The MR Diagram - A Model for Conceptual Database Design

Raghu Ramakrishnan
Avi Silberschatz

Department of Computer Science
The University of Texas
Austin, TX 78712.

ABSTRACT

Traditional database models are not sufficiently expressive for a variety of standard and non-standard database applications. Several models supporting greater abstraction have been proposed to fill this gap, but no one model has gained wide acceptance. This paper defines a model, based on the notion of *molecules* and *non-first normal form relations*, that provides a powerful abstraction mechanism using aggregation. The model also provides a simple pictorial representation that allows a compact and clear specification of a database. We illustrate the model with several examples and show how it can be used in the design of databases.

1. Introduction

It has long been recognised that database modelling and knowledge representation share the objective of representing some abstraction of the real world. While there are important differences, both stand to gain much from a synthesis, and as a first step towards such a synthesis, data models must support greater abstraction (1). Several applications such as Computer Aided Design would also benefit from such abstraction capabilities. This has sparked a growing trend towards the development of data models (2, 3, 4, 5, 6, 7) that support abstraction but no one model has gained wide acceptance.

Semantic data models allow us to model abstractions by providing a mechanism for defining new abstractions in terms of other, already defined, abstractions, using aggregation and generalization. Informally, if A is an aggregation of B and C, an instance of A is a composition of an instance of B and an instance of C. If A is a generalization of B and C, an instance of A is either an instance of B or an instance of C. A more powerful notion of aggregation allows us to view an instance of A as a composition of a set of instances of B and a set of instances of C. Classification allows us to group all instances of A together.

In the Relational model (9), aggregation is supported by associating a sequence of attributes with a relation name, and classification groups all tuples of a given relation. Generalization is not supported directly. Smith and Smith (10, 11) extend the notion of aggregation by allowing an attribute to be a tuple from another relation, and Batory and Buchmann (4) take this a step further by allowing an attribute to be a set of tuples from another relation. Our contribution is to provide a uniform representation for abstraction using an extension of the Relational model which supports generalization and aggregation in the form described by (4). Our scheme has the advantages of a simple pictorial representation and an extended form of the relational query language.

The paper is organised as follows. Section 2 introduces the notion of a molecule, in particular the way aggregation is used to define the structure of an abstract object. Section 3 discusses how we attach meaning to this structure so that our representation is a natural model of the abstraction being modelled. In section 4 we present our model and illustrate it with several examples. Subsequent sections discuss operators and procedural extensions to make the model more expressive, to achieve a clean separation between levels of abstraction, and to clarify the definition of structure and semantics. We then demonstrate how a database can be designed in a top-down fashion. There is a brief discussion of bottom-up design and its relation to view modelling and schema integration. Finally, we consider related work and present our conclusions.

† This research was supported in part by the Office of Naval Research under contract N00014-80-K-0987, and by the National Science Foundation under grant MCS81-04017.

2. The Concept of a Molecule

Our model rests upon the notion of a *molecule type* which describes a type of object or a type of relationship between objects at some level of abstraction. A molecule type is defined as a non-first normal form relation. Each tuple in this relation represents a molecule of this type, and describes an object or a relationship.

An 'object' is some abstraction that is of interest to us. It could be a VLSI chip layout, a corporation, or a cat. Like the word 'entity', it is intuitively defined. It is important to note that our model does not distinguish between entities and the relationships among them. Rather, an object and a relationship between objects are both viewed as abstractions, and it is entirely possible that a relationship at one level is an object at a higher level. The model provides a uniform representation for all abstractions, and they may be viewed as objects or relationships, whichever is more appropriate, in different contexts

A molecule type M is defined in terms of other molecule types. Intuitively, a molecule of type M is a collection of molecules of the component types. There is a set-valued attribute corresponding to each component molecule type, and in a tuple describing a molecule of type M, these sets contain the keys of all component molecules of the corresponding type. The definition of a molecule type may also include 'atomic' or non-set valued attributes which, in a given molecule, are used to describe the molecule as a whole.

Pictorially, a molecule type is represented by an ellipse. The ellipse for molecule type M is labelled 'M' and has the list of M's atomic attributes listed beside it in parentheses.

If M is defined in terms of molecule types M_i, an arrow is drawn from the ellipse for M to the ellipse for each M_i. Thus, outgoing arrows indicate the structure of a molecule type and may be labelled with the corresponding component names. Incoming arrows at the ellipse representing molecule type M indicate the molecule types which are defined using M as a component type.

We present a few examples to clarify these concepts before defining them formally.

Example 1: Supplier-Parts-Projects

Figure 1 describes the familiar Suppliers-Parts-Projects relationship. We describe this using a molecule type called 'Orders'. Figure 1.a describes 'Orders' pictorially, Figure 1.b shows instances of the underlying relations for Suppliers, Parts and Projects, and Figure 1.c shows an instance of the relation for 'Orders'.

The key for this molecule is order_#. An 'Orders' tuple could be interpreted as follows: the set of suppliers jointly supply the set of parts to the set of projects. There are other possible interpretations, and in general, the intended interpretation is not clear from the above definition of a molecule. ||

Example 2: University Departments

Figure 2.a partially describes a department in a university. A department is described by its faculty, students and the courses it offers. At this level of abstraction, it is not known how courses are described. Figure 2.b shows instances of the relations for 'Faculty', 'Students' and 'Courses'. To completely describe the molecule type 'Dept' one need only know the key attributes of the molecule type 'Courses'. Figure 2.b describes an instance of the relation for 'Courses' partially and Figure 2.c shows an instance of the relation for 'Dept' under the assumption that courses have unique names.

This example illustrates two points. One is that a molecule type can be defined using other molecule types. The second is that we only need to know the keys of the underlying molecules to define a molecule type. This implies that the relation describing a molecule will not change even if the relations describing some of its underlying molecule types change (ie the structures of some components change) so long as their keys continue to be valid. ||

We need to be more precise about the nature of a molecule type. We therefore formalize this intuition. A molecule type could represent a type of object or a relationship between objects. In either case, it is defined in terms of other molecule types. If molecule type M is defined in terms of molecule types $M_1, M_2, \ldots, M_n$, then a molecule of type M is a collection of molecules of types M_1 through M_n. Each molecule type has an associated sequence of atomic (indivisible, non-set-valued) attributes whose values, for a given molecule, describe the properties of that molecule taken as a whole.

Mathematically, a molecule type M is defined as follows:

$$M = [A_1, A_2, \ldots, A_m, M_1, M_2, \ldots, M_n]$$

where the A_i's are atomic attributes, the M_i's are molecule types, and m,n $>=$ 0, and not m=n=0. The A_i's are called the atomic or scalar components of M and the M_i's are called the molecular components.

