An Architecture for Automatic Relational Database System Conversion

BEN SHNEIDERMAN
University of Maryland
and
GLENN THOMAS
Kent State University

Changes in requirements for database systems necessitate schema restructuring, database translation, and application or query program conversion. An alternative to the lengthy manual revision process is proposed by offering a set of 15 transformations keyed to the relational model of data and the relational algebra. Motivations, examples, and detailed descriptions are provided.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design—*schema and subschema*; H.2.3 [**Database Management**]: Languages—*data manipulation languages (DML)*; H.2.5 [**Database Management**]: Heterogeneous Databases—*program translation*

General Terms: Design, Languages

Additional Key Words and Phrases: Database systems, automatic conversion, relational model, transformations

1. INTRODUCTION

As logical data requirements change, database administrators are faced with the enormous challenge of database system conversion. The new requirements may call for a simple addition/deletion of a relation or attribute or for a total restructuring of the logical relationships. Simple additions or deletions present a minor problem which can be handled easily, while a total restructuring may require complete rewriting of the database system from scratch [2].

Often the change in requirements is between these extremes, and it may be accommodated by a modest revision of the working system. The change may be to replace a one-to-one relationship with a one-to-many relationship, to add a field to the set of key fields, to decompose a complex record into several simpler records, or to partition a collection of records into two types based on the value of one field.

This work was supported in part by National Science Foundation Grant MCS-77-22509 and the University of Maryland Computer Science Center provided some of the computer resources.

Authors' addresses: B. Shneiderman, Department of Computer Science, University of Maryland, College Park, MD 20742; G. Thomas, Administrative Sciences, Kent State University, Kent, OH 44242.

Revising the schema is a relatively simple task which can be accomplished in a few hours or days. Translating the stored database to match the new requirements may be done by a program which creates the target database while verifying integrity constraints. The third task, rewriting the application programs or queries, is often the major bottleneck in system conversion.

In a health insurance application, it was estimated that two person-months per program were required to convert and test each of the 600 PL/I-IMS application programs which had an average length of more than 1000 statements. Even with compact high-level query languages, it may take months of effort to convert the hundreds of programs and test them to make sure that they still work correctly on the translated database. The burden is greatest with complex, lengthy host-embedded data manipulation languages, but it is still considerable with high-level self-contained languages such as those proposed for the relational model.

2. RESEARCH BACKGROUND

Until now our research on automatic conversion has been based on a specially designed data model which was tailored to meet the needs of automatic database system conversion. Our Pure Data Definition and Manipulation Languages blended elements of the network and relational models to facilitate the design of a Pure Transformation Language processor [8, 10, 16]. The eighteen transformations which we proposed permitted name changes, addition and deletion of fields, records and sets, changes to set keys, and the movement of fields between owner and member record types. The **FIND**, **STORE**, and **MODIFY** data manipulation statements provided explicit descriptions of query semantics through the use of path expressions with Boolean qualifications [9].

In this paper we explore the possibility of developing an automatic database system conversion facility for the relational model [1] using the relational algebra as a data manipulation language. Su and Reynolds [13, 15] suggest some relationally oriented transformations and show how SEQUEL queries might be rewritten in certain cases. Sakai [7] informally proposes several relational transformations as aids in the schema design process.

In related work with other data models, Housel [3] demonstrates the use of CONVERT [11, 12] operators for querying and transforming hierarchically organized, tabular databases described by DEFINE language statements. Su and Lam [14] describe transformations of data traversals and operations in a high-level semantic data model. Navathe [5] offers a high-level data model defined by "schema diagrams" and a set of useful schema diagram transformations. Jacobs [4] describes automatic conversion in the context of his database logic, which provides a formal mathematical foundation for database systems.

Our contribution in this paper is the gross architecture for a complete conversion system. We build on the individual suggestions for schema transformations by describing the implementation issues and integrity constraints for a set of transformations on the schema, the stored database, and the application programs of a relational database system. We believe that our set of transformations, when used individually and in groups, can effectively support a database administrator in coping with changing user requirements.

3. BASIC MODEL

In this section we present the basic concepts of the transformation process and a taxonomy of transformations.

3.1 Transformation Process

We consider a database system (DBS) to consist of a schema S, a stored database D, and a collection of programs

$$P = p_1, p_2, \ldots, p_n$$

so that

$$\text{DBS} = \langle S, D, P \rangle.$$

Applying a program to the stored database yields an output

$$p_i(D) = o_i, \qquad \text{for} \quad i = 1, \ldots, n.$$

A transformation applied to a database system yields a new database system

$$T(\text{DBS}) = \text{DBS}' = \langle S', D', P' \rangle,$$

such that applying the new programs to the new database yields the output o_i'

$$p_i'(D') = o_i', \qquad \text{for} \quad i = 1, \ldots, n.$$

If o_i' is identical to o_i for all i, the transformation maintains *input/output equivalence*, since the execution yields identical results (although ordering of tuples or attributes may vary) for the programs. Of course, input/output equivalence is possible only for *information preserving* transformations; that is, transformations that do not destroy information but merely change the logical data format. For other transformations the database administrator will have to judge the acceptability of the output.

A weaker requirement might be for *attribute subset equivalence*, in which the target output attributes are a subset of the source output attributes. If a transformation eliminates attributes from the database, then a seven-column report may be acceptable in place of the ten-column report generated by the original program. In general, a database administrator may seek still weaker requirements and define *acceptable equivalence* to suit organizational needs.

3.2 Taxonomy of Transformations

We have found it useful to classify transformations on three features: information preservation, data dependence, and program dependence. These are features of a transformation type rather than a specific instance of a transformation. The deletion of an attribute from a relation is classified as not information preserving, even though there are instances when the deletion may result in no loss of information (if all values are nulls or the attribute appears twice).

A transformation is *information preserving* if no information is lost. If a transformation is information preserving, then it is also *invertible* (the original format can be restored), but only immediately after the transformation is performed. After operations such as storing, modifying, or deleting a tuple, invertibility is no longer guaranteed. For example, adding attributes or relations to a database system is information preserving, but deleting attributes or relations is not information preserving.

