable string or icon) or some other entity. Permitting f to be set-valued as in DAPLEX [Shi81] will not alter the generality of our approach. While we speak of functions in this paper, those more accustomed to an object-oriented approach might choose to regard the f as analogous to the methods associated with a class.

Classes are commonly defined in terms of their associated attribute properties; but they can also be defined by imposing predicate restrictions on class membership. For example, we might choose to declare the class JUVENILE, with restriction $age < 21$ and with property *grade_in_school,* to be a sub-class of PERSON.

2. Formal Notation

The preceding intuitive introduction to classes, sub-classes, and a class hierarchy in the entity database model can be made more formal. Let $F_C = \{f_i\}$ denote a set of functions associated with a particular class C. Following the syntax used in ADAMS [PSF88] we will use the expression $x.f_i$ to denote the image of x under f_i. For all $x \in C$ the application expression $x.f_i$ is said to be *valid* for any $f_i \in F_C$ even though its actual value, or image, may as yet be undefined.

By a class restriction, we shall mean an expression E in the predicate calculus with at most one free variable. The expression $V.age < 21$ is an example. (Here we capitalize the logical variable for visual emphasis.) The expression is evaluated by replacing the free variable with an entity instance x to determine if it can belong to the class. For the purposes of this paper we make no further assumptions about the nature of E. The syntax of the implementing language may require them to be quite simplistic, or it may permit, as in ADAMS, fairly complex expressions such as $(\forall W \in surgeons)[\, V.age < W.age \,] \wedge V.complaint = 'mumps'$, so long as there is only one free variable.

Those functions applied to the free variable in E will later be of special importance. We denote this set by

$$F_E = \{ f \mid f \text{ is applied to the free variable of } E \}.$$

A **class** is defined by its set F of associated functions and a restricting expression E. That is, $C = (F_C, E_C)$. Either F_C or E_C may be empty. In the latter case $x.E = true$ vacuously. It will be convenient to assume a meta-linguistic operator *class_of* which given a specific instance x of an entity belonging to a class C returns its class. That is

$$\text{class\_of(x)} = C, \quad \text{for } \forall x \in C.$$

A class $C_i = (F_i, E_i)$ is said to be a **sub-class** of $C_k = (F_k, E_k)$, denoted $C_i \leq C_k$, if

(a) $F_i \supseteq F_k$, and

(b) $E_i \rightarrow E_k$ (that is, E_i logically implies E_k).

Condition (a) seems to be universally accepted in both object-oriented and semantic-network class hierarchies. Several authors (c.f. [Bra83, Tou86]) have found strict implication too strong for many forms of semantic reasoning. They would replace it with a non-monotonic logic that allows exceptions. Nevertheless, a condition similar to (b) would seem to be essential in sub-class definition.

Given this definition, it is easy to show that

Proposition 2.1: ≤ is a partial order on any collection of classes.

Following Hull and King [HuK87], we shall graphically represent sub-class dependence (or *IS_A* inheritance) by

$$\begin{tikzcd} C_k:(F_k, E_k) \\ \\ C_i:(F_i, E_i) \end{tikzcd}$$

Figure 1.

And, of course, a single class may have several non-comparable sub-classes as in,

$$\begin{tikzcd} & C_k:(F_k, E_k) & \\ \\ C_i:(F_i, E_i) & & C_j:(F_j, E_j) \end{tikzcd}$$

Figure 2.

Now we can say that a database implementation belongs to the **entity database model** if the implementation supports

(a) uniquely identifiable entity instances,

(b) a class, sub-class hierarchy, and

(c) the standard set operations—union, intersection, relative complement.

The unique identifiability of any entity instance is an important formal characteristic of the entity database model that immediately distinguishes it from the relational model. However implemented—it could be a literal storage address, a symbolic string, or a functional accessing mechanism—the unique identifier which we denote by lower case letters *x, y* or *z,* is not an attribute of the instance it identifies. In our entity model, two distinct entity instances *x* and *y* belonging to a single set of instances of class C may be functionally identical in all respects, that is we may have $x.f_i = y.f_i$ for all $f_i \in F_C$. This is impossible in the relational model, in which any two tuples belonging to a single relation r with schema F_R must at least differ over some set of key attributes $K \subseteq F_R$. Khoshafian and Copeland give a general discussion of object identity in [KhC86]. Some of the issues encountered in implementing an entity naming paradigm are discussed in [PFW88]. In this paper, the precise mechanism used to identify entities is not at issue.