This represents a non-first normal form relation with tuples

$$[a_1, a_2, \ldots, a_m, \{k_1\}, \{k_2\}, \ldots, \{k_n\}]$$

where the keys in a given set $\{k_i\}$ are distinct.

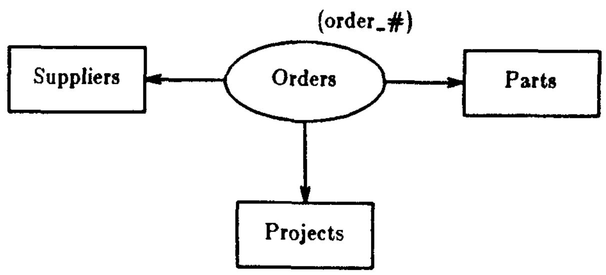

Figure 1.a. MR diagram for Suppliers-Parts-Projects

S_id
s1
s2
s3

Suppliers

P_id
p1
p2
p3
p4

Parts

Pr_id
pr1
pr2
pr3

Projects

Figure 1.b. Relations for Suppliers, Parts and Projects

Order_#	Suppliers	Parts	Projects
1	{s1}	{p1, p3, p4}	{pr1}
2	{s2}	{p1}	{pr2, pr3}

Figure 1.c. Relation for the Orders molecule type

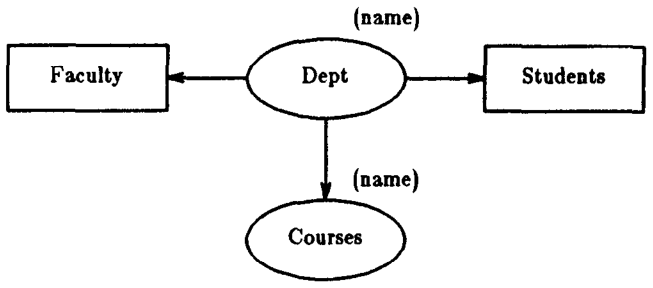

Figure 2.a. The Dept molecule type

f_name
Smith
Jones
Brown
Black

Faculty

s_name
Tom
Ken
Sue
Nick

Students

c_#	...
CS304	...
CHY301	...
CHY104	...

Courses

Figure 2.b. Relations for Faculty, Students and Courses

d_name	faculty	students	courses
CS	{Smith, Brown}	{Tom, Ken, Nick}	{CS304}
CHY	{Jones, Black}	{Sue, Nick}	{CHY301, CHY104}

Figure 2.c. Relation for the Dept molecule type

A molecule of type M is described by a tuple in this relation, and is composed of a collection of tuples from the relations that describe molecule types M_i, i = 1 to n. These underlying tuples are specified by sets of keys $\{k_i\}$, some of which may be null. The a's are attributes describing molecule M and are atomic. Note that some of the M_i's may be identical. This means that objects of these types play more than one role in the relation (or abstract object) defined by molecule type M. For example, consider the following molecule definition:

People = [name]

Father_of = [People, People]

People is a molecule type with the atomic attribute name and no set-valued attributes. Father_of is a molecule type with no atomic attributes and two set-valued attributes.

Thus, the tuple [{John}, {Jim, Susan, Joe}] in the relation for Father_of means that John, Jim, Susan and Joe are people and that John is the father of Jim, Susan and Joe.

We also define a navigation operator as follows:

Given a molecule m (of type M, say), $m\uparrow i$ refers to the set $\{k_i\}$ in the tuple defining m, and $m\uparrow i.k$ refers to the molecule of type M_i with the key k (ie, the tuple with key k in the relation defining M_i). We used $m\uparrow i$ instead of $m\uparrow M_i$ to avoid the ambiguity that arises when some of the M_is are identical. To make this more readable, we add some syntactic sugar:

Molecule type M is defined by $[A_1, .., A_m, C_1, .. , C_n]$ and $[M_1, .. , M_n]$ where the C_i's are distinct component names. The second part of the definition specifies the underlying molecule type of component C_i. The molecule type Father_of may now be defined by [Father, Children] and [People, People].

How do we describe the traditional concepts of entity and relation in terms of molecules? An entity is simply a molecule type with no underlying molecule types. The sets $\{k_i\}$ are absent from the definition of a molecule representing an entity. A traditional first-normal form relation between entities can also be represented as a molecule with no underlying molecules. We call such molecules *independent molecules*. Thus, as in the relational model, we have a uniform representation for objects and relations between objects. An entity is just a special case of an object, and a traditional relation is just a special case of a relation between objects.

Example 3: Courses in a Department

Figure 3 shows a refined version of the 'Dept' molecule type. The point to note is that a given object (in this example, the independent molecule types 'Faculty'
and 'Students') can be used to describe more than one molecule type. []

Example 4: Projects

This example shows how a molecule type may be defined recursively in terms of itself. A project is described by its name, which is unique, the faculty members and students who are working on it, and a set of sub-projects (Figure 4.a).

Figure 4.b shows an instance of the relation for the 'Project" molecule types, using the 'Faculty' and 'Student' relations defined in Figure 2.b. This could also be used to associate a list of related projects with the definition of a project. In fact, the semantics of the 'Project" molecule type is ambiguous in that the molecule type definition does not tell us which of the above two roles a component project plays. We expect, however, that the role played by a component of a molecule type is the same in all molecules of this type. []

3. The Semantics Associated With a Molecule

The above definition of a molecule type specifies only its structure. How do we interpret this structure? Let us examine various examples to clarify this point. Reconsider the Suppliers-Parts-Projects example. The meaning of the 'Orders'
molecule [1, {s1}, {p1, p2, p4}, {pr1}] is clear. Supplier s1 supplies parts p1, p2 and p4 to project pr1. This does not mean that p1, p2 and p4 are the only parts he supplies to pr1. If we wish to enforce the semantics that these are the only parts he supplies to pr1, we could do so by making 'Suppliers' a key for 'Orders'.

Now consider the 'Orders' molecule [2, {s2,s3}, {p1,p6}, {pr2}]. Does this mean that both s1 and s2 supply the named parts or that they supply them jointly? The first interpretation means that this molecule is just a compact representation for the two molecules [2, {s2}, {p1,p6}, {pr2}] and [2, {s3}, {p1,p6}, {pr2}]. The second treats {s2,s3} as a composite supplier, a single indivisible unit. So the semantics depend on the nature of the aggregation represented by {s2,s3}.

