

Amihai Motro and Peter Buneman

Department of Computer and Information Science,
 Moore School, University of Pennsylvania,
 Philadelphia, Pa. 19104.

Abstract

Using a semantic data model, a program has been developed that takes as input a set of assertions about two database schemas and generates a schema for their union. This schema is then used for either a virtual or physical representation of the original databases. In the case of a virtual merge, queries against the merged databases are automatically decomposed. In the case of a physical merge, the data transfer programs are automatically generated. In either case, integrity constraints may be automatically generated to enforce consistency in places where the two databases contain overlapping information. The problem is the converse of providing "user views": given two subschemas, what "super-schema" contains both?

1 Introduction

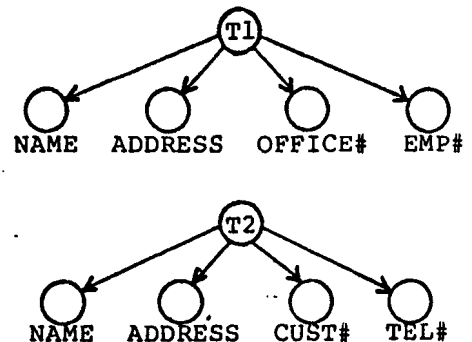
A number of software research and development efforts [1, 2, 3] are currently devoted to the problem of providing a database end-user with a unified method of querying a number of physically distinct databases. An important part of this problem is to construct a global schema that describes the user view of the data, and a local schema for each of the component databases. A mapping from elements of the global schema into elements of the local schemas enables queries expressed in terms of the global schema to be decomposed and translated into a set of queries against the local schemas. In general the local schema for a given database will be a subschema (the data of interest to the end-user) of that database, and it may well be the case that different local schemas will be expressed in different data models.

In the simplest use of this technique, the global schema is no more than a disjoint set of images of the local schemas. This provides the user with a uniform access method for the various databases, but provides no semantic links between them. The user who wishes to merge information from two or more of the component databases will have to write a

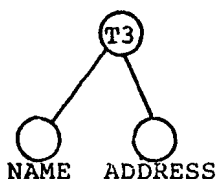
query which explicitly performs the merge (if the query language allows this) or he will have to resort to more cumbersome programs. While to our knowledge no implemented systems yet permit any more sophisticated semantic merging of databases, proposals such as [4] should make this possible in the near future.

The problem we shall attack in this paper is that of creating the global schema. At present this is a task for an applications programmer who must not only be acquainted with the various database management systems, but also have an intimate understanding both of the structures of the component databases and the needs of the end user. We hope to demonstrate that merging two database schemas into a unified structure can proceed automatically from a few simple assertions about the semantic relationships between elements of the two schemas. As a result of this work, it should be possible to provide end-users with a program in which they can interactively examine a variety of local schemas and construct their individual global schemas for particular areas of interest. This may be especially valuable in a "federated" database architecture, where there is no notion of a unique global schema from which all user views are derived. Instead, groups of users agree to share certain subsets of data, and must construct a common structure in which to represent the shared data.

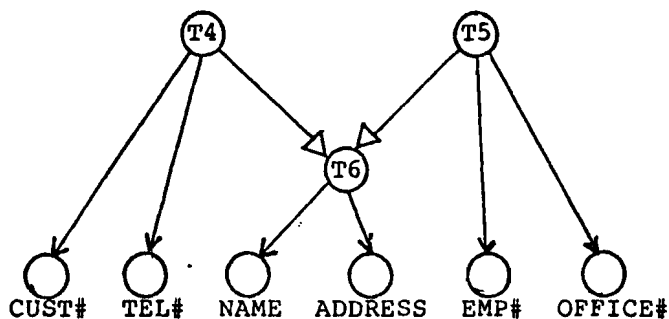
In order to illustrate the problems in performing a merge consider two (very simple) databases:



The arrows in this diagram indicate an attribute relationship, for example that NAME is an attribute of T1. In merging these two databases we may be seeking just the common attributes:



or we may require that a database that represents both the common attributes and the attributes that distinguish the two databases:



where we have introduced a new relationship (indicated by $\longrightarrow \triangleright$). This is a subtype relationship: T4 is a subtype of T6, and as such inherits the attributes of T6, that is, both NAME and ADDRESS are attributes of T4. These two relationships, attribute and subtype, are precisely those of aggregate and generalization proposed by Smith and Smith [5, 6]. The simple, but important, point to be made is that while the original database schemas did not include subtype relationships, it may be necessary to introduce them in order to produce an accurate description of the combined data.

