Database Abstractions: Aggregation and Generalization

JOHN MILES SMITH and DIANE C.P. SMITH
University of Utah

Two kinds of abstraction that are fundamentally important in database design and usage are defined. Aggregation is an abstraction which turns a relationship between objects into an aggregate object. Generalization is an abstraction which turns a class of objects into a generic object. It is suggested that all objects (individual, aggregate, generic) should be given uniform treatment in models of the real world. A new data type, called generic, is developed as a primitive for defining such models. Models defined with this primitive are structured as a set of aggregation hierarchies intersecting with a set of generalization hierarchies. Abstract objects occur at the points of intersection. This high level structure provides a discipline for the organization of relational databases. In particular this discipline allows: (i) an important class of views to be integrated and maintained; (ii) stability of data and programs under certain evolutionary changes; (iii) easier understanding of complex models and more natural query formulation; (iv) a more systematic approach to database design; (v) more optimization to be performed at lower implementation levels. The generic type is formalized by a set of invariant properties. These properties should be satisfied by all relations in a database if abstractions are to be preserved. A triggering mechanism for automatically maintaining these invariants during update operations is proposed. A simple mapping of aggregation/generalization hierarchies onto owner-coupled set structures is given.

Key Words and Phrases: data model, relational database, database design, aggregation, generalization, data abstraction, data type, integrity constraints, knowledge representation
CR Categories: 3.65, 3.69, 3.79, 4.29, 4.33, 4.34

1. INTRODUCTION

An *abstraction* of some system is a model of that system in which certain details are deliberately omitted. The choice of the details to omit is made by considering both the intended *application* of the abstraction and also its *users*. The objective is to allow users to heed details of the system which are *relevant* to the application and to ignore other details.

In some applications a system may have too many relevant details for a single abstraction to be *intellectually manageable*. Such manageability can be provided by decomposing the model into a *hierarchy* of abstractions. A hierarchy allows relevant details to be introduced in a controlled manner. The abstractions on any

This work was partially supported by the National Science Foundation under Grant MCS75-09903.
Authors' address: Computer Science Department, University of Utah, Salt Lake City, UT 84112.

given level of the hierarchy allow many relevant details to be (temporarily) ignored in understanding the abstractions on the next higher level.

One advantage of such an "abstraction hierarchy" is the capability for different users to access the model at different levels of abstraction. For example, if the underlying system is a commercial enterprise, executive users may access global information while clerical users may access detailed information. In this way a single model can be shared among several diverse users without compromising their access requirements. Another advantage of an abstraction hierarchy is an enhanced *stability* of the model as the application, or the system itself, evolves. A change in a detail which is ignored at higher levels of the model will leave these higher levels unaffected. Some changes of course will permeate from a low level to a high level.

A *relation* in Codd's relational schema supports two distinct forms of abstraction. We call these forms of abstraction "aggregation" and "generalization." *Aggregation* refers to an abstraction in which a relationship between objects is regarded as a higher level object. In making such an abstraction, many details of the relationship may be ignored. For example, a certain relationship between a person, a hotel, and a date can be abstracted as the object "reservation." It is possible to think about a "reservation" without bringing to mind all details of the underlying relationship—for example, the number of the room reserved, the name of the reserving agent, or the length of the reservation.

Generalization refers to an abstraction in which a set of similar objects is regarded as a generic object. In making such an abstraction, many individual differences between objects may be ignored. For example, a set of employed persons can be abstracted as the generic object "employee." This abstraction disregards individual differences between employees—for example, the facts that employees have different names, ages, and job functions.

When an appropriate *structuring discipline* is imposed, Codd's relational schema can simultaneously support both hierarchies of aggregation abstractions and hierarchies of generalization abstractions. In a previous paper [4] we proposed a structuring discipline suitable for aggregation abstractions. The present paper develops a structuring discipline for generalization abstractions and integrates it with the one previously proposed for aggregation abstractions.

The benefits of such a structuring discipline are:

(i) abstractions (sometimes called views) pertinent to different database users can be effectively integrated and consistently maintained;

(ii) stability (sometimes called data independence) of models can be provided under several kinds of evolutionary change;

(iii) highly structured models can be supported without a significant loss in intellectual manageability;

(iv) a more systematic approach to database design, particularly of database procedures, can be developed;

(v) more efficient implementations are possible since more assumptions can be made about high level structure.

In [4] we contrasted the primitives for expressing aggregation abstractions that are found in programming languages with those that are needed in databases. We showed how to adapt Hoare's *Cartesian product* structure[1] [2] so that it may be used to define relational models. This adaptation leads to several insights about the role of *aggregation* abstractions in databases. The present work can be regarded as an adaptation of Hoare's *discriminated union* structure[2] [2] to relational models. This adaptation leads to insights about the role of *generalization* abstractions in databases.

Database research has been almost exclusively concerned with aggregation (for example, Codd's normal forms [1]) while generalization has been largely ignored. The reason for this is probably that in simple models generalization can usually be handled, fairly satisfactorily, on an ad hoc basis. Interestingly, artificial intelligence (AI) research on knowledge bases has been principally concerned with generalization (for example, Quillian's semantic networks [3]) while aggregation has not been fully exploited. By combining aggregation and generalization into one structuring discipline, we are cross-fertilizing both the database and AI areas.

Section 2 develops a philosophy for representing generalization abstractions in Codd's relational schema. This philosophy leads to a powerful **generic** structure for defining relational models. This structure is described in Section 3. Section 4 examines how the **generic** structure may be used to handle various modeling situations. Section 5 considers five properties of a relational model which must remain invariant during update operations. These invariant properties form a (partial) axiomatic definition of the **generic** structure. A set of rules for maintaining these invariants is developed by using semantic considerations. Section 6 offers concluding remarks on several aspects of the **generic** structure, including database design methods, implementation techniques, and access languages.

2. GENERALIZATION ABSTRACTIONS

The object of this section is to develop a philosophical position on the nature and representation of generalizations. First we motivate why it is important to represent generalizations in models of (aspects of) the real world. Then we consider hierarchies of generalizations and discuss some of their properties. Finally, we investigate the representation of generalizations as Codd relations.

We will use the term "generalization" in the following way: *A generalization is an abstraction which enables a class of individual objects to be thought of generically as a single named object.* Generalization is perhaps the most important mechanism we have for conceptualizing the real world. It is apparently the basis for natural language acquisition—the child moves from the observation of specific dogs to a model of dogs in general. It allows us to make predictions about the future on the basis of specific events in the past—if this fire and that fire have burned my hand, then perhaps fires in general will burn my hand.

In designing a database to model the real world, it is essential that the database schema have the capability for explicitly representing generalizations. This will allow naming conventions in the model to correspond with natural language and enable users to employ established thought patterns in their interactions with the

[1] Similar to PASCAL's record structure.
[2] Similar to PASCAL's record variant structure.

Table I. Generic objects participating in a relationship

affiliation:

profession	society
computer scientist	ACM
computer scientist	IEEE
doctor	AMA
trucker	Teamsters
electrical engineer	IEEE

database. In particular the explicit naming of generic objects allows the following capabilities: (i) the application of operators to generic objects; (ii) the specification of attributes of generic objects; and (iii) the specification of relationships in which generic objects participate. Essentially we want to allow generic objects to be treated uniformly with, and to have the same capabilities as, all other kinds of objects. The first capability above is clear enough; however, it may be worth giving some examples of the second and third capabilities.

Suppose we have generalized a class of individual objects into a named generic object. Information which "summarizes" attributes of the individuals can be attached as attributes of the generic object. For example, since all dogs have "sharp teeth" and "four legs," this information can be attached as an attribute of the generic object "dog." This information could be attached redundantly as attributes of individual dogs. However, this disguises the fact that dogs in general have these attributes rather than just the individuals mentioned.

As another example, we might generalize a class of truck drivers into the generic object "trucker." We may not be interested in the pay rate of each individual truck driver, but only in the average (maximum, minimum) pay rate of truck drivers in general. This pay rate information belongs as an attribute of the generic object "trucker."

A generic object, like an individual object, can participate in a relationship with other objects. For example, we can take the relationship between "professions" and their affiliated "societies." Particular professions include "trucker," "computer scientist," and "doctor"—all of which are generic objects. This relationship, called "affiliation," is represented in Table I. Notice that "profession" is a generalization of the class which includes "trucker," "computer scientist," and "doctor."

We now examine the properties of a *hierarchy* of generic objects. To fix the discussion, we make this examination via a specific example. We assume that a model must be constructed for the set of vehicles owned by some large organization such as a government agency or an industry. These vehicles include many diverse types such as trucks, submarines, bicycles, and helicopters.