We thus distinguish between two kinds of components, unit components and complex components. In the above example, viewing Suppliers as a complex component leads to the first interpretation and viewing it as a unit

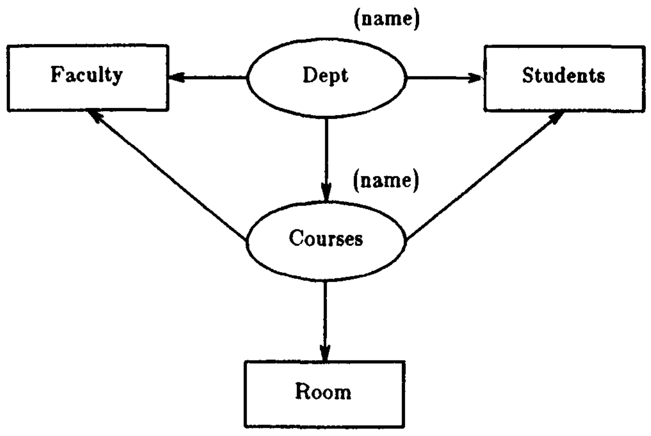

Figure 3. The Dept molecule type refined

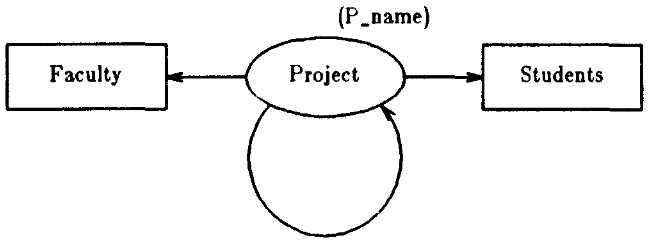

Figure 4.a. The Project molecule type

P_name	Faculty	Students	Sub_projects
CAD	{Smith, Jones}	{Tom, Nick}	{Chemical_design Computer_analysis}
Chemical_design,	{Jones, Black}	{Sue, Nick}	{NULL}
Computer_analysis	{Brown}	{Tom, Ken}	{NULL}

Figure 4.b. Relation for the Project molecule type

component leads to the second. As we saw in the example above, a molecule with complex components is really a set of molecules. If molecule type M has complex components $M_i \ldots M_j$, then a molecule of type M with n_k keys in the set corresponding to complex component M_k really represents a collection of $n_i * \ldots * n_j$ molecules, each of molecule type M' where the definition of molecule type M' is identical to the definition of molecule type M, except that all the complex components are now unit components (In fact, the set corresponding to one of these components in a molecule of type M' is singleton.). An instance of M is just a compact representation for an instance of M' where each molecule of M stands for $n_i * \ldots * n_j$ molecules of M', and it is really the molecules of type M' that describe objects of interest to us. However, defining a molecule type with more than one complex component obscures the nature of the abstraction that is of interest to us, and we strongly discourage it.

4. The Proposed Model

A database definition is a collection of molecule type definitions and a database instance is a set of instances of the underlying molecule types. Database design is the process of defining these molecule types.

For example, a database may contain information about all legal firms in a city. The design of this database is essentially the definition of the molecule type 'Law_Firm'. This can be done hierarchically, using a top-down approach. This molecule type may also be thought of as a view of the lower-level molecule types in this database, and the process of database design may be thought of as the process of building this view from a set of independent molecules. These two approaches are discussed separately in later sections.

The central concept in our model of a database is the molecule. The definition of a molecule includes our notion of aggregation. A molecule is an aggregate object, described by a collection of molecules from a predefined set of molecule types, and may contain an arbitrary number of molecules from any one type. Classification is implicit in that molecules of a given type are represented by tuples in a relation.

If molecule type M is a generalization of molecule types M_i, we define M as if the M_i's were its component types. However, we know that a given molecule of type M is a single molecule of one of the types M_i, and so in a tuple representing a molecule of type M, all sets $\{k_j\}$, except one, will be empty and the exception will be a singleton set.

If we know that $\{k_i\}$ is always either singleton or null, we represent component C_i by a broken arrow. Thus if M is a generalization of molecule types M_i, the arrows to all these components are broken. Further, only one of these sets is non-null. We represent this pictorially by drawing an arc through these arrows. In general, an arc through a set of arrows, broken or solid, indicates that only one of the components associated with these arrows is non-null in any given molecule of this type.

Example 5: Modelling Vehicles

Figure 5.a models the molecule type 'Vehicle' as a generalization of the molecule types 'Car' and 'truck. All vehicles have a license number. The
broken arrows indicate that a molecule of type 'Vehicle' has at most one component molecule of type 'Truck' and one of type 'Car'. The arc through these arrows indicates that at most one of these sets is non-null, in other words, a vehicle is a car or a truck but not both!

Figure 5.b models 'Vehicle' as a molecule type with the underlying molecules 'Car' and 'Truck'. This does not reflect the above semantics. For instance, this definition allows a 'Vehicle' to be a collection of cars and trucks, an abstraction that we would normally think of as a collection of vehicles rather than as a single vehicle.

Figure 5.c models 'Vehicle' as being either a collection of cars or a collection of trucks, again not the intended semantics. []

Example 6: Modelling Versions of Cars

In Figure 6 we illustrate how similar objects can be classed together to reflect the fact that they are identical at some level of abstraction.
A molecule of type 'Car' is one of the named cars. Thus, while detailed descriptions of these cars may be stored at some level, we are able to represent the fact that they are instances of the same abstract object and store those properties that are relevant to this abstracted view. []

Example 7: Modelling Variant Structures

This example illustrates how to define several versions of a molecule or to define a molecule whose structure has variants.

In Figure 7, addresses have two components, a city name and a local address. The local address is either a post-box address or a street address. This example shows how an arc through some of the arrows can be used to realize a variant structure. Since the arrows for the molecule types 'Po_box' and 'Street_address' are linked by an arc, a given address molecule has at most one of these. The component 'City_name' is always present since no arc pass through the arrow linking it to the address molecule type.