2 The Abstract Database Model

The example in the introduction is oversimplified. The original databases appear to be simple relations. The merging technique to be described will handle more complex schemas. In particular, it will allow for an aggregation of attributes (i.e. a type) to become a new attribute which may be incorporated in a higher level aggregation. This approach combining hierarchic relationships with subtype relationships into one database model is due to Smith and Smith [5, 6] and Hammer and McLeod [7, 8]. However we shall use a somewhat different formalism.

A basic assumption is made that an object is purely defined by its attributes. Hence, two objects with the same attributes are actually identical. This assumption will enable a process of identification of similar structures in the two database schemas to take place. For example, if in both databases of the previous example the NAME attribute is composed of FIRST NAME and LAST NAME, then they would be considered identical. The underlying justification is that, had it been necessary to distinguish between them, an appropriate attribute would be available.

The following section formulates a database model that incorporates these ideas. The results are for the most part easy to derive, but describe what we believe to be desirable properties of any database abstraction technique.

2.1 Database Schema

Let P be a finite set of primitive attributes. Define P_∞ by induction as follows:

$$\begin{cases} P_0 = P \\ P_{n+1} = P_n \cup 2^{P_n} \end{cases}$$

A subset $S \subseteq P_\infty$ is hereditary if $s \in S \implies \forall t \in s: t \in S$.

Definition 1. A finite hereditary subset of P_∞ is a database schema (over the set of primitives P). Each element of a database schema is a database type.

The primitive attributes of the schema need not always be the actual atomic fields of the database (character strings, numbers, etc.). Rather, they are those that the user has determined as primitive for the purpose of constructing the merge. For instance, in the example of the introduction ADDRESS may be a "composite" object (consisting of CITY and STREET), while it is a primitive object for the purpose of the merge.

Definition 2. Let S be a database schema and $s, t \in S$ two database types. Define $t \text{ att } s$ (t is an attribute of s) if $t \in s$, $t \text{ gen } s$ (t is a generalization of s) if $t \not\subseteq s$.

The relations att and gen are embedded in the given schema. Note that, since gen requires both participants to be sets, only composite types can be generalizations.

Prop 1. the attribute and generalization relations maintain:

- (1) $s \text{ att } t, t \text{ gen } r \implies s \text{ att } r,$
- (2) $s \text{ gen } t, t \text{ gen } r \implies s \text{ gen } r.$

These properties will be referred to as the inheritance of attributes (over generalizations) and the transitivity of generalizations. Thus, since T6 is a generalization of both T4 and T5, both T4 and T5 inherit NAME and ADDRESS from T6. Throughout the formal definitions we shall use the term generalization in preference to subtype. These relationships are inverses of one another.

Definition 3. Let S be a database schema and $s, t \in S$ two database types. Define $s \wedge t$ (the meet of s and t) = $s \cap t$, $s \vee t$ (the join of s and t) = $s \cup t$.

The meet and join operators create new database types from existing composite types, by taking their intersection and union, respectively (the terms intersection and union are reserved for operators on database population).

Prop 2. Types created by the meet and join operators maintain:

- (1) $S \cup \{s \wedge t\}$ and $S \cup \{s \vee t\}$ are schemas
- (2) $s \wedge t \text{ gen } s$ and $s \wedge t \text{ gen } t$,
- (3) $s \text{ gen } s \vee t$ and $t \text{ gen } s \vee t$.