A second feature for characterizing transformations is *data dependence/independence.* A transformation is data dependent if the stored database must be checked to determine whether the transformation is consistent with the logical format of the target system. For example, eliminating an attribute from the key may not be permissible if the remaining key attributes no longer guarantee uniqueness. If, while checking the stored database, the conversion system encounters an inconsistency between the source and target databases the database administrator must decide on how to modify/delete the source database so that the conversion can continue. We prohibit the creation of a target database which does not adhere to the target schema. This checking process may be extremely costly, but it must be done if data integrity is to be ensured. Efficient techniques, such as scanning summary tables, data dictionaries, or indexes, will have to be implemented to reduce checking costs.

A third feature for characterizing transformations is *program dependence/independence.* A transformation is program dependent if the application programs must be checked to determine whether the transformation is permissible. For example, eliminating an attribute that is the target of a query or is used in a Boolean qualification destroys the possibility of an input/output equivalent transformation. The database administrator must be informed of this condition so that he or she can decide whether the resultant transformation is acceptable. We have found it useful to distinguish between two forms of program dependence: query program dependence and update program dependence. Similar distinctions have been made by researchers in multiple view maintenance.

Note that a transformation may be program independent but still require program modifications. For example, changing a relation name is program independent (assuming the new name is not already in use) even though the programs must be modified. If a transformation is program dependent, this implies that the collection of application programs must be checked to determine whether the transformation is permissible. Since this is a potentially costly process, efficient techniques for scanning on-line libraries of programs or summary information will be useful.

Figure 1 shows how our fifteen proposed transformations fit this taxonomy. Section 4 presents an extended example of how our transformations might be used in a practical situation. Section 5 describes an architecture for a conversion system, and Section 6 provides more details about each transformation.

4. TRANSFORMATION EXAMPLE

We begin with an extremely simple situation and follow the impact of organizational policy changes on the database system. At every stage the database will be kept in fourth normal form. Imagine a chemical plant that begins using a database system by maintaining basic employee data in the relation **emp**:

emp (eno*, name, salary, dept, job, cartag)

which has an employee number (**eno**) as a key (key fields will be marked with an asterisk), the employee's **name**, the employee's **salary**, the department he or she

INFORMATION PRESERVING
(IMMEDIATELY INVERTIBLE)

	PROGRAM INDEPENDENT	QUERY PROGRAM INDEPENDENT UPDATE PROGRAM DEPENDENT	PROGRAM DEPENDENT
DATA INDEPENDENT	CHANGE NAME ADD ATTRIBUTES INTRODUCE	DECOMPOSE INFO REMOVE MULTIVALUED DEPENDENCY	
DATA DEPENDENT	PROMOTE TO KEY DEMOTE FROM KEY	PARTITION INTO EXPORT PARTIAL DEPENDENCY EXTRACT TRANSITIVE DEPENDENCY	

NOT INFORMATION PRESERVING
(NOT IMMEDIATELY INVERTIBLE)

	PROGRAM INDEPENDENT	QUERY PROGRAM INDEPENDENT UPDATE PROGRAM DEPENDENT	PROGRAM DEPENDENT
DATA INDEPENDENT			DELETE ATTRIBUTES SEPARATE
DATA DEPENDENT		COMPOSE FROM IMPORT DEPENDENCY	MERGE FROM

Fig. 1. Taxonomy of transformations.

works in (**dept**), the title of his or her **job**, and the tag number for his or her car (**cartag**) in the parking lot.

After some time, management recognizes that the database system can be used to keep the health records of each employee as well. Adding attributes for **weight**, **lung** capacity, blood pressure (**bp**), and the month (**mo**) and year (**yr**) of the most recent physical exam can be accomplished by issuing a transformation statement:

ADD ATTRIBUTES weight, lung, bp, mo, yr
To emp.

to yield a new relation:

emp (eno*, name, salary, dept, job, cartag, weight, lung, bp, mo, yr)

This transformation is simple to implement and is supported by some currently available systems. The transformation is information preserving, data independent (the database does not have to be checked to verify the allowability of this transformation), and program independent (the programs do not have to be checked to verify the allowability of this transformation). Immediately after the transformation the new attributes have null values which the user can replace with actual values for each employee.

Privacy concerns may prompt the database administrator to decouple the medical information from the accounting or personnel information. This decomposition separates the concerns of different users so each can deal with a single simpler relation. This transformation may be specified as

```
DECOMPOSE emp INTO
  emp-acct (eno*, name, salary, dept, job, cartag),
  emp-med (eno*, weight, lung, bp, mo, yr).
```

yielding two relations. This is another simple information preserving, data independent, and query program independent transformation. However, this transformation requires translation of the database and revision of the programs. Queries directed to the **emp** relation now may require a join to retrieve or qualify attributes. Updates to the **emp** relation may now require updates to one or both new relations.

Instead of maintaining only the most recent medical examination data, management may decide to maintain a history of yearly examinations. This requires that the **yr** attribute be included as a key attribute in the medical relation

```
PROMOTE yr IN emp-med TO KEY.
```

Since the **eno** provided a unique key, adding an attribute **yr** to the key cannot be a problem. The stored database and the programs need not be altered, but future updates will have to satisfy the new integrity constraint about the key of the **emp-med** relation. Once multiple tuples have been added for an employee, queries that formerly returned a single tuple may now return several tuples. User programs may have to be modified if only the most recent medical report is desired.

Now imagine that a new government safety regulation requires that workers coming into direct contact with hazardous substances must have a monthly lung capacity exam and complete historical records must be kept. To satisfy this new requirement, the high-risk employees (those with **dept** = 'production') must be separated out from the low-risk employees (all other departments). The **PARTITION** transformation can be used in this simple way to form a mutually exclusive partition of tuples in a relation:

```
PARTITION emp-med INTO
  (emp-acct.eno = emp-med.eno AND emp-acct.job = 'production'):
           high-emp-med (eno*, yr*, mo, weight, lung, bp);
  ELSE     low-emp-med (eno*, yr*, mo, weight, lung, bp).
```

The number of tuples in **emp-med** will be equal to the sum of the number of tuples in **high-emp-med** and **low-emp-med**.

To record the monthly lung capacity information and keep historical records requires that the month participate in the key

PROMOTE mo IN high-emp-med TO KEY.

so that the relations now are

emp-acct (eno*, name, salary, dept, job, cartag)
high-emp-med (eno*, yr*, mo*, weight, lung, bp)
low-emp-med (eno*, yr*, mo, weight, lung, bp)

This introduces a partial dependency of **weight** and **bp** on **eno**, **yr** in **high-emp-med**, because only **lung** capacity is measured monthly. Eliminating this partial dependency so as to restore the fourth-normal-form status requires the use of the **EXPORT** transformation

EXPORT PARTIAL DEPENDENCY FROM high-emp-med
GIVING high-annal (eno*, yr*, weight, bp).