This technique can also be used to generalize some

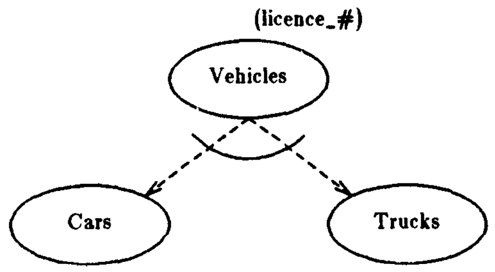

Figure 5.a. Vehicles defined using generalization

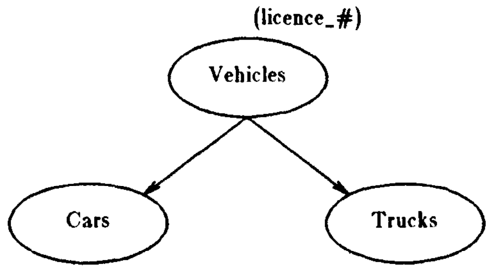

Figure 5.b. Vehicles as a collection of cars and trucks

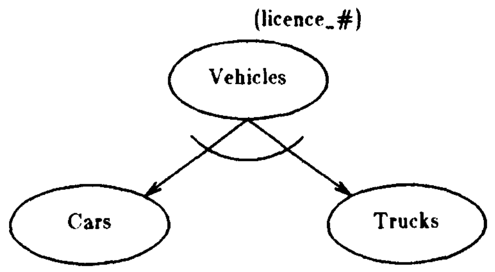

Figure 5.c. Vehicles as a collection of either cars or trucks

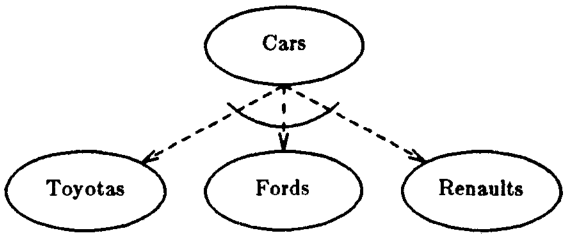

Figure 6. Different versions of cars

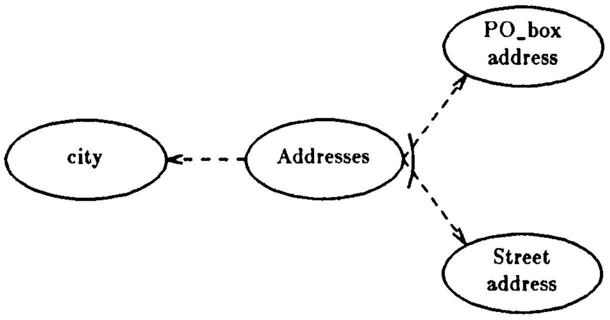

Figure 7. A molecule type with variant structure

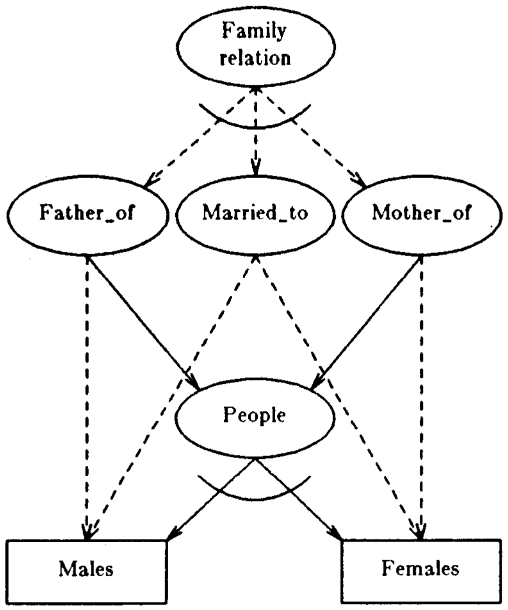

Figure 8. Generalization of relationships

objects (the arrows pointing to these are linked by an arc). []

Example 8: Generalizing Relationships

The previous examples showed how molecule types representing structured entities could be generalized. This example shows that the concept of generalization extends naturally to relations.

Figure 8 models molecule types describing the father, mother, daughter, son and married_to relationships. An instance of the 'Family_relation' molecule is a molecule from one of the above types, reflecting the fact that any of these relationships is a family relationship. []

5. Operations on Molecules

One of the strengths of the Relational model is its powerful query language. In this section, we show how the relational operators can be extended to non-first normal form relations as we use them, thus providing our model with an elegant and natural query language.

When a relational operator is applied to a molecule type M that has complex components, the relation for this molecule type is first expanded to the relation for its underlying molecule type M'. Thus we only need to define the extension of these operators to molecule types with no complex components, which are represented by non-first normal form relations with each set valued component viewed as a single indivisible unit.

The extension of the union, cartesian product and projection operators is straightforward. Extending the selection, intersection and join operators involves testing set equality and containment. A set $\{k_j\}$ is a subset of set $\{k_i\}$ if every element in $\{k_i\}$ is also in $\{k_j\}$. Two sets are equal if each is a subset of the other.

We can also use the navigational operator, defined earlier, in specifying queries but its use within a relational operator is expensive.

The operations discussed so far are retrieval operations. How do we insert, update and delete molecules? We insert a molecule of type M by inserting a tuple for it in the relation for M. When we do this, the tuples corresponding to the keys in sets $\{k_i\}$ should already be in the relations corresponding to to molecule types M_i. We delete a molecule by removing the tuple that describes it. Thus a delete is local to the relation for the given molecule type and does not propagate to the relations for the underlying molecule types. This has two important effects. First, suppose we delete a tuple t1 whose key is part of some set $\{k_i\}$ in another tuple t2. If we now retrieve the components of t2 using the navigation operator, we find that there is no tuple with the key for t1. We assume that this key refers to a null tuple. Thus, a delete at one level may update the structure of a tuple at a higher level, and this change is noticed only when we attempt to retrieve the components of the latter. This may not always reflect the intended semantics of a molecule, and care must be exercised in this regard in the definition of a molecule type. Second, the components of a deleted molecule are not removed. Again, this may not reflect the intended semantics of a molecule.

This provides the motivation for *owned molecule types*. A molecule type may 'own' one of its component molecule types. This owned molecule type will not appear as a component anywhere else. It is introduced solely to help define the owner. Let molecule type M_1 own molecule type M_2. Molecules of type M_2 are inserted, deleted or updated only when a molecule of type M_1 (which owns them) is inserted, deleted or updated. Any changes in a molecule of type M_1 are immediately reflected in the molecules it owns.

There is one subtle problem associated with owned molecule types. Each key must represent a unique molecule, so there cannot be two identical molecules. Now how do we handle the case when two molecules of the owner type wish to own the same molecule? If we let the molecule be shared, deleting it from one owner will cause it to be removed which means it is deleted from the other owner too. To avoid this, we associate some extra fields with the definition of an owned molecule type. These could be either system-generated unique keys, such as tuple id numbers, or the key fields of the owner molecule type. These fields serve as the key for the owned molecule type.