We have been using a graphic representation of database schema with its derived attribute and generalization relations. Each database type is represented by a node. If $t \text{ att } s$, there is a directed arc from node s to node t: $s \rightarrow t$. If $t \text{ gen } s$, there is a directed arrow from node s to node t: $s \Rightarrow t$ (an edge is either an arc or an arrow). However, if $t \text{ gen } r$ and $s \text{ att } t$ then $s \text{ att } r$ is suppressed in the graphic representation. Similarly, if $s \text{ gen } t$ and $t \text{ gen } r$ then $s \text{ gen } r$ is suppressed. The definition of a schema implies the following restrictions on the descriptive graph G: (1) G is acyclic, (2) G has no parallel arcs and no parallel arrows, (3) No two nodes of G have the same out-going neighbors, and (4) A node cannot have an arrow as its only out-going edge.

As an example of a schema, consider the primitive attributes

NAME, SSN, MAJOR, DEGREE, D_NAME, SCHOOL

and the composite types

DEPARTMENT = (D_NAME SCHOOL),

STUDENT = (NAME SSN MAJOR),

GRAD_STUDENT = (NAME SSN MAJOR DEGREE),

INSTRUCTOR = (NAME SSN (D_NAME SCHOOL))

Some attribute relationships embedded in this schema are

SCHOOL att DEPARTMENT,

NAME att STUDENT,

DEPARTMENT att INSTRUCTOR,

The only generalization relationship embedded in this schema is

STUDENT gen GRAD_STUDENT.

The type operators may now be employed to create new types. The meet of STUDENT and INSTRUCTOR is

PERSON = STUDENT \wedge INSTRUCTOR = (NAME SSN).

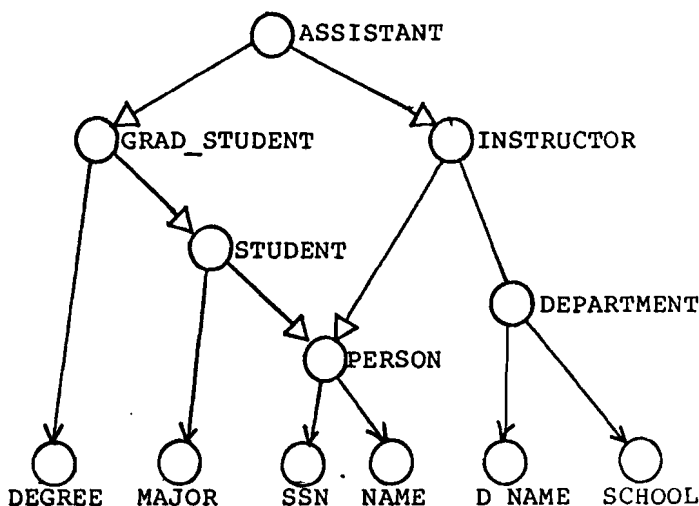
The new entity PERSON has attributes NAME and SSN, and is for either STUDENT or INSTRUCTOR. The join of GRAD_STUDENT and INSTRUCTOR is

ASSISTANT = GRAD_STUDENT \vee INSTRUCTOR = (SSN NAME MAJOR DEGREE (D_NAME SCHOOL))

The type ASSISTANT has the attributes of GRAD_STUDENT and INSTRUCTOR and therefore is a type being both GRAD_STUDENT and INSTRUCTOR. Some new relationships are

PERSON gen STUDENT,
INSTRUCTOR gen ASSISTANT.

The graphic representation is



2.2 Database Assignment

With this definition of a schema we can now define how to "populate" it with objects. Let S be a database schema (over P), X a finite set of objects (the "population" of the database) and $\theta : X \times S \rightarrow X$ a (partial) function. θ assigns objects as attribute values of other objects. Assume $\theta(x, s) = y$. The triplet (x, s, y) is called a possession. For each $x \in X$, the attributes applicable to x are $T_x = \{s \in S \mid \theta(x, s) \text{ is defined}\}$.

Definition 4. Let $s \in S$ be a type and $x \in X$ an object. Define

$x \text{ mem } s$ (x is a member of s) if $T_x \supseteq s$.

The domain of s is

$\text{dom}(s) = \{x \in X \mid x \text{ mem } s\}$.