Figure 1 illustrates one particular decomposition of "vehicle" into lower level generic objects. Note that individual vehicles are not explicitly represented. Each generic object should be thought of as defining a class of individual vehicles. The

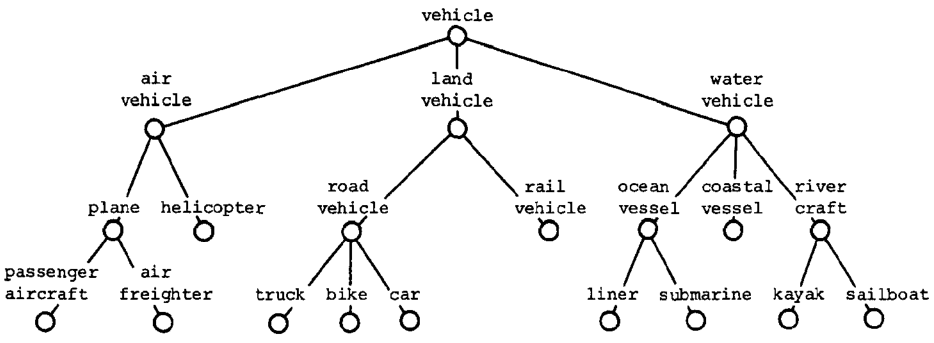

Fig. 1. A generic hierarchy over vehicles

figure indicates, for example, that "truck," "bike," and "car" can be generalized to the notion "road vehicle"; that "road vehicle" and "rail vehicle" can be generalized to the notion "land vehicle"; and that "land vehicle," "air vehicle," and "water vehicle" can be generalized to the notion "vehicle."

Let G be an object in a generic hierarchy. To represent G as a Codd relation, we must select a set of attributes which are common to *all* individuals in the class of G. For example, in representing "vehicle" we may include the attributes "identification number," "manufacturer," "price," and "weight." These attributes are common to all vehicles. In representing "road vehicle" we may include the previous attributes and others such as "number of wheels" and "tire pressure." These attributes are common to all road vehicles; however, the latter two attributes are *not* common to all vehicles. In representing "truck" we may include all the previous attributes and others such as "engine horsepower" and "cab size." These attributes are common to all trucks; however, the latter two attributes are *not* common to all road vehicles.

Now an individual truck will be a member of each of the classes "vehicle," "road vehicle," and "truck." However, the relevant attributes of this truck will vary from class to class. When this truck is considered as an individual *vehicle*, any attributes which distinguish trucks from other *vehicles* will be irrelevant. When this truck is considered as an individual *road vehicle*, any attributes which distinguish trucks from other *road vehicles* will be irrelevant. In general an individual object will have more relevant attributes the lower the generic level of the class in which it appears. We call the attributes of an individual object that are relevant to a class G the *G-attributes* of that object.

The generic hierarchy shown in Figure 1 has two characteristics that do not belong to all generic hierarchies. The first characteristic is that it is a tree (i.e. no generic object is the immediate descendant of two or more generic objects). The second characteristic is that the immediate descendants of any node have classes which are mutually exclusive. Generic hierarchies which do not have these characteristics are shown in Figures 2 and 3, respectively. Our method for representing generic hierarchies as Codd relations can accommodate these general forms of generic hierarchy.

Figure 2 indicates that "helicopter" can be generalized in two ways—either to "motorized vehicle" or to "air vehicle." Figure 3 illustrates a decomposition of

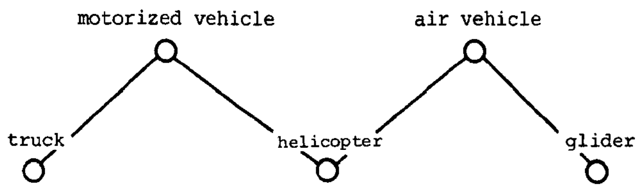

Fig. 2. A generic hierarchy which is not a tree

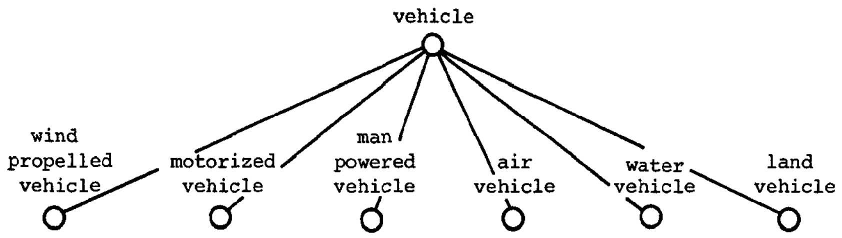

Fig. 3. A generic hierarchy in which the immediate descendants of a node do not form mutually exclusive classes

"vehicle" into two distinct kinds of generic object. One kind of generic object is concerned with the method of vehicle propulsion (wind, human, motor). The other kind of generic object is concerned with the principal medium through/on which the vehicle moves (air, water, land). Some of these descendant objects do not have disjoint classes. For example, some vehicles are both motorized and move through the air. The classes for "motorized vehicle" and "air vehicle" therefore have some common members.

Our method for representing a generic heirarchy requires that the immediate descendants of any node be partitioned into groups. Each group must contain generic objects whose classes are mutually exclusive. In practice this grouping can usually be made quite easily from semantic considerations. For example, the descendants in Figure 3 would be grouped as: {wind propelled vehicle, motorized vehicle, man powered vehicle} and {air vehicle, water vehicle, land vehicle}. The first group contains mutually exclusive classes which correspond to alternative types of "propulsion system." The second group contains mutually exclusive classes which correspond to different types of "transit medium."[3]

We call a mutually exclusive group of generic objects sharing a common parent a *cluster*. We say that a cluster *belongs to* its parent generic object. For example, we may talk about the two clusters belonging to "vehicle" in Figure 3. A leaf node in a generic hierarchy has no cluster belonging to it. In Figure 1 every non-leaf generic object has exactly one cluster belonging to it. In Figure 2 the cluster belonging to "motorized vehicle" and the cluster belonging to "air vehicle" have a common element—helicopter..

We shall find it necessary to give each cluster a (meaningful) name. This name

[3] If amphibious vehicles were of interest, the generic object "amphibious vehicle" would be included as an alternative to "air vehicle," "water vehicle," and "land vehicle."

should be chosen so that it is descriptive of the generic objects in the cluster. For example, the name of the cluster {wind propelled vehicle, motorized vehicle, man powered vehicle} may be "propulsion category." The name of the cluster {air vehicle, water vehicle, land vehicle} may be "medium category."

We will now describe a method for representing a generic hierarchy as a hierarchy of Codd relations. We create one relation for each generic object in the hierarchy. Assume G is a generic object such that (i) I is the class of individual objects associated with G, (ii) $A_1, \ldots, A_n$ are the G-attributes, and (iii) $C_1, \ldots, C_m$ are the names of clusters belonging to G. G is represented by the Codd relation:

G:

A_1	...	A_n	C_1	...	C_m
...	...	...	...	...	...
v_1	...	v_n	v_{n+1}	...	v_{n+m}
...	...	...	...	...	...

where (i) there is one and only one tuple for each individual in I; (ii) if an individual has a value v_i for attribute A_i, then its tuple contains v_i in domain A_i; (iii) if an individual is also included in generic object v_{n+j} in cluster C_j, then its tuple contains v_{n+j} in domain C_j; and (iv) if an individual is *not* also included in any generic object in cluster C_j, then its tuple contains a blank (—) in domain C_j.

Table II illustrates how Codd relations may appear (at some point in time) for the generic objects "vehicle," "motorized vehicle," and "air vehicle" in Figure 4. Notice that one consequence of the representation method is the appearance of relation names as values in domains. For example, the domains "medium category" and "propulsion category" in the relation "vehicle" have relation names as values. This allows us to employ Codd's relational operators in the manipulation of generic objects. As we shall see in Section 4, it also allows us to use a uniform method for representing relationships in which objects, either generic or individual, participate.

We call the domain in a relation which contains the name of a descendant relation the *image domain* for that descendant. For example, the domain "medium category" in the relation "vehicle" is the image domain for the descendant relations "land vehicle," "air vehicle," and "water vehicle." There is a one-to-one correspondence between clusters in a generic hierarchy and image domains in its relational representation.

Notice that in Table II, with the exception of the image domains, all domains in "vehicle" are inherited by its descendant relations "motorized vehicle" and "air vehicle." Although this domain inheritance may often be appropriate in relational models, we do not insist that it must occur. This allows generalization abstractions to be represented in the manner most appropriate to their users.

It is clear that a great deal of information occurs redundantly in a relation hierarchy. This is perfectly acceptable provided there is some way to *implement* a relation hierarchy such that (i) storage space is not wasted owing to data duplication, and (ii) consistency of redundant information can be maintained. This issue is further discussed in Section 6.