After issuing a trasformation to change names

CHANGE NAME FROM high-emp-med TO high-monthly.

we now have the following set of relations:

emp-acct (eno*, name, salary, dept, job, cartag)
high-annual (eno*, yr*, weight, bp)
high-monthly (eno*, yr*, mo*, lung)
low-emp-med (eno*, yr*, mo, weight, lung, bp)

A further change in policy may require that all employees with the same **job** title earn the same **salary**. This means that a transitive dependency now exists among two nonkey fields of **emp-acct**. To **EXTRACT** this transitive dependency and to create a table of **job** titles with a **salary** attribute we could issue the following transformation:

EXTRACT TRANSITIVE DEPENDENCY FROM emp-acct
GIVING job-salaries (job*, salary).

This leaves the **emp-acct** relation without the **salary** attribute, so queries on the **salary** attribute now require a join. This transformation is information preserving and program independent, but it is data dependent since the database must be checked to see if there is no conflicting information.

Finally, the original assumption that employees brought only a single car to work must be changed to accommodate multicar families. This new assumption implies a multivalued dependency between **eno** and **cartag** in the **emp-acct** relation. The following transformation restores fourth-normal-form status:

REMOVE MULTIVALUED DEPENDENCY FROM emp-acct
GIVING car-registration (eno*, cartag).

The final set of relations is

emp-acct (eno*, name, dept, job)
high-annual (eno*, yr*, weight, bp)
high-monthly (eno*, yr*, mo*, lung)
low-emp-med (eno*, yr*, mo, weight, lung, bp)
job-salaries (job*, salary)
car-registration (eno*, cartag*)

Throughout this example the focus has been on changes to the schema as described by the set of relations. For each transformation, there are precise rules for translating the database and revising the programs.

5. SYSTEM ARCHITECTURE

In the simplest case (Figure 2), we assume that the source schema, stored database, and application programs exist in available on-line libraries. The conversion processor reads transformation statements, checks syntax, matches variable names with those in the desired source schema, and prints error messages when there are problems. If all is well, then a target schema can be generated by the system.

The second stage deals with data-dependent transformations which require checking the source stored database to ensure that the constraints of the transformation are satisfied. If attribute values produce constraint violations, error messages are printed and the database administrator must decide about changing or eliminating these values. If such modifications are not possible, then this transformation cannot be allowed to go forward.

Assuming all constraints are satisfied, then the target database can be generated. In the simple case, the source stored database can be quiesced and the target stored database can be generated without concern for concurrent access. If continuous access is required, then more elaborate algorithms and protection mechanisms are necessary.

The third stage handles program-dependent transformations which require checking the source application programs to determine whether the transformation is permissible. Here again the database administrator must decide what to do in cases where an input/output equivalent transformation is not possible. The database administrator must decide whether a transformation is acceptably equivalent or whether a manual application program revision is required. After responding to system messages and warnings, the database administrator can allow the generation of the target application programs.

When all three database system components have been generated, there will probably have to be a testing phase to ensure that the target system functions properly before the old system is retired. In some cases both systems will remain available and in other cases the target system will be copied for use at another site while the source system remains active.

Many variants of this conversion process can be envisioned. The source application programs may not exist in an on-line library; instead, they may be typed in at a terminal when needed. In this case the transformation commands must be maintained and a real time conversion can be made to produce the proper query or update operation. Dynamic transformation can be used to

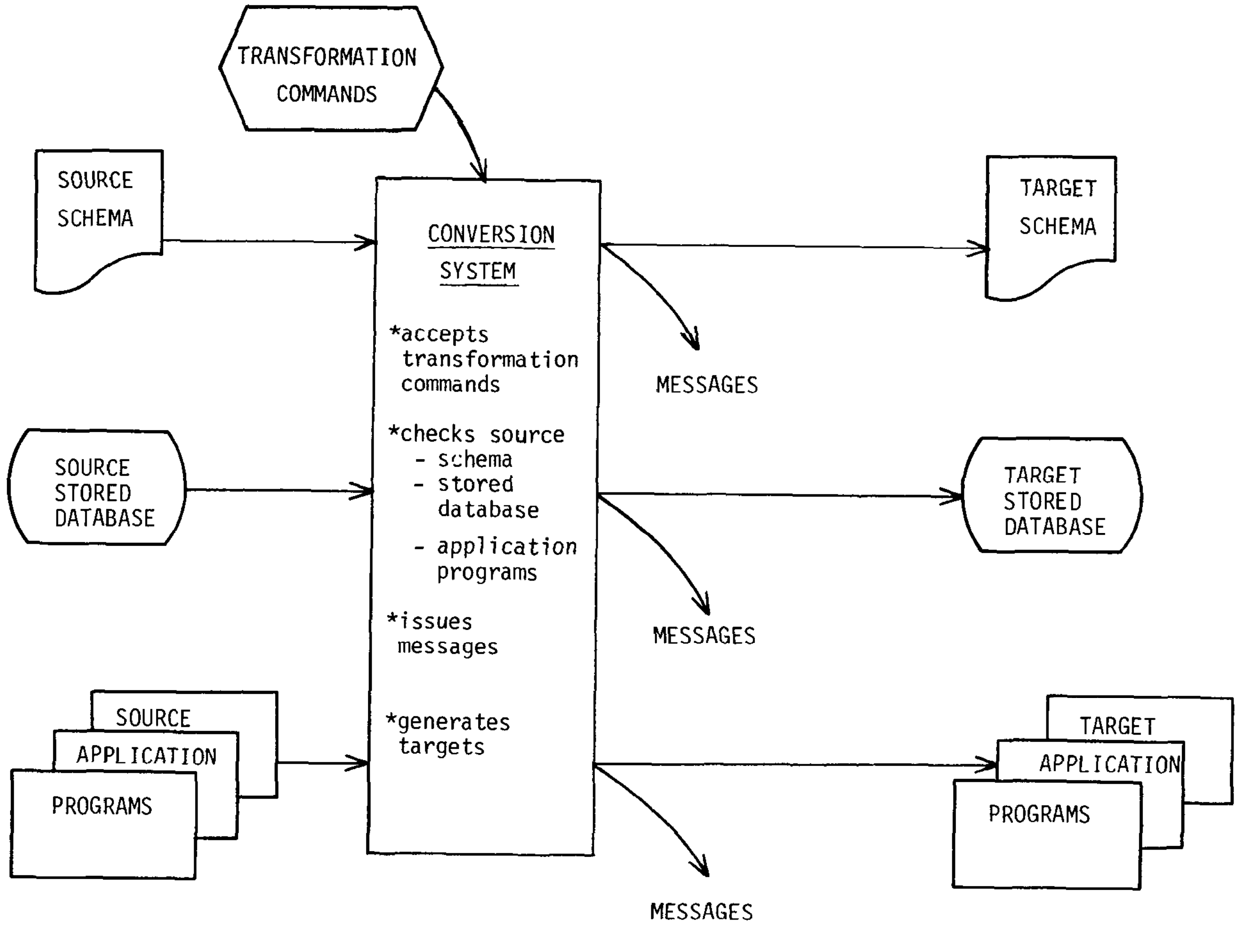

Fig. 2. Architecture for database system conversion.