Each object is a member of all the types with attributes applicable to it. Thus, if an object possesses values for the attributes SSN, NAME and SCHOOL, it is a member of types PERSON and STUDENT. Each type defines a domain, which is all the objects that are members of this type.

Prop 3. The domains of types maintain:

- (1) $x \text{ mem } s, t \text{ gen } s \implies x \text{ mem } t$,
- (2) $\text{dom}(s \wedge t) = \text{dom}(s) \cap \text{dom}(t)$,
- (3) $\text{dom}(s \vee t) = \text{dom}(s) \cup \text{dom}(t)$.

As an example of these properties, consider the previous schema. All members of STUDENT are also members of PERSON. Objects with both GRAD_STUDENT attributes and INSTRUCTOR attributes (i.e. objects in the intersection of these domains) comprise the domain of ASSISTANT. Objects with either STUDENT attributes or INSTRUCTOR attributes (i.e. objects in the union of these domains) comprise the domain of PERSON.

(Note that the definition of mem applies only to composite objects and types. This relation can be extended to primitive objects and types as follows: Let $s \in S$ be a primitive type and $x \in X$ a primitive object. $x \text{ mem } s$ if $\exists y \in X: \theta(y, s) = x$. i.e. a primitive object is a member of the types that assign it as an attribute value).

Definition 5. θ is called an assignment to S (over X) if

- (1) $\theta(x, s)$ is defined $\implies \exists t \in S: s \text{ att } t, x \text{ mem } t$ and $\theta(x, s) \text{ mem } s$,
- (2) $s, t \in S$ primitives $\implies \text{dom}(s) \cap \text{dom}(t) = \emptyset$.

The conditions on an assignment guarantee that each possession is between objects that are members of related types. In particular, each object belongs to at least one domain (a primitive object belongs to exactly one domain).

Prop 4. The above definitions imply

- (1) for every $t, s \in S$ such that $t \text{ att } s$ there is a total function $f_{st} : \text{dom}(s) \rightarrow \text{dom}(t)$ $f_{st}(x) = \theta(x, t)$,
- (2) for every $t, s \in S$ such that $t \text{ gen } s$: $\text{dom}(s) \subseteq \text{dom}(t)$

Thus, there is a function from the domain of all members of INSTRUCTOR into all POSITIONS, assigning each member of INSTRUCTOR exactly one POSITION.

For the purpose of merging two different databases it is necessary that each composite object is identifiable by a combination of primitive attributes, so that, when the two populations are consolidated, identical objects can be recognized as such. We must therefore identify a key relationship between types.

Definition 6. Let $s, t \in S$ be types.

$t \text{ key } s$ (t is a key of s) if $t \text{ att } s$ and f_{st} has inverse.

Thus, attribute t is a key to attribute s , if each object of t determines a unique object of s . If, for example, the function assigning each PERSON one SSN is such that different PERSONS have different SSNs, then SSN is a key attribute to PERSON.

Prop 5. The key relation maintains: $s \text{ key } t, t \text{ gen } r \implies s \text{ key } r$.

As defined, a key is a single identifying attribute. However, this definition can be easily extended to include keys consisting of more than one attribute. By composing keys each non-primitive object can be identified by a combination of primitive objects.

Definition 7. Let S be a database schema (over P) and θ a database assignment to the schema S (over X). $\langle \theta, S \rangle$ is a database if $\forall s \in S, \forall x, y \text{ mem } s: \forall t \in s, \theta(x, t) = \theta(y, t) \implies x = y$.

This requirement identifies all objects that possess the attributes of a given type with identical attribute values. In other words, the members of each type are distinguishable by at least one possession. Consequently, a member can be identified by its set of possessions. Thus, every type is guaranteed to have at least one key: the trivial key consisting of its complete attribute set.

According to the definition of gen, every containment between sets of attributes implies a generalization relationship between these two types. For example, $\text{SIZE} = (\text{HAT_SIZE SHOE_SIZE})$ is a generalization of $\text{CUSTOMER} = (\text{NAME SSN ADDRESS HAT_SIZE SHOE_SIZE})$. However, such undesirable situations cannot occur: since every type has a key, by Prop 5 every generalization relationship must be supported by a key (i.e. if $s \text{ gen } t$ then s and t share a common key).

