

Copyright © 1979, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

HETEROGENEOUS DATA MODELS - PART I SEMANTIC ISSUES

by

R. H. Katz and E. Wong

Memorandum No. UCB/ERL M79/56

21 August 1979

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ABSTRACT

In this two-part report we shall compare several widely used data models using a number of operational criteria. There being no one model that is uniformly superior, we shall consider ways of dealing with heterogeneous data models in a single system. The problems that must be dealt with include schema conversion and program decompilation.

Research sponsored by the Army Research Office Grant DAAG29-78-G-0186, the Air Force Office of Scientific Research Grant 78-3596 and the Honeywell Corporation.

1. INTRODUCTION

Recent interest in data models has undertaken a subtle change in emphasis, from one on the comparative advantages of different types to a recognition that diversity and heterogeneity in data models may be both necessary and desirable. Two important technological developments have been responsible for this shift. First, in a distributed database system, there may be a need to integrate existing local systems that employ different data models, and a need to adopt a global model that is different from the local models in order to achieve communication efficiency. Second, there is a growing recognition that the different purposes to be served by a data model cannot be simultaneously attained by a data model of a single type. The latter consideration has led to several proposals for multi-schema database architecture.

While the terms "data model" and "schema" are often used interchangeably, we think it is useful to make a consistent distinction. We shall use the term "data model" to mean a generic type, consisting of a collection of data object types, and "schema" to mean a specific set of data objects. For example, a "relational data model" consists of the types "domains" and "relations", while the following is an example of a relational schema:

```
professor (pname, rank, dept)
student   (sno,  sname, major)
course    (cno,  title, prereq, dept)
```

In this report our objective is threefold:

- (1) To articulate the operational purposes that are to be served by a data model.
- 2) To gain a full understanding of the differences and similarities

between different data model types.

(3) To find mapping algorithms that would support different model types in the same system while preserving the underlying semantics. In particular, we think that this means developing the following capabilities.

- (a) Schema Conversion - An ability to convert a schema of one type into a schema of another type without loss of semantic information. The test for information preservation should be reversability.
- (b) Program Conversion - An ability to convert a program expressed in one data manipulation language into one of another DML, even when the source is a procedural language and the target is a specification language.

The remainder of this report will be organized along the lines of these objectives. First, we shall consider the principal operational goals of a data model. Next, we shall briefly consider some major data models and their relative strengths and weaknesses. Using "update consistency" as a focus, we shall propose a data model for the purpose of logical design. The design model will then serve to explicate the semantics of both the relational data model and the network model of CODASYL-DBTG.

We shall present mapping rules for converting a design schema into a schema of either the relational or the DBTG type so as to be free of update anomalies. We shall then augment the semantics of both DBTG and relational data models to make the mapping reversible. In the process, a DBTG-relational schema conversion algorithm is thus obtained.

The introduction of a design data model provides a sufficient semantic bridge between the DBTG and relational data models to render reversible schema conversion possible. However, it is not sufficient to permit

the conversion of a program expressed in DBTG-DML into one expressed in any of the non-procedural relational languages. Here, more than semantic translation is involved, and a process of decompilation is necessary. To make decompilation possible, we need yet another bridge. For that purpose we shall introduce an "access-path data model." The idea here is to define atomic units that can be matched to low level DML operations on the one hand, and combined through simple syntactical rules into high-level operators on the other.

This report will appear in two parts. Part I will deal with all semantic aspects of data models, and Part II will introduce access paths and consider program translation. Some of the material in Part I also appears in [Wong 78].

2. OPERATIONAL GOALS OF DATA MODELS

The role of all data models is to take part in the interface between application programs and the stored data. As such, there are some broad objectives that a data model must serve.

(a) It must be a good interface for the data manipulation. Two issues are involved here. First, the constructs of the model should support, naturally and easily, high-level operators, i.e., operators that specify in a general way subsets of the database. Second, there should be closure. This means that the result of an operation can be operated on once again so that data-manipulation operations can be concatenated to yield an algebraic structure. Set-at-a-time operators and closure are essential to a powerful non-procedural data manipulation language. In addition to providing ease of programming, such a language also contributes to program longevity. As long as the interpreter or compiler continues to be optimized, the program will not only run, but run efficiently as storage structure changes.