support multiple views of a set of base relations [18]. Another variant would be to assume a distributed database environment where each site could maintain its own format for the database. In this case dynamic transformation of the database would be necessary to present data in the required format.

If a sequence of transformation commands is issued, then there are some interesting optimization questions. In some cases the simple approach of processing each transformation command independently may be appropriate and efficient, but it may sometimes be more effective to pass units of data or individual application programs through a sequence of commands, thereby skipping intermediate stages. Optimization strategies may be tailored to the particular sequence of transformation commands. For automatic database system conversion to become practical, implementers will have to cope with some of these efficiency questions.

6. TRANSFORMATION DEFINITIONS

The following subsections present fifteen relational transformations with the constraints that the source database system must satisfy to guarantee input/output equivalence. The target database and the revisions to the relational algebra application programs are described. Special attention is given to transformation side effects requiring database administrator intervention to obtain a target database system that is acceptably equivalent to the source database system. A more formal definition of these transformations is given in [16].

We use the notation

$$R(k_1^*, k_2^*, \ldots, k_n^*, f_1, f_2, \ldots, f_m)$$

to indicate that a relation R has n key attributes $k (n \geq 1)$ and m nonkey attributes $f (m \geq 0)$. Where there is no ambiguity,

$$R(K^*, F)$$

indicates the relation R where K is a grouping of one or more key attributes and F a possibly empty grouping of nonkey attributes. The relational algebra projection of R over a collection a, b, c, ... of its attributes is denoted by

$$R[a, b, c, \ldots].$$

The & join of two relations R and S over the compatible attribute groupings A and B is given by

$$R[A \,\&\, B]S.$$

Finally, the restriction of R to a subset of its tuples based on a Boolean condition is denoted by

$$R[a \,\partial\, v1 \quad \$ \quad b \,\partial\, v2 \quad \$ \ldots]$$

where a, b, ... are attributes of R; v are constants or identifiers; ∂ is one of =, <>, >, >=, <, or <=; and $ is **AND** or **OR**.

6.1 Elementary Transformations

Changing the name of an identifier and adding/deleting a relation or an attribute of an existing relation are elementary transformations that are available on some commerically distributed systems. These transformations are included for completeness and are not the focus of this paper.

Changing an identifier (relation or attribute) requires that the schema and the application programs be revised, but the stored database is untouched. The **CHANGE NAME** transformation accomplishes this task.

Adding an attribute to an existing relation (**ADD ATTRIBUTES** transformation) requires a modification to the schema and some way of indicating that the stored database contains a new attribute with all null values. An application program may be written to populate the attribute with values. Source application programs still function correctly, but the database administrator may wish to manually rewrite update operations to provide values for the expanded relation.

Deleting an attribute from an existing relation (**DELETE ATTRIBUTES** transformation) requires a schema modification, elimination of the values from the database, and changes to the application programs. Any transformation that destroys information, as does the **DELETE** transformation, cannot be guaranteed to produce input/output equivalence. If it happens that the application programs do not refer to the deleted attribute, then it will be possible to produce input/output equivalence.

Introducing a new relation (**INTRODUCE** transformation) is a basic operation which requires an addition to the schema, but no change to the stored database or the application programs. Eliminating an existing relation (**SEPARATE** transformation) from the database system requires deletion from the schema and the stored database. Since information is lost in the **SEPARATE** transformation, application programs which reference the separated relation can no longer function correctly.

6.2 Promoting/Demoting Keys

Assume that the relation R is defined by

$$R(k_1^*, k_2^*, \ldots, k_n^*, f_1, f_2, \ldots, f_m),$$

where the n ($\geq$1) attributes k_i are key and the m ($\geq$0) attributes f_i are nonkey. A basic integrity assertion of the relational model is that there may never be two or more tuples of a relation that have the same value for the key attributes. Otherwise, it would be possible to have two tuples describing the same entity with possibly different and hence conflicting values for the nonkey attributes. When designing a DBS, the DBA (database administrator) must designate one or more attributes of each relation as the key. While the choice made may be appropriate at one time, changes in requirements may cause a redefinition of the key to include more or fewer attributes. The two transformations

PROMOTE "attribute name" **IN** "relation name" **TO KEY**

and

DEMOTE "attribute name" **IN** "relation name" **FROM KEY.**

allow such redefinition.

With **PROMOTE** the DBA may add an existing attribute to the key of a source relation. **DEMOTE** allows the removal of an attribute from the key of an existing relation. Both of these are perceived as being information preserving transformations because all relationships implicit in the source schema S can be derived in the target schema S'. **PROMOTE** is data dependent because, although adding a new attribute to an existing key cannot violate the uniqueness constraint, the new attribute may contain nulls that are not permitted for key attributes. **DEMOTE** requires an examination of the source stored database D to determine whether or not the resulting shortened key will be sufficient to uniquely identify the tuples. Should any two or more tuples share the same value for the target key, the DBA must interact with the transformation system to obtain a target stored database D' that will not violate the uniqueness constraint. With both the hierarchical and network models, the value of a record key affects its logical position in the stored database. The relational model makes no use of keys beyond enforcing the uniqueness constraint above. For this reason programs accessing and/or updating the stored database are unaware of the designation of fields as key or nonkey. Hence both **PROMOTE** and **DEMOTE** are program independent, yielding a target set of programs that is identical to the source programs P. Insert commands which do not have values for all key attributes will fail, so at execution time some inserts which succeeded before may now fail, but the program needs no revision. **PROMOTE** may be inverted by **DEMOTE**, and **DEMOTE** may be inverted by **PROMOTE**.