Definition 8. A type with its domain is a database class. A naming function which assigns each class a unique name is assumed. Let s be the name of a class. The type and domain of this class will be denoted $\text{type}(s)$ and $\text{dom}(s)$, respectively. The definition of the database class s is therefore

$$s = \langle \text{type}(s), \text{dom}(s) \rangle.$$

Also, whenever $\text{type}(t)$ is an element of $\text{type}(s)$, the name t will replace $\text{type}(t)$ in the description of $\text{type}(s)$. Finally, many of the concepts defined for types (such as the attribute and generalization relations) are trivially extended to classes.

The database model described above can be summarized as follows: A database is a schema and an assignment. The schema defines the different types (created from a set of primitives), while the assignment determines the distribution of the objects into the domains of these types (thus creating classes). Four semantic relations (att, gen, key and mem) between elements of the database, as well as a set of database functions, are embedded in the database.

3 Merging Two Databases

We can now exploit this database model to produce a formal description of the merge of two databases. As a first step, we define three relationships between databases: consistency, containment and equivalence. For the sake of brevity, some of the preliminary definitions and results are only summarized. We then define the union of two consistent databases. Finally, a merging algorithm is introduced that produces this union.

3.1 Relations Between Databases

Given a schema S , we define two related schemas. The base schema S^+ is obtained by removing from S all types that are "pure" generalizations. The span schema S^* is obtained by augmenting S through successive applications of the meet operator to pairs of types (if the intersection of their attributes includes both keys) until no further types are created. If the base of schema S_1 is contained in the base of schema S_2 , S_2 is said to be richer than S_1 ($S_1 \triangleleft S_2$). If two schemas S_1 and S_2 have the same base they are said to be equivalent ($S_1 \sim S_2$).

Definition 9. Let S_1, S_2 be two schemas (over P). Assume θ_1 is an assignment to S_1 (over X_1) and θ_2 is an assignment to S_2 (over X_2). Let $X = X_1 \cup X_2$ and $S = S_1 \cup S_2$ and denote $D_1 = \langle \theta_1, S_1 \rangle$, $D_2 = \langle \theta_2, S_2 \rangle$.

D_1 and D_2 are consistent ($D_1 \mid D_2$) if $\forall x \in X, \forall s \in S: \theta_1(x, s) = \theta_2(x, s)$. Two databases are consistent, if they have no contradictory possessions.

D_1 is contained in D_2 ($D_1 \triangleleft D_2$) if $S_1 \triangleleft S_2$ and $\forall x \in X, \forall s \in S_1: \theta_1(x, s) = \theta_2(x, s)$. If one set of possessions is a subset of the other, the former database is contained in the latter.

D_1 and D_2 are equivalent ($D_1 \sim D_2$) if $S_1 \sim S_2$ and $\forall x \in X, \forall s \in S: \theta_1(x, s) = \theta_2(x, s)$. If the two databases have the same sets of possessions, they are equivalent. It follows that $D_1 \triangleleft D_2$ and $D_2 \triangleleft D_1$ imply $D_1 \sim D_2$. (Two partial functions are equal (=) if they have the same value whenever both are defined. If, in addition, both are defined on the same set, they are identical (\equiv)).

Theorem 1. Let $D = \langle \theta, S \rangle$ be a database and let $D^* = \langle \theta, S^* \rangle$. Then $D^* \sim D$. D^* will be called the canonical form of D .

Definition 10. Let $D_1 = \langle \theta_1, S_1 \rangle$, $D_2 = \langle \theta_2, S_2 \rangle$ be two consistent databases. The union of D_1 and D_2 is $D_1 \cup D_2 = \langle \theta, S \rangle$, where $S = (S_1 \cup S_2)^*$ and $\theta : X \times S \rightarrow X$

$$\theta(x, s) = \begin{cases} \theta_1(x, s), & \text{if } s \in S_1 \\ \theta_2(x, s), & \text{if } s \in S_2 \end{cases}$$

Theorem 2. $D_1 \cup D_2$ is the smallest canonical database that contains both D_1 and D_2 . i.e.