The concept of set is central to all languages subsumed under the entity database model. One deals with sets of entities. These sets, which we will denote by the uppercase letters X, Y, and Z, are themselves entities and so must belong to some class which is distinct from the class of their constituent entities. This distinction must be quite clear in an implementation language, such as ADAMS [PSF88], which employs a syntax like

$\cdots$ is_a SET of <class_name> elements.

An instance set comprised of entities belonging to the class C can not itself belong to C; it must belong to a different class. We require a class constructor of type SET which defines a new class, denoted by $S[C]$, of sets whose elements belong to the class C. Many type theories introduce class constructors based on Cartesian product (for tuples) [HuK87] or disjoint sum (for variant records) [AtB87]. It is our belief that class extension using a *set* constructor is conceptually clearer.

Frequently $F_{S[C]} = \varnothing$, although it need not be. A set class $S[C]$ may have a variety of attribute functions associated with the set as a whole, distinct from the individual elements. One example might be *time_last_altered*. But $E_{S[C]}$ can not be empty. At the very least, $E_{S[C]}$ must include the conjunct

$$(\forall v \in V)\ [\ \text{class\_of}(v) = C\ ].$$

Here V is the free variable. The expression would be evaluated by replacing V with a specific set instance, so that if X is an instance set in $S[C]$ then $(\forall x \in X)$ [class_of(x) = C].

Proposition 2.2: If $S[C_i] \leq S[C_k]$ then $C_i \leq C_k$.

Proof: $E_{S[C_i]} \rightarrow E_{S[C_k]}$, hence $E'_i \wedge (\forall v \in V)[\text{class\_of}(v) = C_i] \rightarrow E'_k \wedge (\forall v \in V)[\text{class\_of}(v) = C_k]$. Since V can range over all sets of C_i elements, for all $x \in C_i$, $x \in C_k$. □

The converse of proposition 2.2 need not be true in general, so one can not automatically extend the partial order on classes C_i to a partial order on classes $S[C_i]$. However,

Proposition 2.3: If for all $S[C_k]$, $F_{S[C_k]} = \varnothing$ and $E_{S[C_k]} \equiv (\forall v \in V)[\text{class\_of}(v) = C_k]$, then the sub-class dependency $\leq$ on $\{S[C_k]\}$ is isomorphic to the structure $\leq$ on $\{C_k\}$.

Other issues associated with implementing the set operators, union $\cup$, and intersection $\cap$, we reserve for a following section.

3. The Compass of a Set

By the **compass** of a set, denoted **comp**(X), we mean the set of all entities which *could* belong to X; that is, the abstract set of all entities which could be encompassed by X. Most often the sets X are mathematically defined by means of some rule. For example,

comp(F) = $\{ x \mid$ the application $x.f_i$ is valid for $\forall f_i \in F\}$, and

comp(E) = $\{ x \mid x.E = \mathit{true} \}$.

The compass of such sets is invariably an infinite set, and hence of theoretical rather than operational interest. Note that the second expression is identical to a *relational calculus expression* ([Ull82], p.158) which denotes the set of all tuples which could satisfy a retrieval expression E. To render it finite, or safe, x is commonly restricted to an existing finite set.

Proposition 3.1:
If $F_i \supseteq F_k$, then comp(F_i) $\subseteq$ comp(F_k).
If $E_i \rightarrow E_k$, then comp(E_i) $\subseteq$ comp(E_k).

Proof: Let x **comp**(F_i) implying that $x.f$ is valid for all $f \in F_i$. Since $F_k \subseteq F_i$, if $f \in F_k$ then $x.f$ must be valid, so x **comp**(F_k).