6.3 Decomposing/Composing Relations

A particular schema S is a model of some environment. The entity types of interest may be represented by one or several relations having key attributes to identify instances of the entities and nonkey attributes to retain data about the instances. For a particular entity there may be several valid but different representations. For example, consider an entity E that may be uniquely identified by the values of the grouping K of attributes. It may be desirable to store data describing occurrences of this entity as n mutually disjoint groupings of attributes F_i. Two possible representations of E are

(a) $R(K^*, F_1, F_2, \ldots, F_n)$

and

(b) $R_1(K^*, F_1)$
$R_2(K^*, F_2)$
$\vdots$
$R_n(K^*, F_n)$.

Representation (a) implies that all data about an occurrence of the entity will be stored and accessed as a single tuple of the relation R. Should information about some subset F_i of the attributes be desired, this can be obtained by the projection $R[K, F_i]$. Specific user views may similarly be defined by projection over the desired attributes. Representation (b) stores data about an entity as n tuples—one for each relation R_i. All data about the entity may be retrieved by the equijoin of the n relations R_i on the common key attributes K. Hence it may be argued that both representations are equivalent. However, we feel they differ in two respects.

The first difference is at the logical level. With representation (b), the data for an instance of the entity E is scattered across n tuples. In fact, it is possible that for some entity there may be $m(<n)$ tuples in the database. Hence a query for this instance might result in values for some attribute F_i and in the statement that there is no such entity for other attributes $F_j (i \neq j)$. Under representation (a) there would be one tuple for this instance. While the same values might be presented for a query on F_i, the query on F_j would presumably produce the answer "null" or "unknown". This semantic difference may be overcome by imposing the integrity constraint

$$\exists v(\forall r_i \in R_i, r_i[K] = \{v\} \Rightarrow \forall h \ (1 \leq h \leq n, i \neq h) \exists r_h \in R_h \ni r_h[K] = \{v\})$$

on representation (b) requiring that for each instance of E represented in any relation R_i, the entity must be represented in all relations R_j. If the underlying database management system enforces this integrity constraint, then the two representations are indeed equivalent. Otherwise they differ at the logical level because (b) represents components of the entity E by n separate and independent entities.

The second difference between representations (a) and (b) occurs at the physical level. With most database management systems, the data values composing a tuple are stored in physically contiguous locations. Hence with representation (a), should some attribute F_i be of extreme importance or require a high level of security, the entire relation will be physically stored or protected based on the attributes F_i. This may be wasteful of high-speed storage or impose needless security on some attributes F_j. Further, multiple projections may be required to materialize user views resulting in excessive CPU and channel use. Representation (b) may be preferable because it allows selecting storage devices and protection levels based on the importance of the individual attribute groupings F_i while more closely modeling diverse user views. Thus the DBA may prefer representation (b) for performance reasons, even though this might result in violation of the above integrity constraint.

The transformations **DECOMPOSE** and **COMPOSE** [7] allow the transformation of a DBS employing representation (a) to one employing representation (b), or vice versa. Whenever such a transformation is made, the DBA must be cognizant of the subtle logical differences between these representations. Especially with representation (b), a policy decision is required to determine whether or not the n relations R_i are independent or dependent.

If they are dependent, this requires users to be aware of the dependence whenever the stored database is updated by **INSERT**, **MODIFY**, or **DELETE** commands. The syntax of these transformations is

DECOMPOSE "relation name" **INTO** "relation definition list".

and

COMPOSE "relation definition" **FROM** "relation name list".

where **DECOMPOSE** maps representation (a) to (b), and **COMPOSE** maps (b) to (a).

DECOMPOSE is an information-preserving, data-independent, and program-independent transformation, while **COMPOSE** is not information preserving, is data dependent, and is program independent.

DECOMPOSE maps representation (a) to representation (b). Each of the n target relations must have the same key K as the source relation R. The n attribute groupings F_i must be pairwise disjoint. The target schema S' is derived from S by replacing the source relation R by the n relations R_i. The target database D' is obtained by executing the program

```
$R_1$        $\leftarrow R[K, F_1]$.
$R_2$        $\leftarrow R[K, F_2]$.
  .
  .
  .
$R_n$        $\leftarrow R[K, F_n]$.
DELETE $R$.
```

An examination of the program for populating the target database shows that

$$R = R_1[K = K]R_2[K = K] \cdots R_{n-1}[K = K]R_n$$

holds at the instant of the transformation. This leads to the observation that all source program queries of the form

$$R(\ldots),$$

where $(\ldots)$ is some form of projection, join, or restriction, may be converted to an input/output equivalent form by substituting the above equijoin for the reference to R. Because of the semantic difference between representations (a) and (b), source program updates of the form **INSERT** R, **MODIFY** R, and **DELETE** R cannot be transformed in such a clean manner. If the n target relations are dependent, then source statements of the form **INSERT** R may be automatically transformed to a sequence of target statements

```
INSERT $R_1$.
INSERT $R_2$.
  .
  .
  .
INSERT $R_n$.
```

with a control structure that guarantees that all succeed or none succeeds. Similar modifications may be made for **MODIFY** and **DELETE**. However, this implies that all programs yet to be written be aware of the dependency and be written in a manner that ensures the dependency will be maintained. If the n target relations are independent, then all source program statements that update the relation R must be deleted to create the target set of programs P'. Thus input/output equivalence of queries can be guaranteed, but the DBA must decide how to transform updates. **DECOMPOSE** may be inverted by **COMPOSE**.

COMPOSE maps representation (b) to representation (a). In addition to the constraints that the target relations and source relation share the same key and that the attribute groupings F_i be pairwise disjoint, **COMPOSE** requires that the n relations R_i be dependent. This implies that the source database must be examined to determine whether or not

$$R_1[K] = R_2[K] = \cdots = R_n[K]$$

holds. If it does, then the target database may be populated by executing the program

$$R \leftarrow R_1[K = K]R_2[K = K] \ldots R_{n-1}[K = K]R_n$$

and deleting the tuples of the relations R_i for $i = 1, n$. If the examination of the source database shows a violation of the above, then DBA interaction is required to resolve the violation before the transformation can be allowed. Hence **COMPOSE** is data dependent. At the schema level, the source schema relations R_i are replaced by the target relation R. This is not information preserving, because the n relations R_i lose their separate identity as a result of the transformation. All source program queries of the form

$$R_i(\ldots),$$

where $(\ldots)$ is a projection, join, or restriction, may be replaced by

$$R[K, F_i](\ldots)$$

to preserve input/output equivalent behavior. Source program updates of the form **INSERT** R_i will require DBA interaction to yield equivalent target program updates. Because **COMPOSE** is not information preserving, there is no inverse for this transformation.