(b) A data model must be a good interface for logical design.

Specifically, it should facilitate the design of schemas that are free of update anomalies and support flexible growth and change. The issue here is primarily one of semantics. There must be sufficient semantics in the data model to allow the designer to control atomicity and side effects of update, and to keep apart in the schema components that are independent semantic units.

(c) A data model must be a good interface for storage design. There are several issues here. First, the constructs of the data model should lend themselves to simple implementation. In particular, any integrity constraint associated with the data construct should be supported by implementation without requiring procedural verification on updates. The data model should also be a vehicle for conveying access requirements, not at a detailed quantitative level, but at a structural level.

It will turn out that a data model that achieves one or another of these operational goals will also attain other subsidiary objectives. For example, a model suitable for logical design will have enough semantics for it to be a good vehicle for schema translation. A model that is particularly good for expressing access requirements will also be a good vehicle for re-expressing and aggregating the record-at-a-time operations of a procedural language into higher level operators. These capabilities, essential to a system supporting heterogeneous data models, are enhanced by an elucidation of the basic goals served by a data model, and how they are served by models of different types.

3. DATA MODELS OF THE MAJOR TYPES

3.1 Relational [CODD 70]

Let D_1, D_2, \dots, D_n be non-empty sets, not necessarily distinct. A tuple (r_1, r_2, \dots, r_n) is an ordered collection of elements with r_i coming from D_i , $i = 1, 2, \dots, n$. A relation R is any collection of distinct tuples,

and the D_i 's are called the domains of R. A database of the relational type is a collection of time-varying relations in which the tuples of each relation change as it gets updated but the domains of the relations do not change (until the database is redesigned). Data definition for a relational database is given by a schema which specifies for each relation its name and domains, and defines the domains. The domain definition, having nothing to do with the data model, will be omitted from all our examples.

The only data object types in a relational model are sets that serve as domains and relations. In nearly all proposed relational data manipulation languages, the only objects of data manipulation are relations. Each operator takes one or more relations and produces a new relation, which can be operated on again. Clearly, the operators are high level and achieve closure. As a result, a great deal of expressive power and data independence is achieved.

The semantic sparsity that makes the relational model so good for data manipulation also makes it a poor interface for logical design. It has long been known that one can easily arrive at a relational schema that poses update problems, and it is not possible to state, in terms of the semantics of the relational model, rules for design schemas that are free of these update anomalies. One approach has been to augment the semantics of the relational model with "functional dependencies" and define "normal forms" in terms of these. The resulting theory is complicated and not without inconsistencies. [CODD 71, BERN 76, FAGI 77]. Most practitioners find the theory difficult to understand, and not usable as a tool for design.

3.2 CODASYL-DBTG (Network) [DATE 77]

The principal constraints here are records and sets. In a simplified view, a record is a tuple of values but there is no restriction that the

same tuple does not recur. Hence, there may be many record occurrences with the same values, and a record occurrence is not self-identified. A record type is defined by specifying its name and the names of the sets from which the data values come (called data items).

A set type (again somewhat simplified) is defined by an ordered pair of record types, called owner and member respectively. A set of a given type consists of one and only one occurrence of the owner type and zero or more occurrences of the member type. Presumably, two record occurrences with identical data-item values will have different set participation somewhere, for otherwise they would represent identical data.

The existing DML (data manipulation language) for systems of the CODASYL-DBTG type involves record-at-a-time operations. The operations involve navigating through the database from record to record using both their data-item values and their set connections. [BACH 73] The language must be considered both low-level and procedural. The programs are generally much longer than the equivalent relational programs, and their execution efficiency is strongly affected by any changes in implementation.

A number of other features exist in a DBTG schema that are either implementation related or update related. In particular, deft use of the "set membership options" on INSERT/DELETE is essential to good schema design, but the constructs of the model do not provide a clear conceptual framework for logical design. It is easy to arrive at reasonable schemas that are not anomaly-free, and it is difficult to give general design rules for avoiding them.