We represent an owned molecule type by a bold ellipse. Note that an owned molecule type may be defined in terms of molecule types that are not owned, either by it or its owner, and that 'owns' is transitive. We will see examples of owned molecule types when we discuss top-down design.

We update a molecule by changing the tuple that represents it. Details of how this is done should be obvious from the above discussion since an update may be thought of as a delete followed by an insert.

Although the motivation for introducing owned molecules was to make the model more expressive, they also permit more efficient implementations since all owned molecules can be stored with their owner molecules.

We may wish to provide a user with a limited view of a molecule type. This is made possible by associating a set of views with each molecule. A view is a subset of the components that define the molecule type. A view may be thought of as another molecule type, but no inserts, deletes or updates on views are allowed. Views are intended to be a window on the parent molecule type, and tuples in a view are changed only when the corresponding tuples (molecules) in the parent molecule type are changed.

6. Procedural Extensions to the Model

Abstract data types have proved to be a powerful and clean abstraction mechanism in programming languages. We now consider how abstract data types can be realized in our model by associating a set of procedures with each
molecule type. These procedures represent the only operations permitted on molecules of this type.

The intention is that a molecule type, as defined by its relation schema and associated procedures, should completely represent an abstract object type at the desired level of abstraction, in analogy with abstract data types in programming languages. The relation for this molecule type is a collection of abstract objects, each tuple being one such object.

Typically, we would include procedures 'insert', 'delete', 'update' and perhaps a sophisticated retrieval operation, reflecting the semantics of the molecule in a natural way. The 'insert', 'delete' and 'update' procedures reflect the intended semantics of the molecule. All these procedures have the corresponding system operations as defaults, and their definition could be omitted if we chose to do so.

Consider a molecule type M_1 defined with molecule type M_2 as a component. The procedures associated with M_1 may only manipulate molecules of type M_2 using the set of procedures (or defaults, if some of them are not defined) associated with M_2. The definition of these procedures may be done bottom-up or top-down. In the top-down approach we assume the existence of procedures which implement the intended semantics at lower levels, and define each of these later on in hierarchical fashion.

The navigational operator may be used in these procedures to locate appropriate components but not to navigate within them. These procedures now provide a clean separation between various levels of abstraction. The implementation of a molecule type in terms of components using aggregation, classification or generalization is captured by these procedures and is opaque to higher levels so long as these procedures retain the same interfaces.

These procedures thus allow us to define the semantics of a molecule type procedurally, complementing the declarative semantics described in section 3. The declarative semantics are reflected by the system-provided default procedures for 'insert', 'delete', etc.

These procedures may be used to enforce semantic constraints, by having the insert and update procedures perform various checks.

They also allow us to define fields which are functionally determined by other fields. These fields are computed by the insert procedure, automatically insuring that any dependencies are satisfied. There is one problem however. If a component which is not owned is specified as one of the determining fields, any changes in this component must be propagated upwards. Thus insert, delete and update procedures at one level could depend on the specified functional dependencies at a higher level. Clearly, this problem does not arise if all determining components are owned. These functionally determined fields provide a powerful tool for defining meaningful views (see section 5) of a molecule type while maintaining as much information hiding as desired.

7. Top - Down Database Design

The design of a database begins with the identification of the abstract objects we wish to describe. We refine each of these objects in steps, introducing more detail at each stage until the description of each object
in the database has been taken down to the lowest desired level of abstraction.

At this stage, we will probably discover, among other things, that components of some molecule types appear as part of some of their other components and can be migrated down. We may also find that several references to molecule types can be replaced by references to some of their components or views, sometimes even prompting us to redefine some molecule types.

Next, we identify owned molecule types. A molecule type can be made the owner of a component type if the latter is not a component of any other molecule type and, further, we do not wish to talk about it as an abstract object in its own right. By now, the design should also be sufficiently advanced to enable us to make a choice of keys.

Next, we bring the database into 3'NF by moving down some molecule types. A non 1NF relation is in 3'NF if it is in 3NF when each of its components is viewed as a single atomic unit. We consider molecule types M with complex components in terms of their underlying molecule types M'. Violations of 3'NF are caused by dependencies in which non-key components determine other components. If non-key components C_1 and C_2 determine component C_3 in molecule M_1, we create a new molecule type M_2 with components C_1, C_2

and C_3 and key (C_1, C_2). M_1 is redefined with components C_1, C_2 and C_3 replaced with a single component of type M_2. This approach to database design is similar to that discussed in (10).

Example 9: Design of a Database for Vehicle Sales

We illustrate the top-down approach by designing a database that describes vehicle sales. We begin with the abstractions we wish to model - manufacturers, dealers and buyers - and refine each in turn. Figure 9.1.a shows the database at this stage. Note that we do not know precisely how they are related as yet.

A manufacturer is described by the vehicles he has produced and sold in the current year. We associate attributes 'name' and 'year' with this molecule type and refine it in terms of components that describe what this manufacturer produced and sold in the given year (Figure 9.1.b).

The products are vehicles, and so a manufacturer's output can be described in terms of the number of vehicles of each type that he produces. The 'Products' molecule type is used to associate the number produced with the vehicle type. A collection of these molecules can thus express a manufacturer's output.

A manufacturer's sales can be described by the number of vehicles of each type that he sells to each of his dealers. We use a molecule type 'Inventory' to express the number of vehicles of a given type that are sold to a given dealer. This is shown in Figure 9.1.c. It is easy to see that 'Products' and 'Inventory' are identical since they have the same structure and describe the same abstraction - the number of vehicles of a given type that participate in some transaction or are part of some structure. So we can use just one molecule type, say 'Inventory', for this purpose (Figure 9.1.d). However, this may not be the best possible design since using two molecule types allows us to make each of them an owned molecule.

We now refine 'Dealer'. A dealer is described by the number of vehicles of each type that he has in stock and his sales for the given year (Figure 9.2). The description of the vehicles he has in stock is similar to the 'Inventory' molecule type. The only difference is that an extra attribute has been added to show the price charged by the dealer. His sales are described by the sale date, vehicle and customer.

To complete this design we need to describe the molecules 'Vehicle' and 'Customer'. A vehicle is either a car or a truck (Figure 9.3) and each of these is characterised by a name and a description. Cars and trucks may have different descriptions. We don't expand the molecule types representing these descriptions, in order to keep this example brief.