- (1) $D_1 \cup D_2$ is a database,
- (2) $D_1 \cup D_2 = (D_1 \cup D_2)^*$,
- (3) $D_1, D_2 \triangleleft D_1 \cup D_2$,
- (4) $D_1, D_2 \triangleleft D \implies D_1 \cup D_2 \triangleleft D$,
for every database D .

Every att, gen and mem relationship in D_1 or D_2 is preserved in D . As for the key relation, a class of D is keyed on the join of its key attributes in D_1 and D_2 .

3.2 A Merging Algorithm

The merge process takes place in two phases. First, the new schema is obtained, thus determining the new att and gen relations. Then, using the new schema, the population of the new database (the mem relation) and the identification of this population (the key relation) are derived.

In practice, the given databases are not detailed by a schema-assignment pair. Instead, each database is described by:

- (1) A set of type-names with two relations (att and gen),
- (2) For each type-name, a domain of objects (relation mem),
- (3) Between each two related domains, a function,
- (4) For each function, a predicate indicating whether the function has inverse (relation key).

Of course, all relations and functions should be derivable from a legal schema-assignment pair.

Given two such databases, their union is arrived at by the the following (informally described) algorithm:

(1) Combine the sets of primitive types (those not in the range of att or gen) into one set. This necessitates asserting which pairs of primitive types are identical. In graph notation, this means tying the two separate graphs together, by merging pairs of terminal nodes.

(2) Recursively combine composite types that have identical attributes. This is a "bottom-up" process. In graph notation, the result is a graph in which nodes with similar descendants are merged into one.

(3) Augment the schema with types created by applying the meet operator to every two types, if the result will contain both keys. In graph notation, a new node is introduced by every such application.

This new node is connected to the original nodes via arrows, and all outgoing nodes shared by the original nodes are transferred to the new node. If identical types have been introduced, they should be combined. The result is a new set of types with new att and gen relations. That is, a new graph.

The "connections" between the two original schemas are of three kinds. Assume $s1 \in S1$ and $s2 \in S2$. We may have
 (a) $s1$ and $s2$ merge into one type $s \in S$;
 (b) $s1$ and $s2$ produce a third type $s1 \wedge s2 \in S$;
 (c) $s1$ and $s2$ maintain in S : $s1$ gen $s2$.

(4) The keys of the new schema are determined as follows: In the above three cases the join of the key attributes of $s1$ and $s2$ is the key to either s (case a) or $s1 \wedge s2$ (case b) or $s1$ (case c). In each case the join is guaranteed to be part of this type. All other keys are carried over from the original schemas.

(5) The domains of the new schema are determined as follows: In the above three cases the union of the domains of $s1$ and $s2$ is the domain of either s (case a) or $s1 \wedge s2$ (case b) or $s1$ (case c). All other domains are carried over from the original schemas.

(6) Assuming the original databases are consistent, the functions of the new database are readily derived from the original functions.

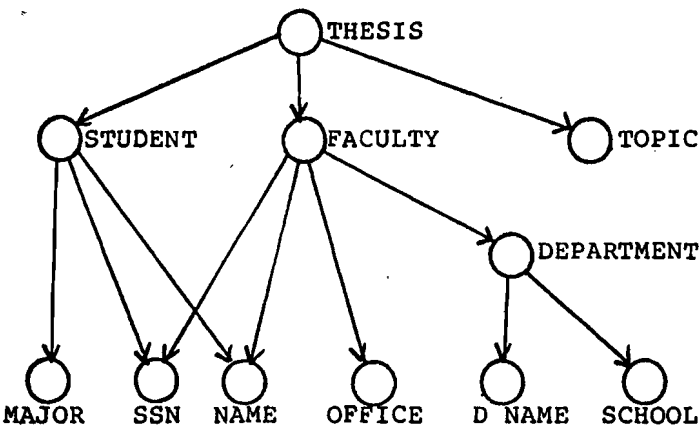
The above procedure determines the new schema and its keys, domains and functions. In other words, a new database.