6.4 Partitioning/Merging Relations

Consider a source relation R defined by

$$R(K^*, F)$$

where K and F are groupings of attributes. This relation may model a generic entity, such as employee. It may be desirable to partition [6] this entity into several disjoint specialized entities, such as machinist, plumber, or typist, where each is modeled by a relation of the form

$$R_i(K^*, F).$$

having the same attribute groupings as the entity modeled by R. The two transformations **PARTITION** and **MERGE** [6] allow the DBA to partition a representation of an entity into disjoint representations or merge specialized representations into a single merged one. The syntax of these transformations is

PARTITION "relation name"
INTO "partition clauses"
ELSE "relation definition".

and

MERGE "relation definition" **FROM** "relation name list".

where "partition clauses" is one or more clauses of the form

("condition") : "relation definition"

separated by semicolons. For the above relations, the transformations would be

```
PARTITION R
  INTO (c_1):      R_1(K*, F);
       (c_2):      R_2(K*, F);
        .
        .
        .
       (c_{n-1}):  R_{n-1}(K*, F)
  ELSE R_n(K*, F).
```

and

MERGE $R(K^*, F)$ **FROM** $R_1, R_2, \ldots, R_n$.

The constraints that these transformations must satisfy and their effects are presented using the above rotation.

These transformations may be illustrated by a simple schema with only one relation, **employees (name*, title*, age)**, and a second schema with three relations, **managers (name*, title*, age)**, **secretaries (name*, title*, age)**, and **clerks (name*, title*, age)**. The **partition** transformation

```
PARTITION EMPLOYEES
  INTO  (title = 'manager'):  managers (name*, title*, age);
        (title = 'secretary'  OR title = 'typist'):
                              secretaries(name*, title*, age);
  ELSE  clerks(name*, title*, age).
```

would split the tuples in the **EMPLOYEES** relation into three relations. The value of **TITLE** in the **MANAGERS** relation would be the same for all the tuples and could be disposed of with a **DELETE ATTRIBUTES** transformation. The **MERGE** transformation

```
MERGE employees (name*, title*, age) FROM managers,
  secretaries, clerks.
```

would combine the tuples in these three relations into a single relation. Before the **MERGE** would be permitted, a check would have to be made to ensure that a **name** value did not appear in two of the relations to ensure uniqueness in the target relation.

PARTITION is an information preserving, data-dependent, and query program-independent transformation. Each of the target relations defined by "partition clauses" must have the same key and non-key attributes as the relation being partitioned. Each of the $n - 1$ conditions c_j is a Boolean condition that must evaluate to true or false for every tuple of the source relation R. Each tuple $r \in R$ will be mapped by **PARTITION** to one of the n relations R_i by evaluating the $n - 1$ conditions c_j. If, for r, exactly one condition c_j is true, r is mapped to R_j. If, for r, no condition c_j is true, r is mapped to R_n. Otherwise there must be two or more conditions c_j that are true for r. In this case, the transformation system is unable to map r to a specific R_i without DBA intervention. Hence **PARTITION** is data dependent. The target schema is obtained from S by replacing the definition of the relation R with the definitions of the n relations R_i. The target database is obtained by executing the following program

```
set $R_1, R_2, \ldots, R_n$ empty
for each $r \in R$ do
  if $c_1$ then $R_1 \leftarrow R_1 \cup r$
  else if $c_2$ then $R_2 \leftarrow R_2 \cup r$
       else if $c_3$ then $R_3 \leftarrow R_3 \cup r$
               .
               .
               .
            else if $c_{n-1}$ then $R_{n-1} \leftarrow R_{n-b} \cup r$
                 else $R_n \leftarrow R_n \cup r$.
DELETE $R$.
```

From the above program, we observe that $R = R_1 \cup R_2 \cup \cdots \cup R_n$. Hence, all source program queries of the form

$$R(\ldots)$$

may be replaced by input/output equivalent queries of the form

$$(R_1 \cup R_2 \cup \cdots \cup R_n)(\ldots)$$

where $(\ldots)$ is a projection, join, or restriction. Source program updates of the form

INSERT R

may be replaced by

```
if $c_1$ then INSERT $R_1$
else if $c_2$ then INSERT $R_2$
      .
      .
      .
     else if $c_{n-1}$ then INSERT $R_{n-1}$
          else INSERT $R_n$.
```

so long as the conditions c_j will continue to map each occurrence of R into exactly one of the relations R_i for $i = 1, n - 1$. If this condition cannot be guaranteed, then the DBA must assume responsibility for transforming source updates to produce equivalent target updates. Finally, **PARTITION** may be inverted by **MERGE**.

MERGE is a non-information-preserving, data-dependent, and program-dependent transformation. As with **DECOMPOSE** and **COMPOSE**, there is a subtle semantic difference between the source and target schemata. Merge creates the target relation R by taking the union of the source relations R_i. This implies that each higher level entity described by R may occur at most once as a lower level entity R_i. Hence the source database must satisfy the constraint

$$\exists\, v(r_i \in R_i, r_i[K] = \{v\}$$
$$\Rightarrow \forall\, h(1 \leq h \leq n, h \neq i), \nexists\, r_h \in R_h \ni r_h[K] = \{v\})$$

for all $i(1 \leq i \leq n)$. The validity of this constraint may be examined for the source database by determining whether or not $R_i[K] \cap R_j[K] = \phi$ for all i and j such that $1 \leq i < j \leq n$. Should any pairwise intersection be other than the empty set, this implies that an instance of the higher level generic entity is represented by two lower level tuples $r_i(K_i, F_i)$ and $r_j(K_j, F_j)$ such that $r_i[K_i] = r_j[K_j]$ holds, but the nonkey attribute values may not be equal. The transformation system will require DBA guidance in determining which of these tuples to map to the higher level target relation R. Because data may be lost as a result of this transformation, and because the lower level entities R_i lose their identity when mapped to R, this transformation is not information preserving. Data dependence follows from observing that the source database must be examined to determine whether or not it is valid with respect to the target database system. Program dependence can best be illustrated by considering a source query of the form

$$R_i(\ldots),$$

where $(\ldots)$ is a projection, join, or restriction. Under the target system, the relations R_i may be materialized by a projection over R. However, the criteria needed to partition R into the disjoint relations R_i are not available to the transformation system. They could be made available in some cases by an elaborate **MERGE** statement, but in general only the DBA can provide these criteria. Hence, DBA interaction is required to modify every source query to an acceptable equivalent target query, if possible. Similary, all source updates will require DBA interaction to produce equivalent target updates. Because **MERGE** is not information preserving, it has no inverse.