As an end product of evolution from extensive use, the DBTG model is rather good as a storage interface. The structures are easily implemented and most of the integrity constraints enjoy support through implementation. Furthermore, a DBTG schema is also a good vehicle for expressing user

access requirements.

3.3 Entity-Relationship Model [CHEN 76]

Perhaps the most natural way of viewing data is as descriptions of objects: things, people, places, etc. Collectively, they will be referred to as entities. We begin with the following constructs: entity sets which are sets of objects to be described, and value sets whose elements are used to describe entities. The simplest description is through properties. We define a property as a function mapping an entity set into a value set. A relationship is a relation (in the sense previously defined) whose domains are entity sets. These are the basic constructs of the E-R model. The following is an example of a schema of the E-R type:

entity sets: Emp, Dept
properties: Emp \rightarrow Ename, Salary
 Dept \rightarrow Dname, Floor
relationship: Assignment (Emp, Dept)
 Manager (Dept, Emp)

There are integrity constraints associated with many relationships that need to be recognized by the semantics of the data model. Two of these: single-valued and complete are of particular operational importance. A relationship $R(E_1, E_2, \dots, E_n)$ is said to be single-valued in E_1 if each element of E_1 occurs at most once in R , and complete in E_1 if each element of E_1 occurs at least once. A binary relationship (i.e., one with two domains) $R(E_1, E_2)$ that is both single-valued and complete in E_1 is a function mapping E_1 into E_2 . Such a relationship will be distinguished and identified as a new construct that we shall call an association of E_1 to E_2 and denote by $E_1 \rightarrow E_2$. An association of E_1 to E_2 requires that a unique element in E_2 be associated with every element in

E_1 , and represents a tight coupling of E_1 to E_2 . For example, let E_1 be Emp, E_2 be Dept and the association be Assignment. Then, an employee is assigned to a unique department and no employee can exist without being assigned to a department.

It is useful to define "property of relationship" as a construct. For example, consider the relationship enroll (student, course). For each instance of the relationship, we may have "grade" as a property. It is also possible to add relationship of relationships, but to do so makes the model unnecessarily complex. Situations where such a construct might be appropriate can always be modelled by introducing an additional entity set, and doing so has no important operational disadvantage that we know of.

As we shall attempt to demonstrate in a later section, the E-R model is an ideal data model for logical design, because design decisions are few and the consequences of design choices are exceedingly clear.

As an interface for data manipulation the E-R model is relatively undeveloped. There have been data manipulation languages proposed for it, but these are rather simple variants of the relational languages, without specific attention being given to the semantics of the E-R model. A major goal in designing a data manipulation language for it should be to make every syntactically correct query a meaningful one.

3.4 Summary

We see that the relative merits and elements of data models depend on the goal. What is good for one thing is far from good for another. Heterogeneity, then, is not only necessary in certain circumstances, but is desirable and indeed essential for a full understanding of the different roles that data models play.

4. LOGICAL DESIGN

4.1 Why is the E-R Model Ideal for Logical Design?

I think the answer is multi-faceted.

(a) Design decisions are few

Identifying objects in a schema in most cases is both natural and automatic. There is really only one design decision that is operationally important, and it is in the choice of associations.

(b) The consequences of a design choice are clear

The principal consequences of choosing to identify a binary relationship as an association is to cause certain side effects on insertions and deletions. If, for example, "assigned" is an association of Emp to Dept, then deleting a department will force the deletion of every employee assigned to that department. Further, no employee can be inserted without the department to which he is assigned already having been inserted.

(c) Update anomalies are well understood

We shall adopt "no update anomalies" as the primary goal for logical design. By update anomalies we mean either

fragmentation of an atomic operation

or

unplanned side effects.

To make these ideas clear requires atomicity and side effects to be defined. These concepts are simply and naturally defined in terms of the constructs of the E-R Model.

4.2 Update Anomalies

By an atomic operation we shall mean one of the following:

- (i) Inserting or deleting an entity
- (ii) Inserting or deleting an instance in a relationship
- (iii) Changing the value of an association or a property.