As an example of a database merge consider the previous University Personnel database and the following Thesis File database with primitive attributes

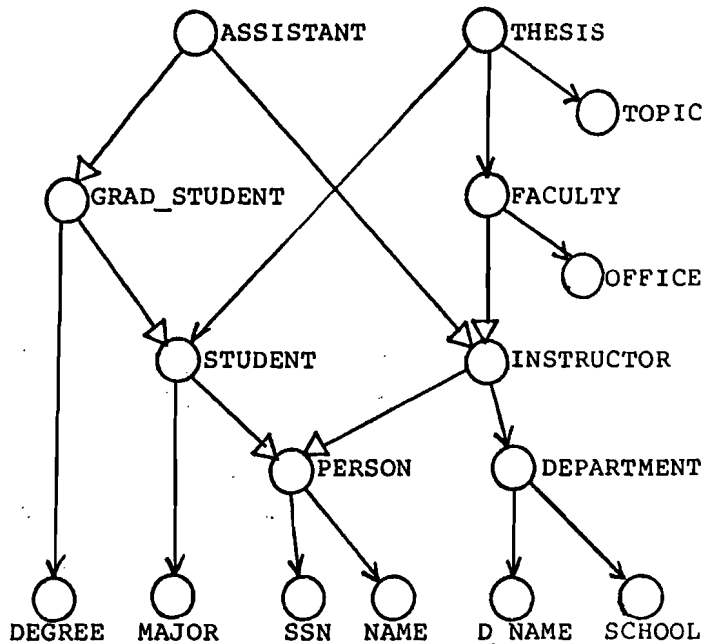
SSN, NAME, MAJOR, D_NAME, SCHOOL, OFFICE, TOPIC and composite types

DEPARTMENT = (D_NAME SCHOOL),
 STUDENT = (SSN NAME MAJOR),
 FACULTY = (SSN NAME (DEPARTMENT OFFICE))
 THESIS = (STUDENT FACULTY TOPIC).

The graphic representation of the schema is



Assume now that all primitives with the same name are similar. The merge of these databases will be



Having the same primitive attributes, the two types called DEPARTMENT were merged. The meet of FACULTY and STUDENT (in the second database) was identified with PERSON (of the first database), thus enabling a further merge of the two types called STUDENT. The type INSTRUCTOR (in the first database) was a subtype of PERSON (PERSON with DEPARTMENT). In the merge it was entered as a generalization of the type FACULTY (INSTRUCTOR with OFFICE).

3.3 Implementation

A program has been written, that accepts a description of two schemas of abstract databases, along with a set of assertions on similar types. It then generates a schema for the union database. Once the schema is available, and with the above rules for determining the domains, functions and keys, a query language and construction primitives can be used to obtain a physical representation of the merge database.

However, a physical representation is not needed in order to query the union schema. Information collected during the merge process is used by another program to enable querying of the union schema. The program decomposes each query to a set of queries, submits them to the component databases and recomposes the results to form an answer for the original query.

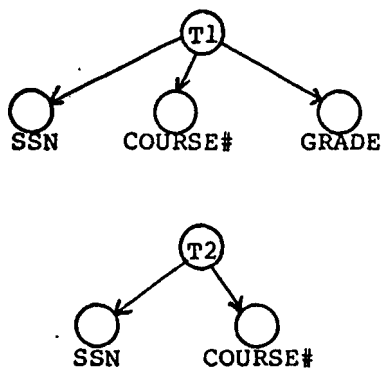
Together, the schema-merging program and the query-decomposition program provide

the end-user with a virtual database system.

Two databases may be merged, even though they are not completely consistent. An inconsistency is a multi-valued function. This may be allowed in the virtual database, as its resolution requires outside help. During the course of a query, if an inconsistency is detected, it is reported to the user.