6.5 Functional Dependency Transformations

We have assumed thus far that the source and target DBSs are in fourth normal form; that is, the key of every relation is minimal and there are no partial, transitive, or multivalued dependencies. As an example, consider the functional dependency $\mathrm{fd}_1 : K, P \rightarrow F, G$ where K, P, F, and G are nonempty groupings of attributes. The relation

$$R(K^*, P^*, F, G)$$

is a fourth normal form relation representing this functional dependency. Should the environment change so that in addition to fd_1 the functional dependency $\mathrm{fd}_2 : P \rightarrow G$ becomes true, then R is no longer in fourth normal form. This can be rectified by replacing the relation R by the two relations

$$R'(K^*, P^*, F)$$

and

$$T(P^*, G),$$

both of which are in fourth normal form. In a similar manner, the existence of transitive dependency $\mathrm{td}: F \rightarrow G$ or a multivalued dependency $\mathrm{md}: K, P \rightarrow\rightarrow G$ would imply that the relation R above was no longer in fourth normal form and should be replaced by other fourth normal form relations reflecting the new dependencies. The first three transformations in this class may be employed to restore DBSs to fourth normal form in response to dependency changes. The fourth transformation may be used to invert the effects of the first three.

6.5.1 *Partial Dependencies.* The transformation.

```
EXPORT PARTIAL DEPENDENCY FROM "relation name"
  GIVING "relation definition".
```

may be employed to restore a DBS system to fourth normal form when "relation name" is no longer in fourth normal form as the result of the perception of a partial dependency of one or more nonkey attributes of "relation name" on the key. Specifically, the transformation

EXPORT PARTIAL DEPENDENCY FROM R **GIVING** $T(P^*, G)$.
transforms the relation

$$R(K^*, P^*, F, G)$$

to the two relations

$$R'(K^*, P^*, F)$$

and

$$T(P^*, G).$$

EXPORT requires that the key attributes of the target relation T be a proper subset of the key attributes of R; the nonkey attributes of T be a subset of the nonkey attributes of R; and for all pairs of tuples (P_1, G_1) and (P_2, G_2) in the projection $R[P, G]$ over the source database $P_1 = P_2 \Rightarrow G_1 = G_2$ holds. If these constraints are satisfied, **EXPORT** is an information preserving, data-dependent, and query program-independent transformation. Violation of the first two constraints may be determined by an examination of the source schema S. The validity of the third constraint requires an examination of the source database to determine whether or not the attributes P are a key for the proposed target relation T. If they are not, DBA intervention is required to modify the source stored database. Information preservation follows from observing that the equijoin of R' and T over the attributes P recreates the source relation R. Hence no information is lost in the transformation. The target database may be derived from the source database by executing the program

$T \leftarrow R[P, G]$.
$R' \leftarrow R[K, P, F]$.

Source queries of the form

$$R(\ldots)$$

may be replaced by

$$(R'[P = P]T)(\ldots)$$

preserving input/output equivalent retrieval behavior. Source updates of the form **INSERT** R may not be handled so neatly because such a statement implies the insertion of an R' tuple and possibly a T tuple. Hence DBA interaction will be required to transform source updates. **EXPORT** may be inveted by the transformation **IMPORT**.

6.5.2 *Transitive Dependency Transformations.* Let the source relation R defined by

$$R(K^*, F, G)$$

model the functional dependency fd:$K \rightarrow F, G$. With the passage of time, the transitive dependency td:$F \rightarrow G$ may be perceived. In this case, R is no longer in fourth normal form. The transformation

EXTRACT TRANSITIVE DEPENDENCE FROM R **GIVING** $T(F^*, G)$.

yields the target relations

$$R(K^*, F)$$

and

$$T(F^*, G).$$

This transformation differs from **EXPORT** only in that the key of the target relation T must be a proper subset (if G is empty, then there is no transitive dependency) of the nonkey attributes of the source relation R. Otherwise the constraints and effects of this transformation are identical to those of **EXPORT** for like reasons. Hence **EXPORT** is information preserving, data-dependent, and query program independent with an inverse of **IMPORT**.

6.5.3 *Multivalued Dependency Transformations.* The treatment of multivalued dependencies differs from that for partial and transitive dependencies in that the target relation added to the schema is composed solely of key attributes. Hence, given a source relation R defined by

$$R(K^*, F, G)$$

and a change in the external environment resulting in the perception of the multivalued dependency md:$K \rightarrow\rightarrow F$, the transformation

REMOVE MULTIVALUED DEPENDENCY FROM R **GIVING** $T(K^*, F^*)$.

yields a target database system containing the relations

$$R'(K^*, G)$$

and

$$T(K^*, F^*).$$

Because the projection $R[K, F]$ is a set and the target relation T is "all key", this transformation is data independent. For reasons similar to **EXPORT** and **EXTRACT**, this transformation is information preserving and query program independent. The target schema is derived from the source by adding the relation T and deleting the nonkey attributes of R that are mapped to T from the definition of the target relation R'. The target database may be obtained from the source by executing the program

```
$T \leftarrow R[K, F]$
$R' \leftarrow R[K, G]$.
DELETE $R$.
```

Source program queries of the relation R may be replaced by input/output equivalent queries of the equijoin of T and R' over the attributes K. Source program updates will require DBA interaction to produce acceptably equivalent target program updates. Inversion of **REMOVE** is a two-step process. In the first step, the attributes F of the relation T are demoted from key to verify that the functional dependency fd: $K \rightarrow F$ has replaced the multivalued dependency md: $K \rightarrow\rightarrow F$. In the second step, the relation T is imported into R' to restore the original source relation R.