Since if $x.E_i$ is true, $x.E_k$ must be true by implication, the second containment follows immediately □

The compass of a class, **comp**(C), is that abstract set of all possible entities that *could* belong to the class. Since any x in **comp**(C) must have all $f_i \in F_C$ associated with it, and must also satisfy any restricting expression E, it is evident that

$$\mathbf{comp}(C) = \{ x \mid \text{class\_of}(x) = C \} = \mathbf{comp}(F_C) \cap \mathbf{comp}(E_C).$$

Recall that C_i was said to be a sub-class of C_k if $F_i \supseteq F_k$ and $E_i \rightarrow E_k$, so we immediately have

Proposition 3.2: If C_i is a sub-class of C_k then comp(C_i) $\subseteq$ comp(C_k).

This accords completely with our intuition. The set of entities which could belong to the class DOCTOR must be contained in the set which could belong to the class PERSON.

As observed above, the compass of a class definition is almost invariably an infinite set. But the compass of an actual set instance must be finite, since it must be finitely represented. We emphasize this observation by

Proposition 3.3: If X is a set instance in $S[C]$, then comp(X) $\subset$ comp(C).

4. Set Operators in a Class Hierarchy

Given specific instance sets, the fundamental set operators union ($\cup$) and intersection ($\cap$) must be supported. One can adopt implementation semantics which require that the elements of the operand sets be of precisely the same type, or class, in order for the operator to be defined. For example, in a strongly typed language, such as Pascal, one can not apply the boolean AND operator to two operands, one of which is boolean and one of which is real. It makes no sense. However, it is also customary to loosen the implementation semantics whenever possible, provided that the type of the result is meaningful. Thus, in Pascal, it is legal to perform arithmetic operations (+, -, *, /) even though the operands are of mixed real and integer type—the result is assumed to be real. (Notice that one could regard the class *integer* as a sub-class of the class *real,* and that **comp**(integer) $\subset$ **comp**(real).)

The concept of a union of a set of "doctors" and a set of "patients", regarded simply as "people" clearly makes sense. Cardelli and Wegner call this *inclusion polymorphism* in [CaW85], an entity of type "doctor" can be regarded as being of the form "person" as well. Similarly, the concept of the intersection of a set of "doctors" and "patients" to denote those persons who are both "doctors" and "patients" also makes polymorphic sense. Let us seek for implementation semantics that will be consistent with the class hierarchy.

4.1. Set Representation

Unfortunately, we can not discuss the semantics of set operators without first considering the way that sets are represented. As stated in section 2, we assume that entities, such as $x \in C$, actually exist in some form as *identifiable* objects. When one forms an *instance set X*, one operationally creates a set (linked list, heap, or file) of identifiers of (or references to) its constituent elements.

In the relational database model a tuple is uniquely identified by its attributes, or a subset of its *key* attributes. A relation is a physical set of such tuples. Consequently, the "same" tuple can not belong to two sets (relations); and no two tuples of a relation can have identical attributes. In the entity database model, a single entity can belong to many sets; and two distinctly identifiable entities in a set can be indistinguishable in all other respects.

This distinction has profound implications on the interpretation of the basic set operations. In the relational model, the union of two relations is a *new* relation into which the operand tuples are literally copied. (Although most implementations do so, the tuples need not be physically copied at the time of execution (c.f. [RoK86]), but the operational semantics must behave as if they were.) In our entity data model, only identifiers need to be copied, that is we replicate the reference to, not the entity instance itself. In our discussion below, we will refer to both "copy" semantics and "reference" semantics.

4.2. Set Union

Suppose that the operation X ∪ Y is well defined. To what class should the elements of the resultant, $X \cup Y$, belong? With a slight abuse of our notation we let $C_{X \cup Y}$ denote this class.

Let X be an instance set in $S[C_1]$ and let Y be a set in $S[C_2]$. Readily, if $C_1 = C_2$ then $C_{X \cup Y} = C_1 = C_2$, and if $C_1 \leq C_2$ then $C_{X \cup Y} = C_2$. For the most general case assume that C_1 and C_2 are non-comparable sub-classes of C_0 ($C_1 \leq C_0$ and $C_2 \leq C_0$) which is the most restrictive super-class of them both; C_0 is the least upper bound (l.u.b.) in the partial order $\leq$. That is, we have a case such as shown in Figure 2 of section 2.

There are at least three different possible implementation semantics to be considered.

Option 1: $F_{X \cup Y} = F_1 \cup F_2$, $E_{X \cup y} = \{$ any definition $\}$.