Table II. Examples of Codd relations for three generic objects from the hierarchy of Figure 4

vehicle:

iden. num.	manufacturer	price	weight	medium category	propulsion category
V1	Mazda	65.4	10.5	land veh.	motorized veh.
V2	Schwin	3.5	0.1	land veh.	man powered veh.
V3	Boeing	7,900	840	air veh.	motorized veh.
V4	Aqua Co	12.2	1.9	water veh.	wind propelled veh.
V5	Gyro Inc	650	150	air veh.	motorized veh.

motorized vehicle:

iden. num.	manufacturer	price	weight	horse-power	fuel capacity	motor category
V1	Mazda	65.4	10.5	150	300	rotary veh.
V3	Boeing	7,900	840	9600	2600	jet veh.
V5	Gyro Inc	650	150	1500	2000	rotary veh.

air vehicle:

iden. num.	manufacturer	price	weight	maximum altitude	takeoff distance	lift category
V3	Boeing	7,900	840	30	1000	plane
V5	Gyro Inc	650	150	5.6	0	helicopter

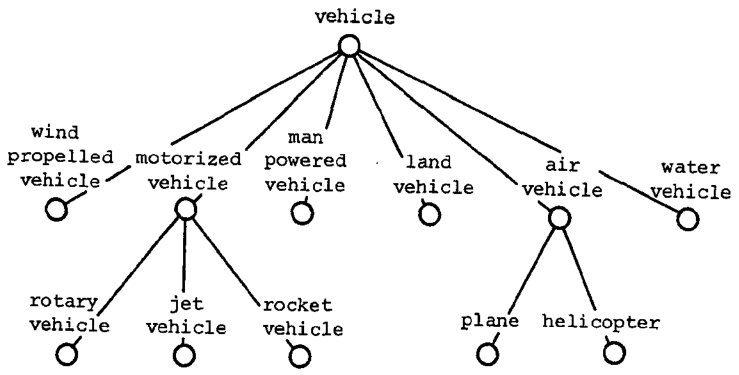

Fig. 4. A generic hierarchy over vehicles

3. THE GENERIC STRUCTURE

We now describe a structuring primitive for specifying generalizations in relational models. In [4] we introduced the types **collection** and **aggregate** and declared a relation R by writing:

```
var R: collection of aggregate [keylist]
                          s1: {key} R1;
                            ...
                          sn: {key} Rn
                        end
```

In this declaration "keylist" contains the selectors for the key domains of R. The curly brackets around "**key**" indicate that it does not always have to occur. Now R can actually be thought of as the name of a generic object. To define the position of the generic object R in a generic hierarchy, we only need to specify its descendants in this hierarchy. This suggests that the structure shown in Figure 5 is appropriate for defining Codd relations.

We use the term **generic** rather than the looser term **collection** to indicate that a generic object is being defined. The **generic** structure simultaneously specifies two abstractions: (i) It specifies R as an aggregation of a relationship between objects R_1 through R_n, and (ii) it specifies R as a generalization of a class containing objects R_{11} through R_{mp_m}. The domains with selectors s_{k1} through s_{km} are image domains. If no image domains are specified, then the **generic** structure is the same as the **collection** structure of [4].

Before discussing the five syntactic requirements of the **generic** structure, we define the three relations of Table II. These definitions are shown in Figure 6. Note in the definition of "vehicle" that its generic descendants are listed following "**generic**" and its aggregate descendants are listed following "**aggregate.**" The generic descendants are grouped into clusters, and each cluster is associated with the selector of its corresponding image domain. In this case the image domains have selectors MC and PC.

We shall not discuss the first two syntactic requirements in Figure 5—these are explained in [4]. Requirement (iii) demands that each generic descendant of R be declared elsewhere as a generic object in its own right. In Figure 6 this is illustrated by the declarations of "motorized vehicle" and "air vehicle." Furthermore, these

```
var R: generic
          sk1 = (R11, ... , R1p1);
            ...
          skm = (Rm1, ... , Rmpm)
        of
        aggregate [keylist]
          s1: {key} R1;
            ...
          sn: {key} Rn
        end
```

where:

(i) R_i $(1 \leq i \leq n)$ is either a generic identifier (in which case "**key**" must appear) or a type identifier (in which case "**key**" must not appear);

(ii) "keylist" is a sequence of s_i's $(1 \leq i \leq n)$ separated by commas;

(iii) each R_{ij} $(i = 1, 1 \leq j \leq p_1; \dots ; i = m, 1 \leq j \leq p_m)$ is a generic identifier whose key domains are the same as those of R;

(iv) each s_{ki} $(1 \leq i \leq m)$ is the same as some s_j $(1 \leq j \leq n)$;

(v) if s_{ki} is the same as s_j, then the type "{**key**} R_j" is the range $(R_{i1}, \dots , R_{ip_i})$.

Fig. 5. The **generic** structure

```
var vehicle:
  generic
      MC = (land vehicle, air vehicle, water vehicle);
      PC = (motorized vehicle, man powered vehicle, wind propelled vehicle)
  of
  aggregate [ID#]
        ID#: identification number;
          M: manufacturer;
          P: price;
          W: weight;
         MC: medium category;
         PC: propulsion category
  end
var motorized vehicle:
  generic
     MTC = (rotary vehicle, jet vehicle, rocket vehicle)
  of
  aggregate [ID#]
        ID#: identification number;
          M: manufacturer;
          P: price;
          W: weight;
         HP: horsepower;
         FC: fuel capacity;
        MTC: motor category
  end
var air vehicle:
  generic
      LC = (plane, helicopter)
  of
  aggregate [ID#]
        ID#: identification number;
          M: manufacturer;
          P: price;
          W: weight;
         MA: maximum altitude;
         TD: takeoff distance;
         LC: lift category
  end
```

Fig. 6. Definitions for the relations in Table II

descendant objects must all have the same key domains as R. This requirement allows us to reference individual objects in a uniform way regardless of the generic class in which they appear. An occasionally useful exception to this rule is that if s_{ki} is in the key of R, then it does not have to appear in the key of any R_{ij} ($1 \leq j \leq p_i$). If it did appear, the domain would have the same value for each individual in R_{ij}. For example, suppose PC was declared as belonging to the key of "vehicle" in Figure 6. If we then added "propulsion category" as a new domain in the relation "motorized vehicle" in order to satisfy requirement (iii), this domain would have the same value ("motorized vehicle") in every individual.

Requirement (iv) ensures that each specified cluster is associated with a particular (image) domain. Requirement (v) ensures that each image domain can actually assume as values the generic identifiers listed in its associated cluster. For example, in the definition of "vehicle" in Figure 6, the type "medium category" must be defined (elsewhere) as ranging over the identifiers "land vehicle," "air vehicle," and "water vehicle."

If the definition of a relation satisfies the five syntactic requirements, this is no guarantee that the definition specifies a meaningful aggregation abstraction and a meaningful generalization abstraction. We now consider what additional requirements are necessary for meaningful abstractions to be specified. These requirements must be semantic and somehow related to our intuitive understanding of the real world.

Since databases are usually designed to model the real world as we understand it, we can safely require that all object names in a relation definition be natural language nouns. These nouns then provide the bridge between our intuitive understanding of the real world and its intended reflection in the relation definition. If natural language nouns are not used, any discussion of the meaningfulness of a relation definition seems moot.

Assuming that R, each R_i, and each R_{ij} (in Figure 5) are all natural language nouns, five semantic conditions are necessary for a relation definition to specify an aggregation and a generalization:

(i) Each R-individual must determine a unique R_i-individual.

(ii) No two R-individuals determine the same set of R_i-individuals for all R_i whose selectors are in "keylist."

(iii) Each R_{ij}-individual must also be an R-individual.

(iv) Each R-individual classified as R_{ij} must also an R_{ij}-individual.

(v) No R_{ij}-individual is also an R_{ik}-individual for $j \neq k$.

By an R (R_i or R_{ij})-individual, we mean an instance of the generic object R (R_i or R_{ij}) as it occurs in the real world.

The first two conditions are necessary for an aggregation abstraction and are discussed in [1]. The remaining three conditions are necessary for a generalization abstraction. Condition (iii) ensures that R_{ij} is a subclass of R and thus that R_{ij} can be generalized to R. For example, in Table II, the motorized vehicles V1, V3, and V5 are also vehicles. Condition (iv) ensures that R_{ij} contains all R-individuals classified as belonging to R_{ij}. For example, in Table II, "motorized vehicle" contains all vehicles so classified in "vehicle." Condition (v) ensures that clusters contain mutually exclusive classes.

We say that a relation is *well-defined* if its definition satisfies the five semantic requirements above.

4. MODELING WITH THE GENERIC STRUCTURE

The objective of this section is to show how aggregation and generalization abstractions are used in designing real world models. When aggregation and generalization are employed separately, they can only model relatively simple situations. However, by employing them together, we will see that a rich variety of models can be defined. We first describe a graphical representation for relation definitions. This notation makes models much easier to visualize.