A customer may be a person or a corporation, but is, for the purposes of this abstraction (his or its role in a sale) completely characterised by his or its address. This insight may be used to guide the design of the molecule types representing them. We illustrate this by partially expanding 'People' (Figure 9.4.a). The information that is viewed as an abstract object in the 'd.Sales' molecule type is made a distinct component. Figure 9.4.c shows the molecule types for 'd.Sales' and 'People' at this stage. However this has a subtle flaw - the semantics of the 'People' molecule type makes it desirable to make the 'Name and address' component an owned type. This makes it impossible for us to use this as a component in 'd.Sales', so we really need to use another molecule type with the same structure to define 'd.Sales'. Thus it turns out that the insight gained with regard to the structure of the 'd.Sales' molecule type is inapplicable, but this may not always be so.

Examining our design so far, it is easy to see that all vehicles have a name, so the name attributes for 'Car' and 'Truck' can be moved up.

At this stage we can identify the owned molecule types using the criteria discussed earlier. This is a fairly obvious exercise and we show the resultant design at this stage (Figure 9.5).

If we assume that the social security number is a key for 'People', then our design is not in 3'NF since name and address is also a key and hence determines the third component, 'Occupations'. This can be rectified by moving 'Occupations' down to the 'Name and address' molecule type. This requires 'Occupations' to be owned, which means we have to store details about engineers for each engineer. The solution to this problem is to make 'Occupation name' an owned molecule, and store details of each occupation in a separate molecule type. The final design for 'People' is shown in Figure 9.4.b.

There is one last point that is instructive. Each molecule type may be used to define part of a user's view. Given this, we may wish to restrict the amount of detail visible from a given molecule type. For example, the 'Manufacturer' type need only contain the dealer's name. Details of his business are none of the manufacturer's business. We could enforce this by making 'Dealer_name' an independent molecule (owned by 'Sales' and NOT 'Dealer') and replacing 'Dealer' in the definition of 'Sales' with 'Dealer_name'. []

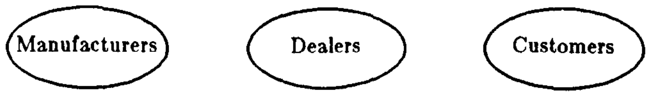

Figure 9.1.a. The initial database description.