Suppose the implementation literally copies (using the "copy" semantics of the relational database model) the representation of any element x from its original operand set into the resultant set $X \cup Y$. Then readily, we must have $F_{X \cup Y} = F_1 \cup F_2$. And if $x \in X$, for its copy $x' \in X \cup Y$, $x'.f_i = x.f_i$ if $f_i \in F_1$ and $x'.f_i = null$ otherwise. Similarly, if $x \in Y$ then $x'.f_j = x.f_j$, if $f_j \in F_2$ with $f_j = null$ otherwise.

There is a significant operational drawback to employing these "copy" semantics. Using a traditional record implementation, one either obtains very long records most of whose fields are simply *null*, or one employs *variant records* with some kind of tag to indicate which fields, or functions, are represented. For this reason, the relational model does not allow *union* over relations with different schema.

If the implementation employs "reference" semantics, and only copies entity references, or identifiers, into the set representation $X \cup Y$, no serious implementation problems arise. If $x \in X \cup Y$ came from the operand set X and belongs to class C_1, then $x.f_1$ will be defined if $f_1 \in C_1$ and will be *null* otherwise. Here we employ a kind of variant reference as opposed to a variant entity representation.

There is, however, an important theoretical anomaly. Since $F_{X \cup Y} = F_1 \cup F_2 \supseteq F_1$, **comp**($X \cup Y$) = **comp**($F_1 \cup F_2$) $\subseteq$ **comp**(F_1) $\subseteq$ **comp**(X). But this contradicts our understanding of the union operator for which we surely expect **comp**(X) $\subseteq$ **comp**($X \cup Y$).

Option 2a: $F_{X \cup Y} = F_1 \cap F_2, \;\; E_{X \cup Y} = (E_1 \vee E_2)$.

Using either "copy" or "reference" semantics, only attribute properties which are common to both operand classes are defined for the elements of the result. No additional *null* fields, or pointers, are introduced besides those that existed in the original operands.

The additional boolean restriction $E_{X \cup Y} = (E_1 \vee E_2)$ is clearly reasonable, since for any element $x' \in X \cup Y$ it must be the copy of $x \in X$ whence $x'.E_1 = true$, or the copy of $x \in Y$ whence $x'.E_2 = true$. Moreover, since $F_1 \cap F_2 \subseteq F_1$ implies that **comp**(F_1) $\subseteq$ **comp**($F_{X \cup Y}$ and $E_1 \rightarrow E_{X \cup Y}$ implies that **comp**(E_1) $\subseteq$ **comp**($E_{X \cup Y}$, we have that **comp**(X) $\subseteq$ **comp**($X \cup Y$) which agrees completely with our understanding of the union operation.

There is but one serious problem with this semantic interpretation of the union class. $E_{X \cup Y} = E_1 \vee E_2$ need not be well defined on $F_1 \cap F_2$. That is, $F_{E_1 \vee E_2}$ may not be contained in $F_1 \cap F_2$. Note that only functions applied to the free variable can cause trouble. This leads to the following variation.

Option 2b: $F_{X \cup Y} = F_1 \cap F_2, \;\; E_{X \cup Y} = E'$.

This refinement of option 2a assumes that it is possible to find a maximally restrictive expression E' that is defined over $F_1 \cap F_2$ with the property that $E_1 \rightarrow (E_1 \vee E_2) \rightarrow E' \rightarrow E_0$ and $E_2 \rightarrow (E_1 \vee E_2) \rightarrow E' \rightarrow E_0$. This would yield a structure such as

$$\begin{tikzcd}
 & C_0{:}(F_0, E_0) & \\
 & C_{X \cup Y}{:}(F_1 \cap F_2, E') & \\
C_1{:}(F_1, E_1) & & C_2{:}(F_2, E_2)
\end{tikzcd}$$

Figure 3.

Unfortunately such a "decomposition" is computationally complex, and probably in the general case is NP-complete.

Option 3: $F_{X \cup Y} = F_0,\ E_{X \cup Y} = E_0$.

This latter option seems to be the interpretation of choice by several existing entity based systems. If "reference" semantics are employed, an element x is treated as if it had only the attributes and restrictions of C_0 when it is regarded as an element of $X \cup Y$. Using "copy" semantics, only those attributes of C_0 would be replicated.