The notation is based on the observation that aggregation and generalization are independent activities. Given a particular object, generalizations of this object can be considered independently of relationships in which the object participates. This suggests a graphical notation in which generalization and aggregation are represented *orthogonally*. We have chosen to represent aggregation in the plane of the page, and generalization in the plane perpendicular to the page. The **generic** structure of Figure 5 is denoted graphically in Figure 7.

High level aggregate objects will appear toward the top of the page and low level aggregate objects toward the bottom. Aggregation therefore occurs *up* the page. High level generic objects will appear (in a simulated three-dimensional space) in the surface of the page and low level generic objects will appear below the surface. Generalization therefore occurs *out of* the page.

Suppose that we must model the "employees" of a certain company. Let's assume that this company has three different types of employees at some point in time—truckers, secretaries, and engineers. Information must be maintained about each individual employee—though different kinds of information are required depending on the type of employee. In addition information must be maintained about each generic type of employee.

Specifically, assume that the attributes "employee ID#," "name," "age," and "employee type" are important for *all* individual employees. The necessary additional attributes for each employee type are given in the following table.

employee type	additional attributes
trucker	vehicle ID#, number of license endorsements
secretary	typing speed
engineer	highest degree, type (mech, elec, etc.)

For each generic type of employee the following attributes must be recorded: number of employees (size), number of vacant positions (vacancies), hiring agency (agency).

We will now construct a relational model which meets the preceding require-

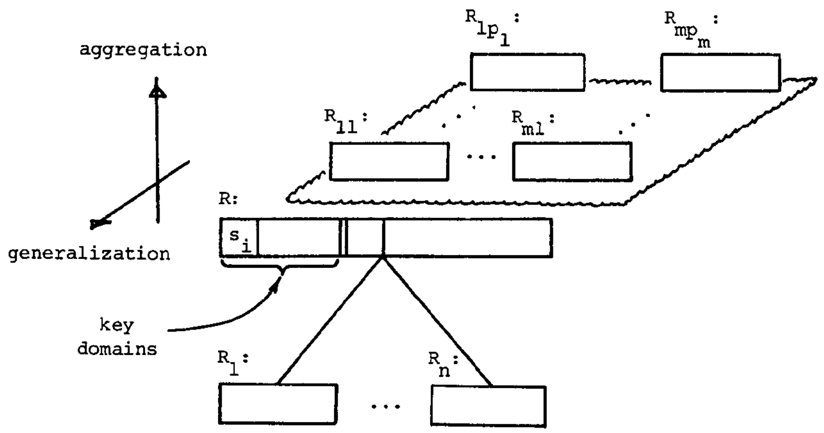

Fig. 7. A graphical notation for the **generic** structure

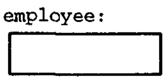

Fig. 8. Initial model for "employee"

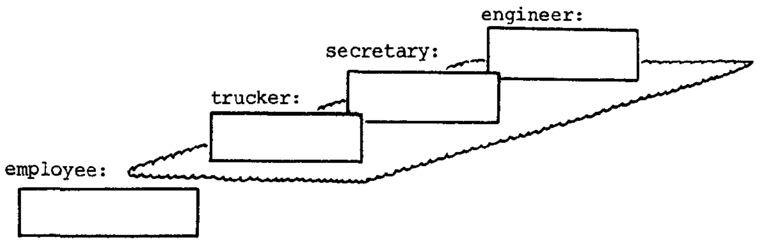

Fig. 9. Decomposition of "employee" in the generalization plane

ments. Since the most abstract object that must be represented is "employee," the initial model contains precisely this object. This is shown in Figure 8. The next step is to decompose this object in both the aggregation and generalization planes. We have decided to decompose first along the generalization plane. In this plane the components of "employee" are the generic objects "trucker," "secretary," and "engineer." These three objects form a single cluster. The model then appears as in Figure 9.

We now decompose "employee" in the aggregation plane. There are four attributes of an employee that must be recorded: employee ID#, name, age, and employee type. We therefore include four objects as the components of "employee." These objects are included in Figure 10. With the exception of "employee-type," the components of "employee" are *primitive* objects—that is, they are only thought of as a whole. The object "employee-type," however, has several components which are important. These correspond to the attributes: typename, size, vacancies, and agency. All these components are primitive. The model now appears as in Figure 10.

We can now continue to develop the model by decomposing any object (which is not already decomposed) along either plane. Since we are not interested in any subtypes of "employee-type," this object is not decomposed along the generalization plane. For similar reasons, we do not decompose "trucker," "secretary," or "engineer" along the generalization plane. However, these three objects must each be decomposed into (primitive) objects in the aggregation plane. These decompositions are included in Figure 11. To avoid cluttering the figure, we have left incomplete the lines which should connect each of the objects "emp. ID#," "name," and "age" with each of the objects "trucker," "secretary," and "engineer." Finally, we can choose selectors and keys for each object. The relational model then appears as in Figure 11.

Definitions for the relations "employee" and "employee-type" are given in Figure 12. Notice that the relation "employee-type" describes details about "trucker," "secretary," and "engineer" when they are thought of as generic objects. The relation "employee" refers to these details through the image domain (i.e. the domain with selector "TN"). This allows each individual employee to

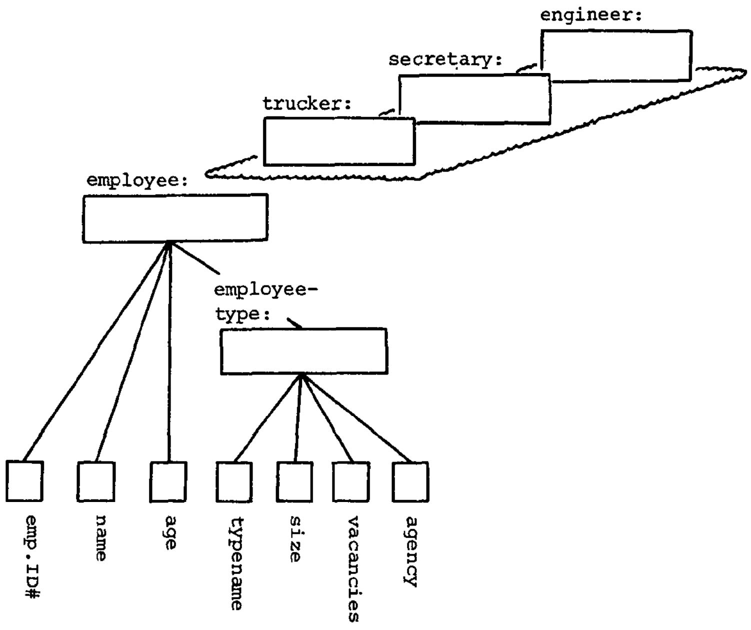

Fig. 10. Decomposition of "employee" in the aggregation plane

refer to all the generic properties of the employee subclass to which it belongs.

In general a relation R may have several clusters in the generalization plane which belong to it. For each cluster C there will be a corresponding relation referenced from R in the aggregation plane. This relation will describe the attributes (if any) of the generic objects in C. We can summarize this general property of relational models as: *The generic components of an abstract object R have their attributes defined in relations which are aggregate components of R.*

We now consider the modeling of various additional aspects of the real world related to employees. As our first aspect, assume that the "trade unions" affiliated with different employee types must be modeled. The attributes of a trade union that are relevant include its name, address, and senior officer. Figure 13 shows the object "trade union" in the aggregation plane of "employee"—the generalization plane is omitted (see Figure 10).