4 Discussion

The object of this paper has been to formalize a semantic data model and to exploit it in the problem of merging two databases. In particular we have described a general method of producing the merge and defining the consequent integrity constraints on the data. The method appears to work well when the initial databases are properly differentiated by their semantic structure. However, this is not always the case: an important second phase of this research is to construct a program through which the user may "repair" the semantics of the initial databases at the same time that the merge is constructed. As an example, consider



The method described will produce a database in which T2 is a generalization of T1. This would be the correct merge if T2 stood for enrollments and T1 stood for completed enrollments (i.e. enrollment for which a grade has been assigned). However, it would be totally incorrect in the case that T1 stood for enrollments but T2 indicated teaching assignments (i.e. relationships of teaching between persons and courses). What has happened in this case is that our initial assumption that an object is entirely defined by its attributes does not hold. A program that attempts to merge T1 and T2 must be corrected by disambiguating these attributes. By a back-tracking process, the two classes of SSN must be separated by constructing STUDENT and INSTRUCTOR subtypes. Once this has been done, the merge can proceed correctly.

Another problem with the merging process as we have described it is that it operates only on the subtype structure. There are certain cases in which it is desirable to modify the attribute structure during the merge. Again, this can be performed by user interaction.

The problem of merging databases has been explored through the use of a specific approach to database semantics. It is possible that similar results could be derived for the recent semantic "extensions" of the relational model [9, 10]. These have yet to be investigated.

Among the commonly used types of database management systems, the most awkward merging problems are usually created by CODASYL systems. The model we propose bears some relationship to the CODASYL model: primitive attributes correspond to CODASYL fields and composite attributes correspond to CODASYL sets. However there is no way that we may specify a generalization relationship in CODASYL. When, as frequently happens, a generalization relationship has to be represented, there are two methods available to the CODASYL designer. One is to create a record class for the most general type and have fields for all attributes of all subtypes of this type. In addition, a boolean field must be added for each subtype indicating whether or not it is defined. The other method is to create a set for each subtype whose records contain the additional attributes for that type. Such sets must be constrained to have 0 or 1 member records. The choice of method is purely one of efficiency (in both space and time).

Given that a set of standardized application programs could be developed both to generate the appropriate subtype structures in CODASYL and to provide a uniform access method for them, we see no difficulty in applying the techniques developed in this paper in an operational CODASYL environment.

References

- [1] L.E.Erickson, M.E.Soleglad and S.L.Westermark; ADAPT: Uniform Data Language (UDL)- A Final Specification; Report 76-C-0899-1, Logicon Inc., San Diego CA, January 1978.
- [2] D.Shipman; The Functional Data Model and the Data Language DAPLEX; ACM Transactions on Database Systems, to appear.
- [3] G.G.Hendrix, E.D.Sacerdoti, D.Segalowicz and J.Slocum; Developing a

Natural Language Interface to Complex Data; ACM Transactions on Database Systems, Vol.3, No.2, June 1978.

- [4] D.McLeod and D.Heimbigner; A Federated Architecture for Database Systems; Proceedings of AFIPS National Computer Conference, Anaheim CA, 1980.
- [5] J.M.Smith and D.C.P.Smith; Database Abstractions: Aggregation; Communications of the ACM, Vol.20, No.6, June 1977.
- [6] J.M.Smith and D.C.P.Smith; Database Abstractions: Aggregation and Generalization; ACM Transactions on Database Systems, Vol.2, No.2, June 1977.
- [7] M.Hammer and D.McLeod; The Semantic Data Model: A Modelling Mechanism for Database Applications; Proceedings of ACM SIGMOD International Conference on the Management of Data, Austin TX, 1978.
- [8] M.Hammer and D.McLeod; SDM: A Semantic Database Model; Computer Science Technical Report TR 80-3, University of Southern California, Los Angeles CA, 1980.
- [9] E.F.Codd; Extending the Database Relational Model to Capture More Meaning; ACM Transactions on Database Systems, Vol.4, No.4, December 1979.
- [10] S.A.Borkin; Equivalence Properties of Semantic Data Models for Database Systems; PhD Dissertation, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge Ma, January 1979.