The implementational semantics associated with the latter option are easily implemented, and they are logically consistent. Since $F_1 \cap F_2 \subseteq F_1$ and $E_1 \rightarrow E_0$, we have **comp**(X) $\subseteq$ **comp**($X \cup Y$), and similarly **comp**(Y) $\subseteq$ **comp**($X \cup Y$). For these reason we will assert that

Proposition 4.4: If $X \in S[C_1]$ and $Y \in S[C_2]$ where C_0 = l.u.b.(C_1, C_2) in $\leq$, then $X \cup Y \in S[C_{X \cup Y}]$, where $F_{X \cup Y} = F_0$, and $E_{X \cup Y} = E_0$, that is $X \cup Y \in S[C_0]$. If l.u.b.(C_1, C_2) does not exist in $\leq$, the union operator is not defined.

4.3. Set Intersection

Following the reasoning we employed above regarding the *union* operator, there is only one reasonable choice for the class of resultant elements in the case of the *intersection* operator.

Option: $F_{X \cap Y} = F_1 \cup F_2,\ E_{X \cup Y} = (E_1 \wedge E_2)$.

Applying the containment relations to the compass concept we obtain **comp**($X \cap Y$) $\subseteq$ **comp**(X) and **comp**($X \cap Y$) $\subseteq$ **comp**(Y) as we intuitively expect. Moreover, since any element $x \in X \cap Y$ must denote an entity that is in both the instance sets X and Y, it must necessarily have all $f_i \in F_1$ and all $f_k \in F_2$ defined on it. And it must satisfy both restrictions E_1 and E_2.

Thus, we obtain the following proposition and class structure

Proposition 4.5: If $X \in S[C_1]$ and $Y \in S[C_2]$ then $X \cap Y \in S[C_{X \cap Y}]$, where $F_{X \cap Y} = F_1 \cup F_2$ and $E_{X \cap Y} = E_1 \wedge E_2$.

$$\begin{tikzcd}
 & C_0{:}(F_0, E_0) & \\
C_1{:}(F_1, E_1) & & C_2{:}(F_2, E_2) \\
 & C_{X \cap Y}{:}(F_1 \cup F_2, E_1 \wedge E_2) &
\end{tikzcd}$$

Figure 4.

Note that given two classes C_1 and C_2, their intersection class always exists.

The problem here is that the *dynamic* creation of elements belonging to $X \cap Y$, or to its class $C_{X \cap Y}$, makes no sense in an entity database model! Suppose $X \cap Y \neq \emptyset$, and $x \in X \cap Y$ is an entity. Since all $f_k \in F_2$ are defined on x, x can not denote an entity in X which has only those attributes $f_i \in F_1$ defined (unless $C_2 \leq C_1$, in which case $C_{X \cap Y} = C_2$), thus logically contradicting its membership in $X \cap Y$.

To be meaningful in an entity database model, one must first *statically* declare the *intersection class* of C_1 and C_2. Elements, z, of this previously declared class can now be created. Later, if an instance set X of elements of class C_1 were formed, these elements z could be included in the set. They are also entities of class C_1 by inheritance. Similarly, they could be included in an instance set Y of C_2 entities. Now an intersection operator could determine which entity elements z were common to both instance sets X and Y and create a set identifier $X \cap Y$ denoting the set. In an entity database model, an intersection operator does not *create* the elements of an intersection set; it only creates a set representation that *denotes* those entities which had previously constituted such a set. In retrospect, that is all that the union operator does as well.

Note that the relational model with its "copy" semantics has no such problem implementing an intersection operator whose members have attributes $F_1 \cup F_2$. It is called the *natural join*. This observation emphasizes another important distinction between the "copy" semantics of the relational model and the "reference" semantics of an entity model. The former captures the familiar sense of algebraic assignment operations. Given a Pascal statement, such as

z := x + y;

the values of *x* and *y* are used to create a *new* value, which is then assigned to a location denoted by *z*. Given relational statements, such as

$$\text{RESULT} := \text{X} \times \text{Y}$$

$$\text{RESULT} := \sigma_{f_i = v_i}(\text{X})$$

a *new* relation, denoted by RESULT, with *new* tuples is actually created. With the "reference" semantics of an entity model, we would only create a new identifier "RESULT" to denote some subset of an existing collection of entities. The denotational aspect of the entity database model can be emphasized by observing that in this model, it is perfectly meaningful to have a literal on the left side of an assignment statement, viz.