We are also interested in the "affiliation" of employee types with trade unions. We therefore form an abstract object "affiliation" as an aggregation of "employee-type" and "trade union." Let's assume that an important attribute of an affiliation is the person empowered to speak for union members of a particular employee type. An appropriate object, say "spokesman," must therefore be included as an additional component of "affiliation." When this is done, the model appears as in Figure 14. With selectors and keys appropriately chosen, a definition of "affiliation" is given below:

```
var affiliation:
  generic of
  aggregate [TN, UN]
```

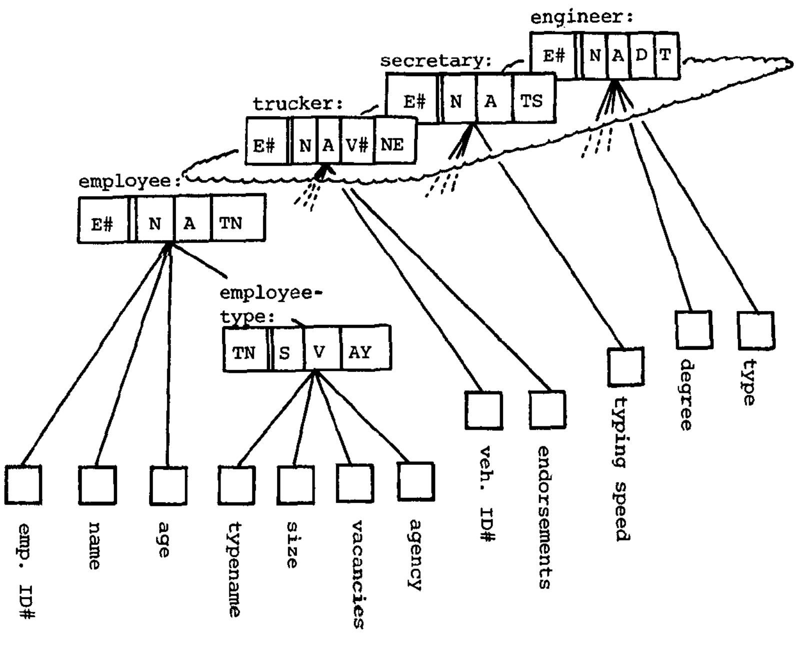

Fig. 11. A relational model for "employee"

```
var employee:
  generic
    TN = (trucker, secretary, engineer)
  of
  aggregate [E#]
      E#: emp. ID#;
       N: name;
       A: age;
      TN: key employee-type
  end

var employee-type:
  generic of
  aggregate [TN]
      TN: typename;
       S: size;
       V: vacancies;
      AY: agency
  end
```

Fig. 12. Definitions for "employee" and "employee-type" in Figure 11

```
      TN: key employee-type;
      UN: key trade union;
       S: spokesman
    end
```

This definition specifies a relationship over generic objects.

We next consider some relationships over the generalization plane of "employee"

(Figure 10). Let's assume that the company where the employees work owns a fleet of vehicles (including cars and trucks). We will model the relationship between employees and vehicles determined by employee usage of vehicles. In particular trucks are used by truckers for hauling materials and cars are used by engineers to visit distant sites. No other (official) vehicle use is permitted. The representation of "employee" and "vehicle" in the generalization plane is shown in Figure 15.

There are two ways that this relationship between employees and vehicles could be modeled. One way is to abstract a *single* relationship between "employee" and "vehicle" as an aggregate object (say, "trip"). Unfortunately this approach is too general to fully capture the intended relationship. It would seem that *any* employee could take *any* vehicle on a trip—including a secretary driving a truck. Furthermore, there is no distinction between trips where materials are hauled and trips where engineers visit a site. It is very likely that different attributes of a trip are important depending on which kind of trip it is.

The second way of modeling the required relationship between employees and vehicles is to decompose it, in the generalization plane, into two relationships—

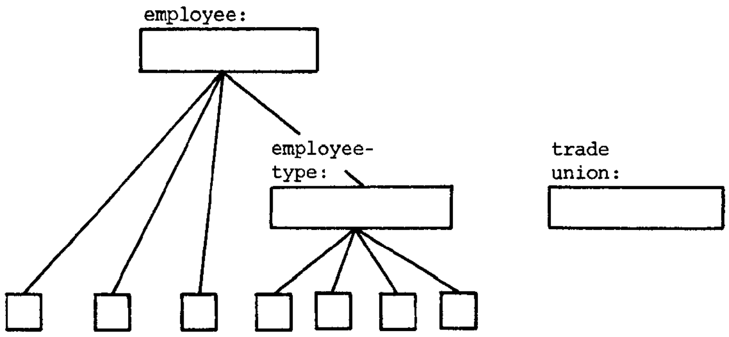

Fig. 13. "Trade union" incorporated in the aggregation plane of "employee"

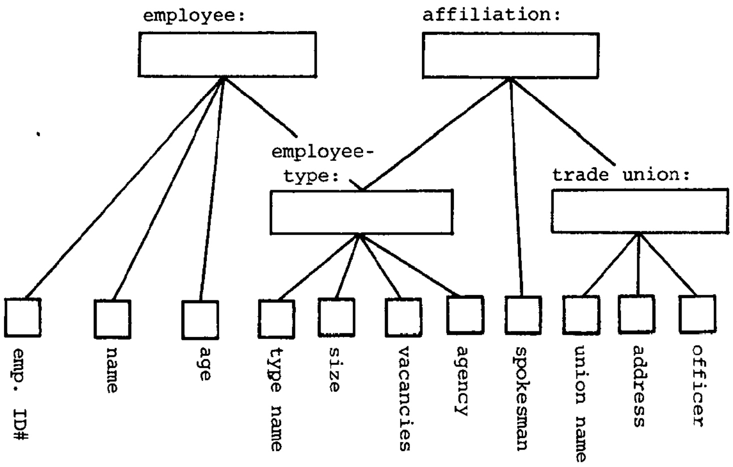

Fig. 14. "Affiliation" incorporated in Figure 13

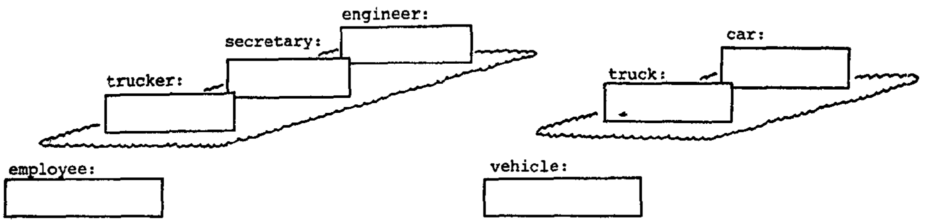

Fig. 15. Decomposition of "employee" and "vehicle" in the generalization plane

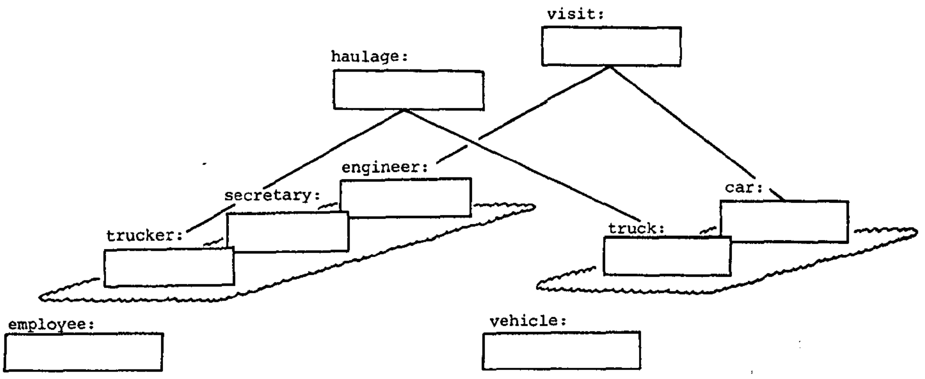

Fig. 16. "Haulage" and "visit" incorporated in Figure 15

one between "trucker" and "truck" and the other between "engineer" and "car." The first relationship can be abstracted as the aggregate object "haulage" and the second as "visit." These objects are shown in Figure 16.

The main advantage of this second approach is that the model is a more precise representation of reality. This makes the model more understandable and thus less prone to erroneous access and manipulation. Furthermore, the model expresses real world constraints on "haulage" and "visit"—namely that "haulage" must relate truckers to trucks and "visit" must relate engineers to cars. If these constraints are enforced during update operations, then certain integrity problems (e.g. secretaries who drive trucks) can be avoided. Another advantage is that by restricting "haulage" and "visit" to explicit subsets of "employee" and "vehicle" there is more scope for optimizing retrieval operations at a lower level of implementation.

Let's consider another example of a relationship over the generalization plane of employee. We assume that the assignment of secretaries to engineers must be modeled. The most direct way of modeling this relationship is to abstract an object (say, "assignment") as an aggregation of "secretary" and "engineer." The object "assignment" is shown on the right in Figure 17. Although this relationship could be modeled as an aggregation of "employee" and "employee," to do so would create unnecessary ambiguity.