It is understood that inserting/deleting an entity means the simultaneous specification/removal of the values of all its functions. For example, if the entity set Emp has properties: "Ename", "Age", and an association: "Assigned", then inserting an employee means specifying his name, his age and the department to which he is assigned.

By a side effect of an atomic update operation we mean either an additional update required for preserving the integrity of the schema, or a constraint on the order of insertion of another update. For example, if Assigned is an association mapping Emp to Dept, then the deletion of a department will require the deletion of all employees assigned to it. Furthermore, before an employee can be inserted, the department to which he is assigned must already exist in the database. A consequence of the constraint in the order of insertion is that no cycle of associations can exist. For example, we cannot have Emp $\xrightarrow{\text{Assigned}}$ Dept and Dept $\xrightarrow{\text{Mgr}}$ Emp both as associations, because then a department and its manager cannot be inserted in either order without the integrity constraint of the association be violated, at least temporarily.

We observe that only update operations of the type: Inserting/Deleting an entity have side effects. In addition to the side effects due to association, the deletion of an entity causes any instances of any relationship in which it participates to be deleted.

4.3 An Example of Logical Design

Through an example, we hope to show that for the E-R model schema

design is simple, and the design decisions limited.

Suppose that our database concerns course-offerings, teaching-schedule and class-enrollment. Clearly, the database describes "students", "courses" and "professors" and these should correspond to entity sets. In addition, assume that dept is also an entity set. It is easy to list for each entity set "things" that describe the entities but are not themselves being described in the database, and these will be the properties. Let's say we have the following properties:

entity set	properties
student	regno, sname, class yr
professor	ssno, pname, rank
course	cno, title
dept	dname, location

We may hesitate momentarily on one or two of the properties. For example, should the "ranks" be represented by an entity set? The answer is an operational one. It depends on whether the "ranks" are themselves being described in the database, e.g., by being given a salary for each rank. For our example we assume that they are not, so that the collection of all ranks is a value set and the correspondence of "professor" to "rank" is a property, not an association.

Next, consider data that relate entities from different sets. Some of these involve a pair of entity sets (i.e., binary), e.g.,

major (student, dept)
faculty (professor, dept)
chairman (dept, professor)
preq (course, course)

We need to decide for each of these whether it is a function. That is, is it single-valued and complete in one of the two entity sets? For example, must every student have a major-dept, and only one? Must every professor be on the faculty of one and only one department? The ones that are functions can be identified as associations subject to the constraint that no cycle of associations can exist. Observe that we say "can be identified" and not "must be." The designer is free to decide whether the integrity constraints "single-valued" and "complete" are to be enforced through implementation, and whether the update side effects that are concomitants of this enforcement are desirable. Let's say that for our example, we decide to identify student $\xrightarrow{\text{major}}$ dept and dept $\xrightarrow{\text{chairman}}$ professor as associations, but not "faculty" and "preq."

Next, we note that a "class" involves students, professors, and courses. Let's tentatively denote this by a relationship

enrollment (course, professor, student).

However, by definition a relationship cannot have duplicate tuples. Hence, as it stands, "enrollment" cannot represent the possibility that a student repeats a course from the same professor. To circumvent this problem, we introduce an additional entity set offering with properties: year, term, time, and redefine the enrollment relationship as

enrollment (course, professor, offering, student).

Finally, let grade be a property of enrollment, i.e., a grade-value is assigned to each instance of enrollment.

Our final schema is given as follows:

entity set	properties
student	regno, sname, class yr
professor	ssno, pname, rank
course	cno, title
dept	dname, location
offering	year, term, time

associations: student $\xrightarrow{\text{major}}$ dept

dept $\xrightarrow{\text{chairman}}$ professor

relationships: faculty (professor, dept)

prereq (course, course)

enrollment (course, professor offering, student)

with property: grade

4.4 Designing a Relational Schema

Our approach to designing a relational schema will be as follows: first, we design a schema on the E-R model. Then, we shall find an algorithm that maps the E-R schema into a relational schema so as to preserve atomicity and side effects. We shall define an atomic operation in a relational system as any operation that affects a single tuple. Once we have a one-to-one correspondence in atomic operations between the E-R and relational models, side effects, being defined in terms of atomic operations, are automatically defined for the relational data model.