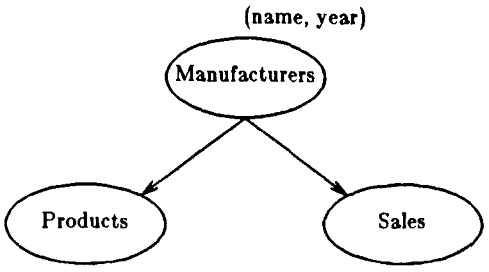

Figure 9.1.b. Manufacturers

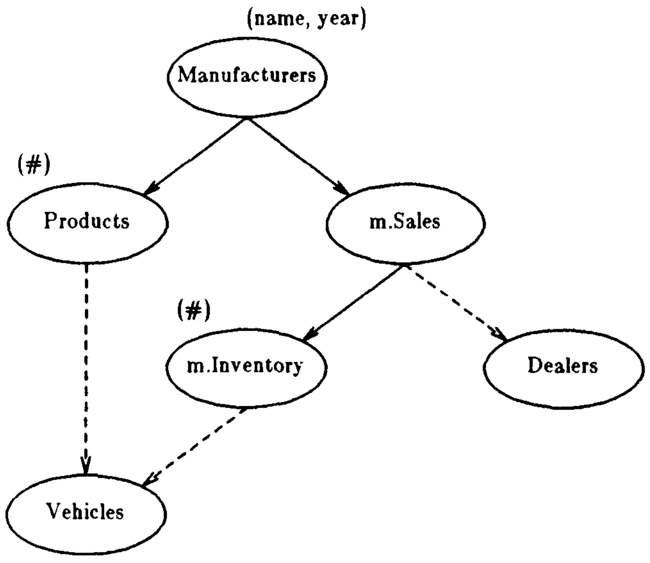

Figure 9.1.c. The Manufacturers molecule type refined

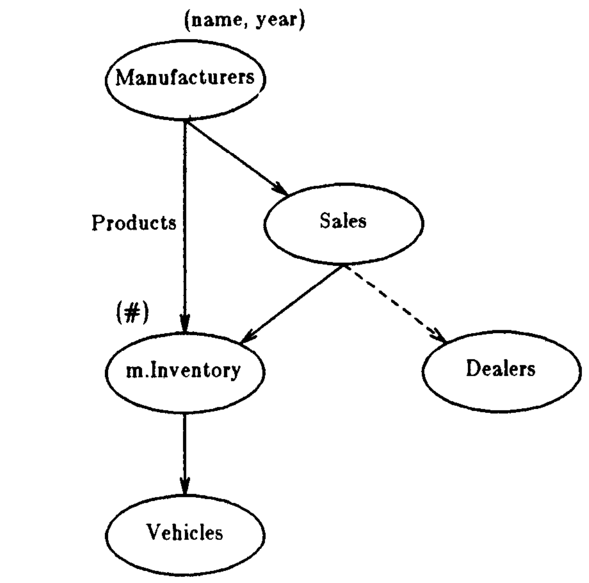

Figure 9.1.d. Another way of refining Manufacturers

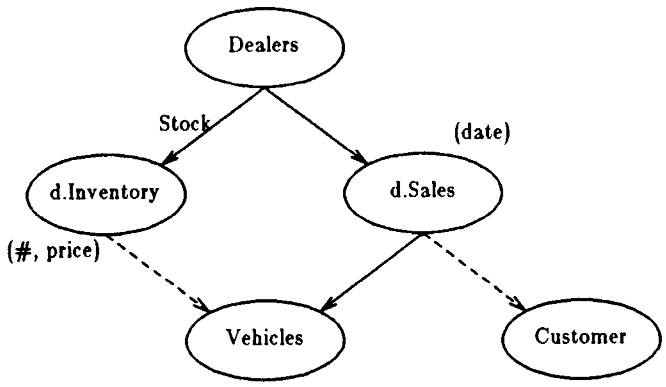

Figure 9.2. Dealers refined

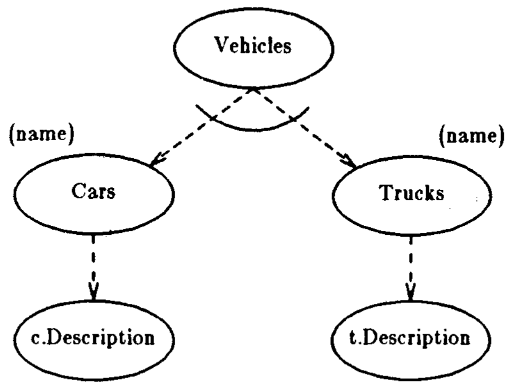

Figure 9.3. Vehicles refined

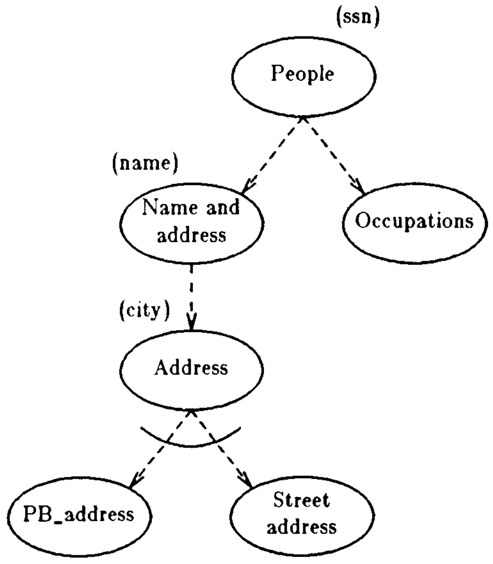

Figure 9.4.a. The People molecule type refined

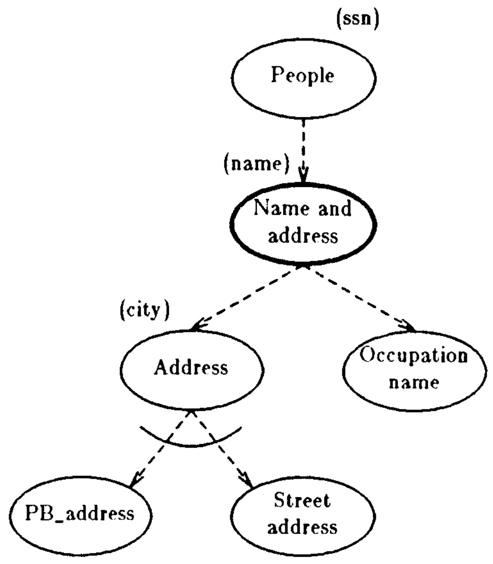

Figure 9.4.b. The final design for People

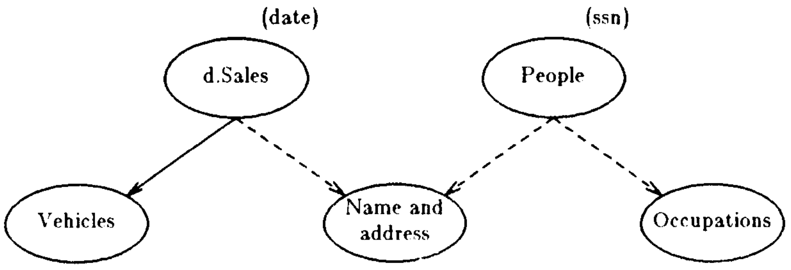

Figure 9.4.c. A Possible Refinement of d.Sales

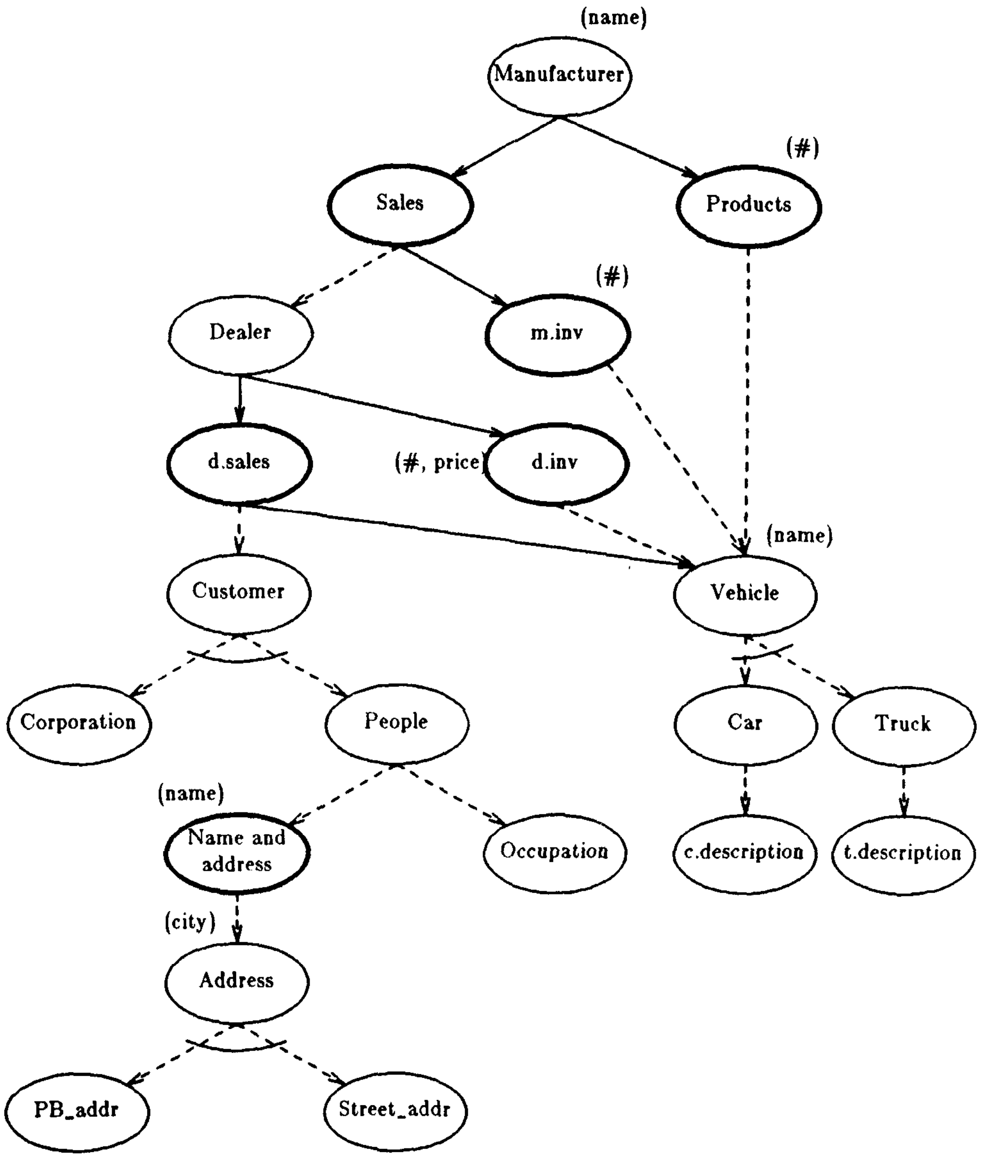

Figure 9.5. The vehicle sales database

8. Bottom - Up Database Design

The top-down approach is probably the best way to design a database if we are designing it from scratch. However, if several objects at the lower levels of abstraction have already been designed, as for instance when we wish to integrate some given schemas or build some abstractions on top of them, a bottom-up approach may be desirable.