The previous three relationships (haulage, visit, and assignment) all have the property that employees of some, but not all, types participate. We now examine a relationship in which employees of all types participate. Consider the relationship

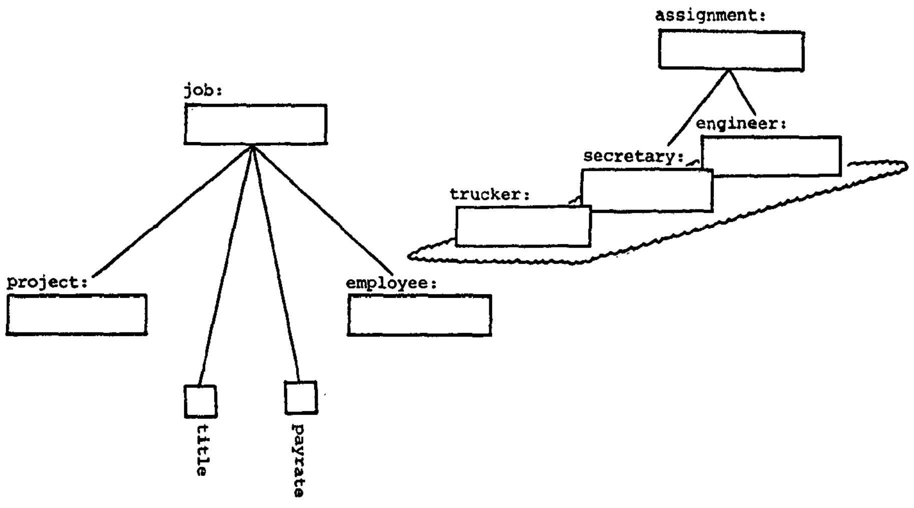

Fig. 17. "Assignment" and "job" incorporated in Figure 16

between employees and projects which identifies the jobs employees hold on projects. We assume that a given employee may hold jobs on several projects simultaneously but not several jobs on the same project. This relationship is best modeled by aggregating "employee" and "project" into an abstract object (say, "job"). Other attributes of "job" such as "title" and "payrate" may be introduced. The object "job" is shown in Figure 17.

A definition of "job," with appropriate choices for selectors and key, is given below:

```
var job:
  generic of
  aggregate [E#, P#]
       E#: key employee;
       P#: key project;
        T: title;
       PR: payrate
  end
```

The key "E#, P#" is chosen since, by the constraints given above, jobs are in one-to-one correspondence with employee-project pairs. The definition assumes that no decomposition of "job" in the generalization plane is of interest. Alternatively, suppose some model users are interested in specific types of jobs. Let's assume that among the projects is one to develop modems and another to upgrade printers. Some users may want to think of "modem project job" and "printer project job" as generic objects. Figure 18 shows how the model appears when these objects are incorporated as descendants of "job" in the generalization plane.

Let's examine why Figure 18 expresses the appropriate structure. First, since we are decomposing job into generic objects (in the generalization plane), a new object must be introduced (in the aggregation plane) which is a generalization of these objects (thought of as individuals). This follows from the requirement of the

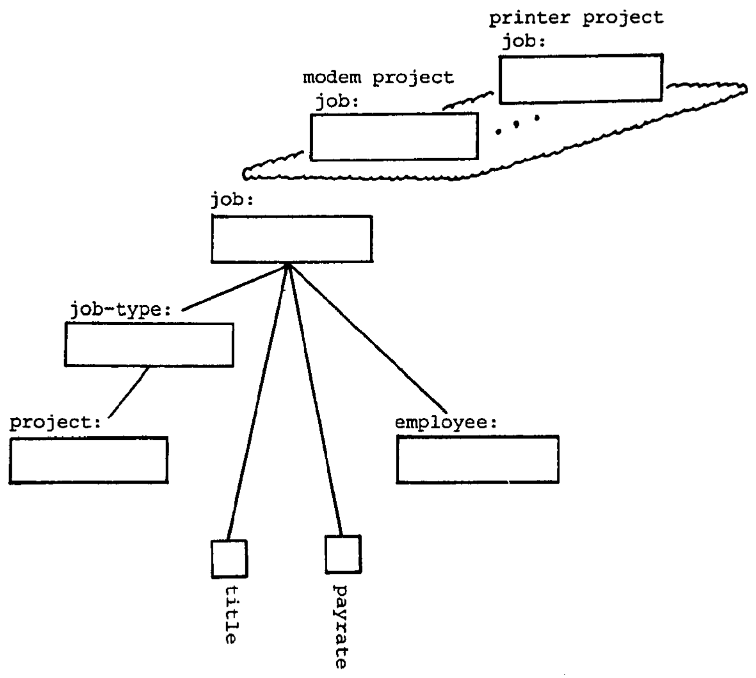

Fig. 18. Effect of decomposing "job" in the generalization plane

generic structure that each cluster have its own image domain. This new object is called "job-type" and is a component of "job." "Job-type" may itself be decomposed into its attribute objects.

Second, note that each job type is the abstraction of all jobs belonging to a certain project. Thus one attribute of a job type is the project to which all its jobs belong. Accordingly, in Figure 18 "project" is shown as an attribute of "job-type." "Project" is no longer a *direct* component of "job" as in Figure 17—it is now an *indirect* component via "job-type." Definitions of "job" and "job-type" are given below:

```
var job:
  generic
    TN = (modem project job, ... , printer project job)
  of
  aggregate [E#, TN]
     E#: key employee;
      T: title;
     PR: payrate;
    TN: key job-type
  end

var job-type:
  generic of
  aggregate [TN]
    TN: typename;
     P#: key project;
    AP: average payrate
  end
```

Note that "job" is an example of a relation in which the key contains an image domain.

The introduction of "job-type" illustrates a form of *restructuring* that may often become necessary as the application of a model evolves. In the initial application users may be interested in a general class C of objects. As the application evolves, certain users may become interested only in the subcategory of C-objects whose A attribute is some value v_1, while others may become interested in the subcategory whose A attribute is v_2. These *subcategories of C with respect to attribute A* must then be represented as a cluster in the generalization plane of C. In Figure 18 the cluster {modem project job, ... , printer project job} represents a subcategorization of "job" with respect to "project."

In general, if it becomes necessary in an existing model to subcategorize an object O with respect to attribute A, the following restructuring is required: (i) Define a new object S which abstracts the class of subcategories; (ii) replace in O the domain which references A by a domain which references S; (iii) if the key of O contains the selector for A, replace it with the selector for S; and (iv) insert in S a domain which references A.

5. RELATIONAL INVARIANTS

We first consider the properties of relations that must remain invariant during update operations. We then examine each update operation in turn and consider methods for ensuring the invariance of these properties. The update operations we consider are:

insert—this operation is performed when a new individual object becomes of interest;

delete—this operation is performed when an existing individual object ceases to be of interest; and

modify—this operation is performed when the details of an existing individual object are subject to change.

In Section 3 we stated the conditions that must be satisfied by the real world objects named in a relation definition in order for the definition to express abstractions. If an object R is being defined in terms of the aggregate components R_i and the generic components R_{ij}, these conditions are:

(i) Each R-individual must determine a unique R_i-individual.

(ii) No two R-individuals determine the same set of R_i-individuals for all R_i whose selectors are in "keylist."

(iii) Each R_{ij}-individual must also be an R-individual.

(iv) Each R-individual classified as R_{ij} must also be an R_{ij}-individual.

(v) No R_{ij}-individual is also an R_{ik}-individual for $j \neq k$.

It is important that the relations and tuples which represent these real world objects at some point in time satisfy a corresponding set of conditions. This ensures that the relations faithfully represent the abstract structure of the real world. In stating these conditions the following definitions are useful.

Let t be a tuple in R_{ij}. A *parent image* of t is a tuple t' in R for which: (i) t and t′ have the same values in all common domains and (ii) t′.s_{ki} has the value "R_{ij}." Semantically speaking, a tuple and its parent image both describe the same real world individual; however, the parent image is at a higher level of generalization.[4] If t′ is a parent image of t, we say that t is a *child image* of t′.

We can now state the conditions for a relation, together with its aggregate and generic components, to represent abstractions:

(i) For each R-tuple t, if t.s_i is nonblank, then when R_i is a generic identifier t.s_i is the key of an R_i-tuple, and when R_i is a type identifier t.s_i is of type R_i.

(ii) No two distinct R-tuples have the same key.

(iii) Each R_{ij}-tuple has a parent image in R.

(iv) For each R-tuple t, if t.s_{ki} is nonblank and has the value R_{ij}, then R has a child image in R_{ij}.

(v) No R-tuple has a child image in both R_{ij} and R_{ik} for $j \neq k$.

We call these five conditions *relational invariants*.[5] They are a transformation of the prior five conditions in accordance with the method of representing generic objects as relations. There is one minor exception embodied in the first and fourth invariants. In practice it is necessary to permit the occurrences of blanks (which mean "unknown" or "don't care") in some domains. The first and fourth invariants make allowance for this possibility.

The relational invariants can be thought of as constraining a relation to represent an "abstract object." These invariants must be satisfied by *every* relation in a relational model irrespective of the kind of abstract object the relation represents. In addition each relation must usually satisfy a set of invariants (often called "integrity constraints") which are peculiar to that relation alone. These invariants constrain a relation to the representation of a *particular kind* of abstract object. Since the relational invariants apply to every relation, they are the most fundamental form of integrity constraint.

Users do not normally interact with a database at the level of primitive update operations such as insert, delete, and modify. In most cases these primitives are used to construct higher level update operations called *transactions*. The notion of a transaction allows the difference between the relational invariants and special integrity constraints to be characterized in another way. The relational invariants must be satisfied before and after each user initiated update operation. However, special integrity constraints need only be satisfied before and after each transaction.