"all_people" := X **union** Y

"sick_doctors := X **intersect** Y

where X and Y denote existing sets of entities of class DOCTOR and PATIENT respectively with "all_people" and "sick_doctors" being the literal (and possibly persistent) names of their union and intersection sets.

Intersection classes can be declared at the same time that any two sub-classes C_1 and C_2 of a super_class C_0 are defined, thereby allowing the creation of entity instances in the intersection. But the process is an open ended one. Suppose that we create a third sub-class C_3 of C_0; it might be JUVENILE. Then to support all possible intersections we would need the class structure as shown in Figure 5.

$$C_0{:}(F_0, E_0)$$

$$C_1{:}(F_1, E_1) \qquad C_2{:}(F_2, E_2) \qquad C_3{:}(F_3, E_3)$$

$$C_{1,2}{:}(F_1 \cup F_2, E_1 \wedge E_2) \qquad C_{1,3}{:}(F_1 \cup F_3, E_1 \wedge E_3) \qquad C_{2,3}{:}(F_2 \cup F_3, E_2 \wedge E_3)$$

$$C_{1,2,3}{:}\, F_1 \cup F_2 \cup F_3, E_1 \wedge E_2 \wedge E_3)$$

Figure 5.

Since these intersection classes are independent of individual instance sets, we will drop our earlier notation, $C_{X \cap Y}$, in favor of $C_{i,j}$, as shown.

There is no logical reason to resist the construction of such a lattice of class declarations; but to manually declare it is surely tedious. An implementation of an entity database model can do this automatically whenever two or more sub-classes are declared. The necessary information is available; and the automatic construction of $C_{i,k}{:}(F_i \cup F_k, E_i \wedge E_k)$ is quite straightforward. In ADAMS, to prevent a proliferation of unnecessary class definitions, we defer actual creation of intersection classes until at least one entity will become a member of the class, as in

x **belongs to** <class_name_1> **AND** <class_name_2>.

Note that the class hierarchy automatically becomes a lattice of data types as described in [Sco76], or more particularly a semi-lattice structure as described in [Ada85], since we do not assume that the least upper bound of two classes need exist.

5. Specialization and Generalization

The preceding discussion of set operators in a class hierarchy, which inexorably led to the formulation of a class semi-lattice, assumed the existence of an initial set of classes C_1, C_2, ... C_k which were sub-classes (or refinements) of a super class C_0. We never addressed the issue of how they might be formed. In a sense it is irrelevant to the major thesis of this paper. But it is nevertheless an interesting issue.

Abiteboul and Hull, in [AbH87], consider the twin concepts of *specialization* and *generalization* as methods of defining hierarchical class dependencies. Specialization, in which an existing super-class C_0 is refined to form a new more specialized sub-class C_1, is the mechanism most commonly employed in semantic database languages. It is also the method that we tacitly assumed in section 2 of this paper. But as Arbiteboul and Hull note, one *ought* to be able to create the initial sub-class, super-class dependencies through generalization as well. Given two classes C_1 and C_2, we should be able to create a super-class C_0 that subsumes them both as in Figure 2. The example they give begins with MOTOR_BOAT and CAR as classes corresponding to C_1 and C_2 which generalize to the class VEHICLE. Unfortunately, the implementation semantics associated with generalization is not straightforward.

First, we must assume that there is some time delay between the declaration of $C_1:(F_1, E_1)$, $C_2:(F_2, E_2)$ and their generalization C_0, else they could have been formed by specialization. Also we assume that C_0 should subsume all common attributes of its sub-classes and only those entities which belong to at least one of its sub-classes. Consequently, we seek to declare $C_0:(F_1 \cap F_2, E_1 \vee E_2)$

Because of the time delay, it is quite possible that attributes which we would naturally want to regard as common will have been given distinct attribute names. To create a reasonable implementation of the generalization process, one must be prepared to address the potentially sticky issue of *synonyms,* as discussed in [PFW88]. (The issue of synonyms can also arise in specialization, but by the nature of the process itself it is far less likely.)