The mapping rules for transforming an E-R schema into an anomaly-free relational schema are exceedingly simple and are given as follows:

(4.4.1) Make sure that every entity set has a unique identifier. Introduce one if necessary. An identifier is a property of an

entity set such that the identifier-value of any entity can never change. It is a surrogate for the entity. [CODD 79]

(4.4.2) For each entity set E, group its identifier, its properties and all associations of E in a single relation.

(4.4.3) Map each relationship R into a relation \mathcal{R} . The identifiers of the entity sets participating in R are domains of \mathcal{R} , as are any properties of R.

Let's apply these rules to the example of 4.3.

First, we assign the following identifiers to the entity sets:

entity set	identifier
student	regno
professor	eno
course	cno
dept	dno
offering	class-id

The relations that result from entity sets are given by:

relation	domains
student	regno, sname, class yr, major-dno
professor	eno, pname, rank
course	cno, title
dept	dno, dname, location, chairman-eno
offering	class-id year, term, time

The relations that result from relationship are as follows:

relation	domain
faculty	eno, dno
prereq	cno, preq-cno
enrollment	cno, eno, class-id, regno, grade

4.5 Designing a CODASYM-DBTG Schema

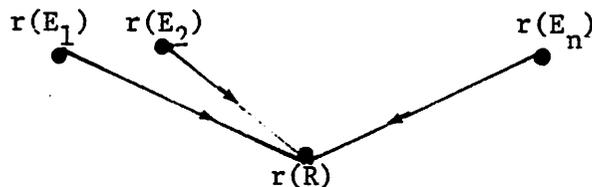
Our approach to designing a DBTG schema is similar to the relational case. Here, an atomic update operation is defined as any operation that affects a single record occurrence and its set participation. The mapping rules are given as follows:

(4.5.1) Choose identifiers as in (4.4.1).

(4.5.2) For each entity set E define a record type $r(E)$ with the properties of E as its data items.

(4.5.3) For an association of E_1 to E_2 define a set type s with $r(E_2)$ as owner and $r(E_1)$ as member. $(r(E_2) \xrightarrow{s} r(E_1))$

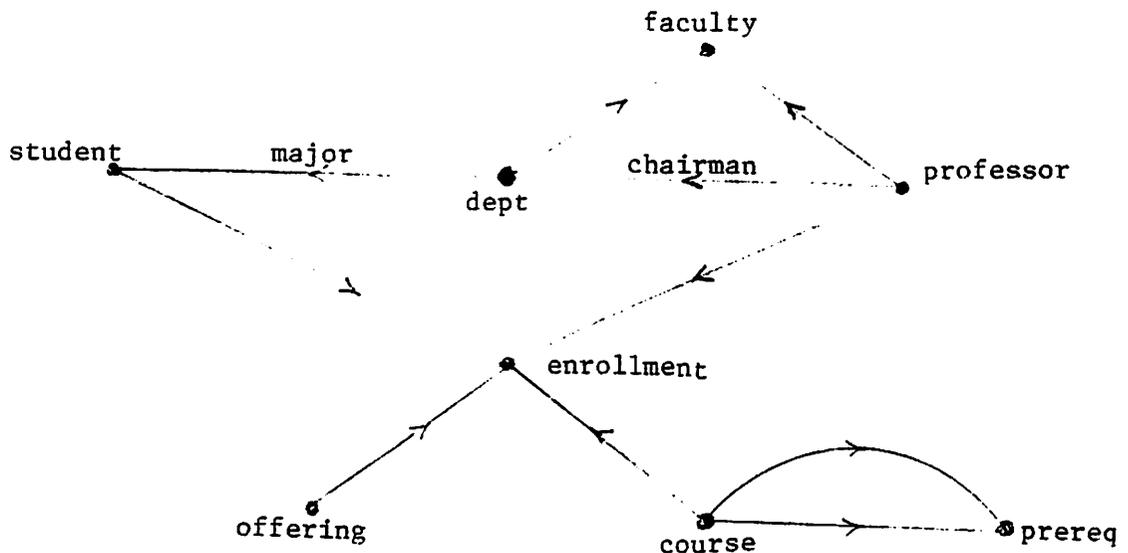
(4.5.4) For a relationship $R(E_1, E_2, \dots, E_n)$ define a "confluent hierarchy" consisting of a record type $r(R)$ and a set type s_i for each E_i with $r(E_i)$ as owner and $r(R)$ as member.