We now discuss the maintenance of the relational invariants during update operations. Throughout this discussion we assume an idealized situation in which (i) the model correctly abstracts the real world structure at the time of update, (ii) all update information accurately reflects the attributes of real world individuals, and (iii) all users have full access rights to all relations. This idealized situation allows us to ignore the separate issues of special integrity constraints and

[4] A tuple may have several parent images, though each must be in a different relation (for example, consider the hierarchy of Figure 2).

[5] These invariants may be used as axioms to specify (in the manner of Hoare [2]) the semantics of the **generic** structure. Other axioms are required in addition to the five invariants.

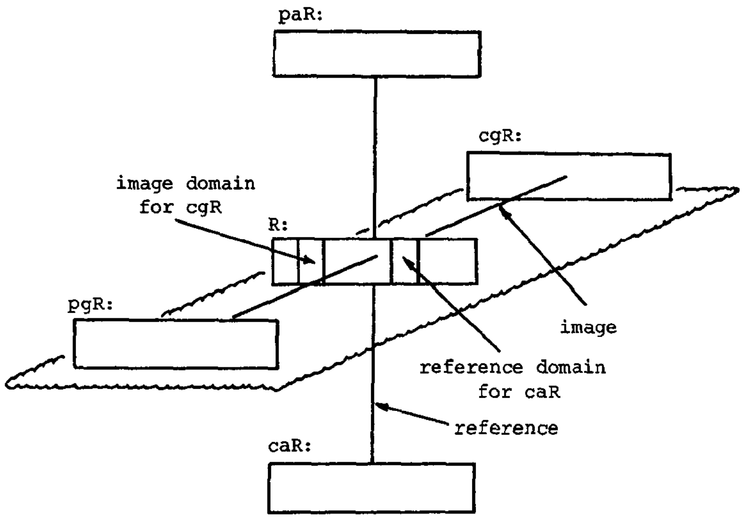

Fig. 19. A relation R and its paR, caR, pgR, and cgR relations

access control. We contend that these issues are best attacked *after* methods have been developed to maintain the relational invariants.

The following abbreviations will be useful in the subsequent discussion. Consider a relational model M which contains, among others, a relation R. Relative to R, we say that a relation is:

paR	if it is a *p*arent in the *a*ggregation plane of *R*;
caR	if it is a *c*hild in the *a*ggregation plane of *R*;
pgR	if it is a *p*arent in the *g*eneralization plane of *R*;
cgR	if it is a *c*hild in the *g*eneralization plane of *R*.

Figure 19 illustrates how a relation R is "connected" to its paR, caR, pgR, and cgR relations.

In Table III we summarize methods for maintaining the relational invariants during update operations. Each type of update operation (insert, delete, modify) is considered in a separate table. Each table records the effect on the five invariants of performing an update operation on a tuple t with key k in a relation R.

It is assumed that the relational invariants are satisfied prior to the update operation. Each table shows how the invariants could be violated as a result of the update operation and also what action can be taken to correct each violation. The action is always to perform an update operation on one or more paR, caR, pgR, or cgR relations. These actions may in turn cause further violations and thus update operations on other relations. In this way a single user initiated update operation can trigger additional update operations which propagate along both aggregation and generalization planes.

We first explain some entries in the tables of Table III and then examine some important properties of triggered operations. As is usual in relational models, we assume that no tuple is ever allowed to have a blank in a key domain. This should be the *only* reason for disallowing a "correct" update operation.

The only time that invariant (ii) can be violated by a correct update is when a duplicate tuple is inserted. In this case the corrective action is simply to delete one duplicate. Invariant (v) can never be violated given our correctness assumptions. Invariants (i), (iii), and (iv) can be violated by any correct update. In each case the corrective action is in accordance with the abstract structure of relational models. Let's consider a few examples.

Table III. Maintenance of the relational invariants during update operations. Part 1 (Part 2 appears on next page.)

INSERT

Invariant	Possible violation after insertion	Method for correcting violation
i)	t references a non-existent tuple in some caR relation R'	*insert* a tuple in R' with the appropriate key values and blanks elsewhere
ii)	t occurs twice in R	*delete* either occurrence of t
iii)	t does not have a parent image in some pgR relation R'	<u>if</u> a tuple with key k already occurs in R' <u>then</u> *modify* this tuple so that it becomes the parent image of t <u>otherwise</u> *insert* a parent image in R' (use blanks where values are unknown)
iv)	t does not have a required child image in some cgR relation R'	*insert* a child image in R' (use blanks where values are unknown)
v)	none	

DELETE

Invariant	Possible violation after deletion	Method(s) for correcting violation
i)	a tuple in some paR relation references a non-existent R-tuple	a) *modify* the tuple so that the reference is replaced by a blank (only possible if reference is not part of tuple's key) b) *delete* the tuple
ii)	none	
iii)	a tuple in some cgR relation has no parent image in R	*delete* the tuple
iv)	a tuple in some pgR relation does not have a required child image in R	a) *modify* the tuple by replacing its image domain value by a blank (only possible if image domain is not part of tuple's key) b) *delete* the tuple
v)	none	

Table III. Part 2
(Part 1 appears on previous page.)

MODIFY

Invariant	Possible violation after insertion	Method for correcting violation
i)	t references a non-existent tuple in some caR relation R'	*insert* a tuple in R' with the appropriate key values and blanks elsewhere
	(key domain modification) a tuple in some paR relation references a non-existent R-tuple	*modify* the tuple so that it references t
ii)	none	
iii)	t does not have a parent image in some pgR relation R'	*modify* t's old parent image so that it becomes t's (new) parent image
	a tuple in some cgR relation R' has no parent image in R	<u>if</u> image domain for R' is modified <u>then</u> *delete* the tuple <u>otherwise</u> *modify* the tuple so that t becomes its parent image
iv)	t does not have a required child image in some cgR relation R'	<u>if</u> image domain for R' is modified <u>then</u> *insert* a child image in R' (use blanks where values are unknown) <u>otherwise</u> *modify* t's old child image in R' so that it becomes t's (new) child image.
	a tuple in some pgR relation does not have a required child image in R	*modify* the tuple so that t becomes its child image
v)	none	

Suppose we violate invariant (i) by inserting a tuple t. This means that t must reference a nonexistent tuple in some relation R′. To correct this violation we must insert an appropriate tuple (say t′) in R′. Lacking other input information, the only detail we know about t′ is its key (which appears in t). Blanks must therefore be inserted in the nonkey domains of t′. Semantically, the effect of this corrective action is to introduce the abstract object t′ which is known to participate in the relationship t.

Suppose we violate invariant (iii) by deleting a tuple t. This means that the tuple (say t′), which was the child image of t, no longer has a parent image in R. To correct this violation we must delete the tuple t′. This corrective action reflects the semantic requirement that an object which does not appear in a class at a high level of generalization cannot appear in a class at a lower level of generalization.

Suppose we violate invariant (iv) by modifying a tuple t. There are two ways in which such a violation can occur. The first way is when t, after modification, has a value (say R′) in some image domain (say d) and yet t has no child image in R′. In this case the corrective action depends on whether or not d was changed by the modify operation. If d was not changed then t's old child image in R′ must be modified to make it t's new child image—all the information required already occurs in t. If d was changed then a child image must be inserted into R′. The key of this child image is contained in t—however, blanks will have to be inserted in any domain whose value is not contained in t.

The second way in which a violation can occur is when t's old parent image (say t′) no longer has a child image in R. In this case the corrective action is to modify t′ so that it becomes once again the parent image of t. In all cases the corrective action reflects the semantic requirement that the details of an abstract object must be consistent no matter what the level of generalization of the class in which it appears.

Examination of Table III will reveal that there are two methods for correcting violations of invariants (i) and (iv) following a delete operation. Each of these methods is semantically appropriate for certain situations. Let's consider invariant (i) first. Suppose we have an abstract object x which has an object y as an aggregate component. The question is whether x should be deleted if y is deleted. The answer seems to depend on the context.

For example, suppose a certain car has a "radio" and a "unibody" as two of its aggregate components. If the radio is destroyed, the car is normally considered to remain in existence. However, if the "unibody" is destroyed, then the car is normally considered to have also been destroyed. Since such questions about deletion can only be answered within a higher semantic framework, it is necessary to leave open the decision of how invariant (i) is maintained. When transactions are designed, the programmer can select which option is most appropriate relative to the integrity constraints he wants to maintain.

Similar considerations apply to invariant (iv). Suppose we have a general class x of objects which includes all objects from a more special class y. The question is whether an object should be deleted from x if it is deleted from y. For example, suppose that an object ceases to be a "Ford car." Since it is not very meaningful to change the manufacturer of a car, the presumption is that the object has ceased to be a "car" of any sort. On the other hand, suppose that an object ceases to be a "red car." Since it is very easy to repaint a red car in the color blue, the presumption might be that the object still remains a "car." The decision as to how invariant (iv) should be maintained under deletion operations must therefore be left to a higher level of modeling.