If, at a still later time, we wish to include a third class C_3, perhaps TRUCK or AIRPLANE to use Abiteboul and Hull's example, then the existing definition of C_0 will have to change dynamically. And if $F_1 \cap F_2 \cap F_3 \subset F_1 \cap F_2$ then it likely that any representations of C_1 and

C_2 which employ inheritance mechanisms will have to dynamically change as well.

The preceding two paragraphs only describe implementation complications associated with generalization that are not encountered with specialization. Both can be overcome. The most serious objection is the same that we encountered when attempting to use option 2 to declare the class $C_{X \cup Y}$. If we want C_0 to represent only instances belonging to its sub-classes then $E_1 \vee E_2$ should be testable, but it need not be evaluable over $F_1 \cap F_2$.

6. Summary

If we seek to implement an entity database model which allows the creation of a class hierarchy through successive refinement, or specialization, as most object-oriented and semantic database models do, and if we expect the implementation to support standard set operations, then certain requirements follow naturally.

1) In addition to base entity classes C_1, C_2, ... C_k there must be corresponding set classes $S[C_1]$, $S[C_2]$, ... $S[C_k]$ characterizing those manipulable entities which are *sets* of entities. Although not discussed in the body of the paper, one must also be able to characterize *sets of sets* by a class of the form $S[S[C_k]]$. A partition of a set S of C_k elements (i.e. $S \in S[C_k]$) would be an entity in $S[S[C_k]]$.

If in the implementation language one strictly restricts set operations to operands belonging to the same set class, then no more is required. But such a restriction is overly stringent. Assuming a more general polymorphic treatment of intersection and union operators as discussed above, we have the following conclusions.

2) In order to obtain closure for the intersection operator, given two non-comparable classes C_i and C_k, one must be able to declare their intersection class $C_{i,k}$ and, of course, the associated set class $S[C_{i,k}]$ to which the resultant set will belong.

It is more difficult to ensure closure for the union operator. A reasonable implementation approach is to restrict unions to sets of elements whose classes C_i and C_k have a least upper bound.

3) The implementation system must maintain a semi-lattice of class declarations $C_1, C_2, \ldots C_n$, whose lower bound (or **inf**) is the intersection class $C_{1,2,\cdots,n}$: $(F_1 \cup F_2 \cup \cdots \cup F_n, E_1 \wedge E_2 \wedge \cdots \wedge E_n)$. A similar semi-lattice of set-class declarations, and possibly of set-set-class declarations must also be maintained.

4) In view of 2) above, a limited form of "multiple inheritance" induced by the formation of intersection classes is required. But it need not be of the generality described in [Car84].

We believe that these represent minimal characteristics which must be present in any entity database model whose class hierarchy is defined by specialization. More general models, such as one which would support generalization, or equivalently the complete closure of the union operator, are possible. These require an effective way of defining the **sup** operator in the class semi-lattice(s), thereby making them true lattices. We note the the greatest element in such a lattice would be a class C_0:$(F_1 \cap F_2 \cap \cdots \cap F_n, E_1 \vee E_2 \vee \cdots \vee E_n)$, where in all probability the first intersection $F_1 \cap \cdots \cap F_n = \emptyset$. Therefore membership of an entity instance x in C_0 would effectively assert only that x is an entity. Such general implementation models may be of only theoretical interest.

7. References

[AbH87] S. Abiteboul and R. Hull, IFO: A Formal Semantic Database Model, *Trans. Database Systems 12*,4 (Dec. 1987), 525-565.

[Ada85] T. Adachi, Powerposets, *Inf. and Control 66*(1985), 138-162.

[ACO85] A. Albano, L. Cardelli and R. Orsini, Galileo: A Strongly Typed Interactive Conceptual Lanugage, *Trans. Database Systems 10*,2 (June 1985), 230-260.

[AtB87] M. P. Atkinson and O. P. Buneman, Types and Persistence in Database Programming Languages, *Computing Surveys 19*,2 (June 1987), 105-190.

[Bra83] R. J. Brachman, What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks, *COMPUTER 16*,10 (Oct. 1983), 30-36.