The properties of the relationship R (if any) are the only data items of $r(R)$.

(4.4.4) All set types have Automatic/Mandatory options for Insert/Delete. Applying these rules for the example of 4.3 yields the following

DBTG schema



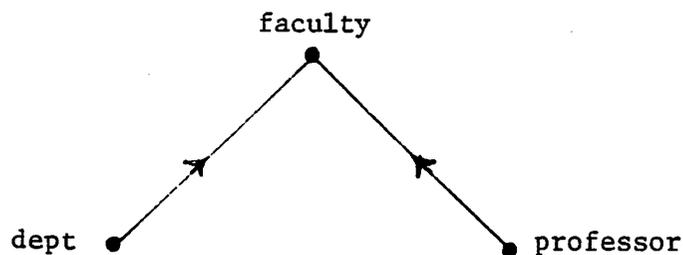
One improvement in the DBTG schema is possible with the addition of the following mapping rule:

- (4.5.6) For a binary relationship $R(E_1, E_2)$ which is single-valued in E_1 and has no property, the confluent hierarchy resulting from (4.5.4) can be replaced by a single set type $s(R)$



with Manual/Optional on Insert/Delete for $s(R)$.

Suppose that in our example the relationship faculty is single-valued in professor. Then, (4.5.6) would change



into dept  professor in the DBTG schema. The improvement consists of not only a simplification of the schema, but also an implementation support for the integrity constraint "single-valued."

4.6 Schema Translation

Let us call a schema of either the relational or the DBTG type well-designed if it results from applying the mapping rules of the last two sections to a schema of the E-R type. The class of all well-designed schemas does not comprise all legal schemas of the two types. Two questions naturally arise in this connection:

- (a) What is the minimum semantic augmentation to the relational and DBTG data model that is necessary in order for all well-designed schemas to be characterizable in terms of the constructs of the two respective models?
- (b) Can the mappings of sections 4.4 and 4.5 be reversed, so that a well-designed schema can be transformed back to the same E-R schema from which it was derived?

If the answer to (b) is yes, then a reversible translation algorithm between well-designed schemas of the relational and DBTG types follows immediately.

To find a characterization of well-designed schemas of the relational type, let us augment the semantics of the model by distinguishing among three types of domains. We shall then have key, foreign-key, (fk for short) and value domains. Consider the mapping rules in section 4.4. An entity set E gives rise to a relation $\mathcal{R}(E)$, and we identify the identifier of E as the key-domain of $\mathcal{R}(E)$, its properties as value-domains, and its associations as fk-domains. For example, the domains of the "student" relation would be classified as follows:

	key	value	value	fk
student	(regno,	sname,	class yr,	major)

We note that the values of the key-domain of a relation are in one-to-one correspondence with the tuples of the relation.

A relationship $R(E_1, E_2, \dots, E_n)$ gives rise to a relation $\mathcal{R}(R)$, and the identifier of the entity sets E_i will be designated as fk-domains of \mathcal{R} , and any properties of R are designated as value-domains.

We can now characterize a well-designed schema of the relational type as follows: A well-designed schema consists of relations of two types:

- (i) E-type - A relation of E-type has one key domain and 0 or more each of value and fk domains. Values of the key domain are unique.
- (ii) R-type - A relation of R-type has no key domain, 2 or more fk domains and 0 or more value domains.

It is obvious that we can now reverse the mapping given in section 4.4.

We observe that semantic augmentation of the relational model occurs at only the lowest level, viz., domains. This distinction between E-relations and R-relations follows automatically once the domains are classified.