6.5.4 *Importing Dependencies.* The preceding three transformations allow the DBA to restore a database system to third normal form whenever a partial, transitive, or multivalued dependency is perceived. **IMPORT** is provided to invert these transformations. Consider the source relations

$$R(K^*, G)$$

and

$$T(P^*, F)$$

where P the key of T is a subset of the attributes of R, and the nonkey attributes F of the relation T are not attributes of R. (We may have $P = K$, $P \subset K$, $P = G$, or $P \subseteq (K \cup G)$. We must have $F \cap (K \cup G) = \emptyset$.) The transformation

IMPORT DEPENDENCY BY MAPPING *T* **INTO** *R*.

yields a target database system in which the relation

$$R'(K^*, F, G)$$

replaces both T and R. For this DBS to be in fourth normal form, the functional dependency fd: $K \rightarrow F$ must be true, as is the case when inverting the three preceding transformations. To ensure that no stored data values are lost, we require that for every source tuple in T there be at least one tuple in R having the same value for the key attributes P of T, and that for every source tuple in R there be exactly one tuple in T having the same value for the attributes P. These two constraints ensure that no tuple of T will be lost when mapping T into R and that for every tuple in R there is a tuple in T to map into R. They also imply that this transformation is data dependent and may require the DBA to add/delete tuples to or from T or R prior to allowing the transformation. **IMPORT** is not information preserving because the two tuples of T and R in the source schema are replaced by the single tuple of R' in the target schema, losing information about the independent existence of T and R. The cardinality of R is the same as R', but given only R' it is impossible to determine the cardinality or contents of T. The target database is derived from the

$R' \leftarrow R[P = P]T$.
DELETE *T*.
DELETE *R*.

Source program queries of the form

$$T(\ldots)$$

or

$$R(\ldots)$$

may be replaced by

$$R'[P, F])(\ldots)$$

and

$$(R'[K, G])(\ldots),$$

respectively, to preserve input/output equivalent retrieval behavior.

Source program updates involving either T or R will require DBA interaction to obtain equivalent target program updates. Because **IMPORT** is not information preserving, it has no inverse.

7. CONCLUSIONS

The relational model transformations offered in this paper may provide database administrators with increased flexibility by easing the conversion process when requirements change. Whether this set of transformations is at the right level of abstraction, suits the needs of commercial environments, or is comprehensible by practitioners needs to be tested empirically. We hope to gain more experience with these transformations and see how they fit with relational query facilities such as query-by-example or SQL.

Our goal is not to create a software product, but to demonstrate to implementers of database systems that automatic conversion is viable. We hope that these implementers will include some of our transformations or additional ones in new systems so that their effectiveness can be tested.

We are investigating what further transformations may be useful, such as combining two database systems or changing functional dependencies. More sophisticated transformations which impact several relations at a time are being considered in the context of the entity relationship or other high-level semantic models. Extending the relational model to include interrelation integrity constraints opens the door to many interesting transformations, such as **DISTRIBUTE** and **FACTOR**, which we explored in the Pure Database System [8, 10, 16]. We are also trying to take a formal approach to demonstrating the completeness of a set of transformations and to proving the correctness of a transformation.

ACKNOWLEDGMENT

We thank Michael Brodie, John Grant, and the referees for their comments.

REFERENCES

1. CODD, E.F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377-387.
2. COLLICA, J., SKALL, M., AND BOLOTOSKY, G. Conversion of federal ADP systems: A tutorial. NBS Special Publication 500-62 (Aug. 1980).
3. HOUSEL, B. A unified approach to program and data conversion. In *Proc. 3rd Int. Conf. Very Large Data Bases* (Tokyo, Oct. 6-8, 1977), ACM, New York.
4. JACOBS, B. Applications of database logic to automatic program conversion. Submitted for publication.
5. NAVATHE, S.B. Schema analysis for database restructuring. *ACM Trans. Database Syst. 5*, 2 (June 1980), 157-184.
6. NAVATHE, S.B., AND FRY, J.P. Restructuring for large databases: Three levels of abstraction. *ACM Trans. Database Syst. 1*, 2 (June 1976), 138-158.
7. SAKAI, H. Entity-relationship approach to the conceptual schema design. Proc. ACM SIGMOD Conf. 1980, pp. 1-8.

8. Shneiderman, B., and Thomas, G. Automatic database system conversion I: Data definition and manipulation facilities. Computer Science Tech. Rep. Series TR-820, Univ. Maryland, College Park, 1980, 39 pp. (Submitted for publication.)
9. Shneiderman, B., and Thomas, G. Path expressions for complex queries and automatic database program conversion. In *Proc. 6th Int. Conf. Very Large Data Bases* (Montreal, Oct. 1-3, 1980), ACM, New York, pp. 33-44.
10. Shneiderman, B., and Thomas, G. Automatic database system conversion: Schema revision, data translation, and source-to-source program transformation. In *Proc. National Computer Conf.*, vol. 51, AFIPS Press, Montvale, N.J., 1982.
11. Shu, N.C., Housel, B.C., Taylor, R.W., Ghosh, S.P., and Lum, V.Y. EXPRESS: A data extraction, processing, and restructuring system. *ACM Trans. Database Syst. 2*, 2 (June 1977), 134-174.
12. Shu, N.C., Housel, B.C. and Lum, V.Y. CONVERT: A high level translation definition language for data conversion. *Commun. ACM 18*, 10 (Oct. 1975), 557-567.
13. Su, S.Y.W. Application program conversion due to database changes. In *Proc. 2nd Int. Conf. Very Large Data Bases* (Brussels, Belgium, Sept. 1976), North-Holland, Amsterdam, pp. 143-158.
14. Su, S.Y.W., and Lam, H. Transformation of data traversals and operation in application programs to account for semantic changes in databases. Dep. Computer and Information Sciences, Univ. Florida, Gainesville, 1979.
15. Su, S.Y.W., and Reynolds, M.J. Conversion of high-level sublanguage queries to account for database changes. In *Proc. National Computer Conf.*, vol. 47, AFIPS Press, Montvale, N.J., 1978, pp. 857-875.
16. Thomas, G., and Shneiderman, B. Automatic database system conversion II: A transformation language. Computer Science Tech. Rep. Series TR-821, Univ. Maryland, College Park, 1980, 46 pp. (Submitted for publication.)
17. Thomas, G., and Shneiderman, B. Specifications for automatic relational database system conversion. Submitted for publication.
18. Thomas, G., and Shneiderman, B. Automatic database system conversion: A transformation language approach to sub-schema implementation. In *Proc. IEEE COMPSAC 1980 Conf.*, IEEE, New York, 1980, pp. 80-88.

Received October 1980; revised February 1981; accepted April 1981