[Car84] L. Cardelli, A Semantics of Multiple Inheritance, in *Semantics of Data Types, Lecture Notes in Computer Science 173*, Springer Verlag, June 1984, 51-67.

[CaW85] L. Cardelli and P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys 17*,4 (1985), 471-522.

[CaS84] M. A. Casanova and J. E. A. Sa, Mapping Uninterpreted Schemes into Entity-Relationship Diagrams: Two Applications to Conceptual Schema Design, *IBM Jour. Res & Dev. 28*,1 (Jan. 1984), 82-94.

[CGY81] D. D. Chamberlin, A. M. Gilbert and R. A. Yost, A History of System R and SQL/Data System, *Proc. 7th VLDB Conf.*, Cannes, France, Sep. 1981, 456-464.

[Che76] P. P. Chen, The Entity-Relationship Model---Toward a Unified View of Data, *Trans. Database Systems 1*,1 (Mar. 1976), 9-36.

[CoM84] G. Copeland and D. Maier, Making Smalltalk a Database System, *Proc. SIGMOD Conf.*, Boston, June 1984, 316-325.

[HaM81] M. Hammer and D. McLeod, Database Description with SDM: A Semantic Database Model, *Trans. Database Systems 6*,3 (Sep. 1981), 351-386.

[HuK87] R. Hull and R. King, Semantic Database Modeling: Survey, Applications, and Research Issues, *Computing Surveys 19*,3 (Sep. 1987), 201-260.

[IsT86] Y. Ishiwkawa and M. Tokoro, A Concurrent Object-Oriented Knowledge Representation Language, Orient84/K: Its Features and Implementation, *OOPSLA '86, Conf. Proc.*, Sep. 1986, 232-241.

[KhC86] S. N. Khoshafian and G. P. Copeland, Object Identity, *OOPSLA '86, Conf. Proc.*, Sep. 1986, 406-416.

[MSe86] D. Maier, J. Stein and et.al., Development of an Object-Oriented DBMS, *OOPSLA '86, Conf. Proc.*, Sep. 1986, 472-482.

[PFW88] J. L. Pfaltz, J. C. French and J. L. Whitlatch, Scoping Persistent Name Spaces in ADAMS, IPC TR-88-003, Institute for Parallel Computation, Univ. of Virginia, June 1988.

[PSF88] J. L. Pfaltz, S. H. Son and J. C. French, The ADAMS Interface Language, *Proc. 3th Conf. on Hypercube Concurrent Computers and Applications*, Pasadena, CA, Jan. 1988, 1382-1389.

[RoK86] N. Roussopoulos and H. Kang, Principles and Techniques in the Design of ADMS, *IEEE Computer 19*,12 (Dec. 1986), 19-25.

[RoS87] L. A. Rowe and M. R. Stonebraker, The POSTGRES Data Model, *Proc. 13th VLDB Conf.*, Brighton, England, Sep. 1987, 83-96.

[Sco76] D. S. Scott, Data Types as Lattices, *Siam J. on Computing 5*,3 (Sep. 1976), 522-587.

[SSE87] A. Sernadas, C. Sernadas and H. Ehrich, Object-Oriented Specification of Databases: An Algebraic Approach, *Proc. 13th VLDB Conf.*, Brighton, England, Sep. 1987, 107-116.

[Shi81] D. W. Shipman, The Functional Data Model and the Data Language DAPLEX, *Trans. Database Systems 6*,1 (Mar. 1981), 140-173.

[Sho79] J. E. Shopiro, Theseus--A Programming Language for Relational Databases, *Trans. Database Systems 4*,4 (Dec. 1979), 493-517.

[SWK76] M. Stonebraker, E. Wong, P. Kreps and G. Held, The Design and Implementation of INGRES, *Trans. Database Systems 1*,3 (Sep. 1976), 189-222.

[TYF86] T. J. Teorey, D. Yang and J. P. Fry, A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model, *Computing Surveys 18*,2 (June 1986), 197-222.

[Tou86] D. S. Touretzky, *The Mathematics of Inheritance Systems*, Morgan Kaufmann Publ., Los Altos, CA, 1986.

[Ull82] J. D. Ullman, *Principles of Database Systems, 2nd Ed.*, Computer Science Press, Rockville, MD, 1982.

Table of Contents