For the DBTG case, let us distinguish data items of two types: key and value. The mapping rules of section 4.5 transform an entity set E into a record type $\mathcal{R}(E)$. This identifier of E maps into the key data-item of $\mathcal{R}(E)$, and the properties of E map into the value-data-items of $\mathcal{R}(E)$. We shall call a record type containing a key a self-identified record type. A relationship R is mapped into a record type $\mathcal{R}(R)$ which has no key-data-items and 0 or more value-data-items. These will be called link record types. We shall call a set type total if it has

automatic/mandatory rules for insertion/deletion, and partial if the membership option is manual/optional. For a well-designed schema no other combination is possible.

We can now characterize a well-designed DBTG schema as follows: A DBTG schema is well-designed if and only if it has self-identified and link records types, total and partial set types such that a link record is the owner of no set, but the member of two or more total sets.

The mapping of an E-R schema into a well-designed DBTG schema can now be reversed as follows:

(4.6.1) For each self-identified record type r , we define an entity set $E(r)$. The data items of r define the properties of $E(r)$, and the total sets of which r is a member define associations of r .

(4.6.2) For each link record type ℓ , we define a relationship $R(\ell)$. The data items of ℓ define the properties of $R(\ell)$, and the owners of the set of which ℓ is a member define the participating entity sets of $R(\ell)$.

(4.6.3) For each partial set type s with owner o and member m , we define a binary relationship $R(E(m), E(o))$ that is single-valued in $E(m)$.

Direct schema-translation between relational and DBTG data models is now straightforward save for the wrinkle that there is no provision in the relational model for recognizing a single-valued binary relationship that is not an association. We can make provision for such distinction in various ways, the simplest being to identify the single-value entity set with a key-domain. The result, however, is not entirely satisfactory. The basic problem is that in augmenting the semantics of

the relational model we have provided a means for recognizing the combined integrity constraint of the E-R model "single-valued" and "complete" but not the two separately. Of course, we could have added sufficient semantics to identify each one separately, but there is some question as to whether it is worth doing.

We observe that the characterization of well-designed schemas requires no augmentation of the semantics of the DBTG model. The concepts introduced: link and self-identified record types, total and partial set types, are defined in terms of constraints that already exist in the model. In this sense the semantics of the DBTG model are equivalent to that of the E-R model, except that the primitive constraints of the DBTG model are less suitable for logical design.

the relational model we have provided a means for recognizing the combined integrity constraint "single-valued" and "complete" but not the two separately. Of course, we could have added sufficient semantics to identify each one separately, but there is some question as to whether it is worth doing.

We observe that the characterization of well-designed schemas requires no augmentation of the semantics of the DBTG model. The concepts introduced: link and self-identified record types, total and partial set types, are defined in terms of constructs that already exist in the model. In this sense the semantics of the DBTG model are equivalent to that of the E-R model, except that the primitive constructs of the DBTG model are less suitable for logical design.

REFERENCES

- [BACH 73] Bachman, C. W., "The programmer as navigator." CACM 16, 11, (Nov. 1973), pp. 653-658.
- [BERN 76] Bernstein, P. A., "Synthesizing third normal form relations from functional dependencies," Transactions on Database Systems 1, 4 (Dec. 1976), pp. 277-298.
- [CHEN 76] Chen, P. P., "The entity-relationship model - towards a unified view of data." Transactions on Database Systems 1, 1 (Mar. 1976), pp. 9-36.
- [CODD 70] Codd, E. F., "A relational model of data for large shared data banks." CACM 13, 6 (June 1970), pp. 377-387.
- [CODD 71] Codd, E. F., "Further normalization of the data base relational model," Courant Computer Science Symposia 6, Data Base Systems, Prentice Hall, New York (May 1971), pp. 65-98.
- [CODD 79] Codd, E. F., "Extending the data base relational model." SIGMOD 79. Full paper to be published in CACM.
- [DATE 77] Date, C. J., An Introduction to Database Systems, 2nd. Ed., Addison-Wesley, Reading, Mass. 1977.

- [FAGI 77] Fagin, R., "The decomposition versus the synthetic approach to relational database design." Proceedings 1977 Very Large Data Bases Conference, 1977, pp. 441-446.
- [WONG 78] Wong, E., Katz, R. H., "Design goals for relational and DBTG databases," University of California, Berkeley, Electronics Research Laboratory Memorandum M78/89.