In bottom-up design, we may think of the database as a set of underlying objects - independent molecules, at the lowest level - with a set of user views on top. The design process consists of identifying these objects and then constructing the user views. When we apply this technique to schema integration, there is the further problem of ensuring that the given schemas are consistent with each other. This is a difficult problem which we do not discuss. It is possible that attempting to build the desired user views bottom-up, and restructuring conflicting schemas top-down will prove fruitful, but this is an area where a great deal of work needs to be done before we can come up with good answers.

We give a simple example of how views can be built on top of a set of given schemas. Consider the database in example 8. We may wish to define an abstract molecule type to represent a family. The 'Family' molecule type could be defined with 'Family_relation' as a component, and attributes such as the family address. (We could also define it in several other ways, with components 'Father_of', and 'Married_to', for instance.)

9. Relationship to Other Models

Our definition of a molecule is equivalent to that given in (4). The important difference between the two is that we establish the correspondence between a molecule of type M and its underlying molecules of type M_i in the relation defining M. Batory and Buchmann establish this correspondence by associating with each molecule of type M_i the keys of all molecules of type M that it helps to describe. They do this in order to achieve a separation between the abstract specification of a molecule type and its actual implementation, in keeping with the programming language paradigm of abstract data types. This makes it difficult to define a molecule precisely and to refer to its components. Our approach avoids these problems, and it also supports the abstraction paradigm, as we demonstrated in sections 5 and 7.

The use of non-first normal forms in VERSO (2, 3) is similar to our approach, but it differs in the following. Suppose we say that a molecule type M_1 is 'used' in the definition of molecule type M_2 if it is a component type of M_2 or 'used' in the definition of one of M_2's component types. Then, in the VERSO approach, a molecule type M cannot be defined with component types M_1 and M_2 if there is some M\d\s-13\s+1\u such that it is used in the definition of both M_1 and M_2. Essentially, this restriction forces all their MR diagrams to be trees in which the sub-trees at each level are disjoint.

For example, the database design in Figure 9.1.c is illegal in VERSO because 'Manufacturer' is defined in terms of 'Products' and 'Sales' and both of these are defined using 'Vehicle'.

All their 'molecules', to use our terminology, are owned molecules and are stored with their owner molecules.

Also, they view the atomic attributes 'a' as being drawn from underlying attribute types, which may be shared between two molecule types. Thus the 'a's differ from the components $\{k_i\}$ only in that they are atomic. In our model, they are intended to describe the molecule as a whole, and are distinct from the components $\{k_i\}$ in that they are independent of other molecule types.

Smith and Smith aggregation (10, 11) also imposes a tree structure on the aggregates but without the restriction that the sub-trees be disjoint. However, their notion of aggregation does not allow us to group several tuples of a single component type. In terms of our definition of a molecule type, the sets $\{k_i\}$ are always singleton or null. In fact, the Smith and Smith model may be viewed as a generalization of the Relational model, where an attribute in a relation may be a tuple from another relation. Our model goes one step further and allows an attribute to be a set of tuples from an underlying relation.

Clearly, our model subsumes the Relational and ER models. It may be appropriate to stress the differences in the pictorial representations of molecule types and the ER model. MR diagrams use ellipses to represent molecule types with molecular components. Such molecule types have no counterpart in ER diagrams. Rectangles are used to represent independent molecules, which may be either entities or relationships in the ER model. In other words, the MR diagram represents just the structure of the molecules. Their semantics must be defined elsewhere, although an appropriate choice of molecule names often helps. Arrows here define structure, not relationships. So although MR and ER diagrams may look similar, they are interpreted in totally different ways.

10. Conclusions

The MR model provides a natural tool for database modelling. Objects in this model have a uniform and simple description, and can be expressed clearly in diagrams. The model provides a high level of abstraction and supports both top-down and bottom-up design of a database. The MR model is built around the concept of a molecule. Defining a molecule in terms of non-first normal form relations provides a precise representation as well as the basis for a relational query language. Further, it allows us to express generalization of objects and relationships between objects in a natural way. Our approach is clearly object-oriented, and, with the procedural extensions discussed in section 6, completely captures the programming language concept of abstract data types.

The fact that bottom-up design is similar to building a set of views suggests that this might be an appropriate model for view modelling and view integration. This, however, is beyond the scope of this paper.

References

[1] H.K.T. Wong and J. Mylopoulos, "Two Views of Data Semantics: A Survey of Data Models in Artificial Intelligence and Database Management", INFOR, October 1977.

[2] F. Bancilhon et al., "VERSO: A Relational Back End Data Base Machine", Proc. Inter. Workshop on Database Machines, San Diego, 1982.

[3] S. Abiteboul and N. Bidoit, "Non First Normal Form Relations to Represent Hierarchically Organized Data", Proc. ACM Symposium on Principles of Database Systems, Waterloo, 1984.

[4] D. Batory and A. Buchmann, "Molecular Objects, Abstract Data Types, and Data Models: A Framework", Proc. VLDB, 1984.

[5] M. Hammer and D. McLeod, "Database Description with SDM: A Semantic Database Model", ACM Trans. Database Systems, Vol. 6, No. 3, 1981.

[6] R. Hull and C.K. Yap, "The Format Model: A Theory of Database Organization", JACM, July 1984.

[7] S. Abiteboul and R. Hull, "IFO: A Formal Semantic Database Model", TR-84-304, University of Southern California.

[8] P.P.S. Chen, "The Entity Relationship Model - Towards a Unified View of Data", ACM Trans. Database Systems, Vol. 1, No. 1, 1976.

[9] E.F. Codd, "Relational Model of Data for Large Shared Data Banks", CACM, June 1970.

[10] J.M. Smith and D.C.P. Smith, "Database Abstractions: Aggregation", CACM, June 1977.

[11] J.M. Smith and D.C.P. Smith, "Database Abstractions: Aggregation and Generalization", ACM Trans. Database Systems, Vol. 2, No. 2, 1977.

[12] B.H. Liskov and S.N. Zilles, "Programming with Abstract Data Types", Proc. ACM SIGPLAN Symp. on Very High Level Languages, SIGPLAN Notices (ACM) Vol 9, No 4, 1974.