From the previous discussion, it should be clear that the maintenance procedures in Table III are designed to reflect the abstract structure of the real world. In principle, therefore, no matter what relational model is involved, triggered sequences of update operations should reflect "natural" side effects of an original user defined update. However, if we disregard our real world interpretation and consider the question formally, it is not at all easy to verify that sequences of triggered updates will necessarily be well behaved.

Conceptually, triggered updates are propagated simultaneously along many paths. What happens when two paths cross? Can one triggered update undo the work of another? Is the net result of a user defined update independent of the order in which triggered updates are scheduled? Can a sequence of triggered updates become cyclic? If a triggered update cannot be accepted (because invariant (ii) or (v) would be violated), then the sequence of triggered updates must be backed up and the original update disallowed. How do we know that no "correct" user defined update will be disallowed for this reason?

We would require that any invariant maintenance procedure satisfy at least the following three properties:

(i) The propagation of triggered updates must eventually terminate.

(ii) The overall effect, after propagation terminates, is independent of the order in which triggered updates are scheduled.

(iii) A "correct" user defined update is only disallowed when it requires a blank to be inserted in a key domain.

We call these properties *termination*, *determinancy*, and *compliancy*. We conjecture that the invariant maintenance procedure of Table III satisfies all three properties.

6. CONCLUDING REMARKS

In this section we discuss several aspects of the **generic** structure, including database design methods, implementation techniques, and query languages. We will begin with database design methods—the discussion will first concentrate on "normal forms" and then move on to consider design methodology.

In Section 3 we said that a relation is well defined if it satisfies the five relational invariants. Codd [1] and others have proposed several *normal forms* which are also supposed to capture some notion of a "good" relation. What is the relationship between a well-defined relation and a normal-form relation?

Essentially, normal forms are an attempt to formalize the notion of "basic aggregate object" in terms of the concept of functional dependency. These normal forms are not intended to, and do not, address properties of generalization. Given an aggregate object, these normal forms are useful in deciding whether the object is basic (i.e. that the object has no other object embedded within it). However, normal forms are not, at present, able to distinguish what is, or is not, an aggregate object.

Intuitively an aggregate object is something that we can *name* with a noun (or noun phrase) and presumably think of as a whole. By requiring relations to be named, the **aggregate** structure captures this aspect of an aggregate object. Normal forms do not explicitly consider naming and as a result normal-form relations may be effectively unnameable (except by a sentential description). For example, the relation R in Figure 20 satisfies (we believe) all published normal forms. However, the relation does not represent an aggregate object because it has, apparently, no name which satisfies the semantic requirements for being well defined.

R(man, property value, woman, property value)

Tuple $\langle x, y, z, w\rangle$ *is in R iff x owns a piece of property worth y, z owns a piece of property worth w, and* $y > w$.

(Assume that both men and women can own several pieces of property.)

Fig. 20. A normal-form relation which is effectively unnameable

On the other hand, while the **aggregate** structure is a stronger attempt at capturing the notion of "aggregate object," it does not capture the notion of "basic." We suggest, therefore, that "well-defined relation" and "normal-form relation" are complementary criteria for use in database design. Once well-defined relations have been discovered, their internal dependency structure may be checked against normal-form criteria. If a well-defined relation is not in normal form, it may (or may not) be appropriate to further decompose the relation.

In [4] we suggested a methodology for database design which leads to aggregate objects and then to well-defined relations. However, this methodology did not consider decomposition in the generalization plane (i.e. single-level generic hierarchies were implicitly assumed). The methodology can be extended in the obvious way to consider decomposition in both planes. There may be a marginal advantage to decomposing in the generalization plane before decomposing in the aggregation plane. This is because each cluster introduced in the generalization plane requires a new object to be inserted in the aggregation plane.

We now comment on techniques for implementing relational models at lower levels. In such an implementation there are two factors to consider: (i) how the relational *structure* is represented, and (ii) how the relational *invariants* are maintained. One consideration in choosing the structure representation is to save storage space by eliminating redundant information. Such information occurs in both the aggregation and generalization planes. In the aggregation plane it occurs as a result of key values being repeated up the hierarchy. In the generalization plane it occurs when attribute values (key or nonkey) are repeated up the hierarchy. One effect of the relational invariants is to ensure that all redundant information is consistent. Therefore, by removing as much redundancy as possible, the relational invariants should be maintainable in the most efficient manner.

Preliminary investigation indicates that relational models can be represented in terms of DBTG (Database Task Group) "owner-coupled set" structures so that no redundancy occurs. For the aggregation plane, each hierarchic branch becomes a different set type with the upper relation as member type and the lower relation as owner type. Each set instance contains as members all tuples which reference the owner tuple. For the generalization plane, each collection of all hierarchic branches from a given relation becomes a different set type with the upper relation as owner type and all descendant relations as member types. Each set instance contains as members all child images of the owner tuple. It seems likely that most of the relational invariants can be maintained using DBTG Data Definition Language options—however, this issue is not yet fully resolved.

Finally, we discuss some issues associated with access languages for relational models. When we began our study of generalization and its representation in relational models, we had assumed that a higher order query language would be essential to exploit the rich hierarchic structure. It seemed that higher order variables would be needed to range over higher order relations (i.e. relations of relations, etc.) in addition to first order variables which range over first order relations (i.e. relations of individuals).

However, we had underestimated the power of abstraction. Our motivation for the **generic** structure was to provide uniform treatment for all kinds of objects—individual, aggregate, and generic. This means that higher order relations are structurally identical to first order relations. As a result a first order query language appears to be adequate for all relational models no matter how many levels of generalization they contain. In essence, by imposing an appropriate structuring discipline, we have been able to exploit a richness that was already implicit in Codd's original work.

Despite the fact that a first order language may be adequate for querying, there are strong reasons for developing higher order primitives for applications programming. Consider the following query over the relational model of Figure 11:

"*Find out what type of employee E5 is, and then retrieve his record for that employee type.*"

This query can be conveniently expressed in two parts as shown below.

Part 1:
```
  R ← RESTRICT (employee to E# = E5);
  S ← PROJECT (R on TN);
  RETRIEVE S;
```
Part 2 (assuming response to Part 1 is "trucker"):
```
  T ← RESTRICT (trucker to E# = E5);
  RETRIEVE T;
```

Tho first part returns E5's employee type (say "trucker"), and the second part retrieves E5's record from "trucker." In this case the user has to formulate the second part from information obtained in the first part.

If this query were to be implemented as an application program, the user would be "replaced" by a case statement as shown below.

```
READ X;
R ← RESTRICT (employee to E# = X);
S ← PROJECT (R on TN);
T ← case S of
        trucker: RESTRICT (trucker to E# = X);
        secretary: RESTRICT (secretary to E# = X);
        engineer: RESTRICT (engineer to E# = X)
      end
RETRIEVE T;
```

The problem with using the case statement is that the program becomes *dependent* on the generic components of "employee" *at one point in time*. If a new type of employee is hired (say "guard") then the program will not work when the input happens to be the employee number of a guard.

This "time dependency" can be removed with the aid of a higher order operator which we will call SPECIFY. Given the name of a relation, SPECIFY will return the relation so named. We can now rewrite the preceding program as shown below.

```
READ X;
R ← RESTRICT (employee to E# = X);
S ← PROJECT (R on TN);
T ← SPECIFY S;
U ← RESTRICT (T to E# = X);
RETRIEVE U;
```

The new program is independent of the generic components of "employee" at all points in time.

The introduction of higher order primitives can thus increase the stability of programs as the database evolves in accordance with the real world. It is not clear what other higher order operators are useful besides SPECIFY. As far as we know, these aspects of applications programming have yet to be investigated.

ACKNOWLEDGMENTS

We are grateful for the comments of R.W. Taylor and the anonymous referees on an earlier version of this paper.

REFERENCES

1. CODD, E.F. Further normalization of the data base relational model. In *Courant Computer Science Symposium 6: Data Base Systems*, Prentice-Hall, Englewood Cliffs, N.J., May 1971, pp. 33–64.
2. HOARE, C.A.R. Notes on data structuring. In *APIC Studies in Data Processing No. 8: Structured Programming*, Academic Press, New York, 1972, pp. 83–174.
3. QUILLIAN, M.R. Semantic memory. In *Semantic Information Processing*, M.I.T. Press, Cambridge, Mass., 1968, pp. 227-268.
4. SMITH, J.M., AND SMITH, D.C.P. Database abstractions: Aggregation. To appear in *Comm. ACM* in June 1977.

Received November 1976; revised February 1977