To the Graduate Council:

I am submitting herewith a thesis written by Benjamin Y. Lin entitled "Database Abstractions: An Interpretation and Implementation." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Kevin C. O'Kane, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Chancellor
Graduate Studies and Research

DATABASE ABSTRACTIONS: AN INTERPRETATION AND IMPLEMENTATION

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Benjamin Y. Lin

June 1982

ABSTRACT

In the relational data model, the database consists of a potentially large number of relations. No means of logically grouping the relations exists. In [3,4,5,6], Smith and Smith introduce two abstractions, aggregation and generalization, as a way to build hierarchical relationships among relations at the conceptual level. These abstractions give the database user better understanding and control over the conceptual design of the whole database. At the same time, they preserve the properties of the relational model:

> A relation is still a table and consequently the relation operations still can be performed on these relations; normalization theory still applies to tables so that tables can be transformed to various normal forms which are free of undesirable redundancy and update anomalies.

This thesis is based on the work by Smith and Smith [3,4,5,6] on database abstractions. It discusses problems with their approach, justifies changes made to their syntax and semantics to overcome those problems, and allows for a broader class of interpretations of what they term "relational invariant" properties of hierarchies of relations. Several possible implementations of aggregation and generalization abstractions are considered before the actual implementation based on the relational database system SUR[7] is discussed. The implementation is not proved correct formally but is demonstrated using examples in a SITBOL program designed to run on a DEC-10 computer.

TABLE OF CONTENTS

CHAPTER PAGE

1. INTRODUCTION, OVERVIEW, AND ORGANIZATION..... 1

    1.1 Data Models
    1.2 Abstractions

2. AGGREGATION AND GENERALIZATION..... 8

    2.1 Aggregation
    2.2 Generalization
    2.3 Smith and Smith's Syntax for Aggregation and Generalization and the Problems Associated With the Syntax
    2.4 The Proposed Syntax
        2.4.1 The proposed syntax for aggregation
        2.4.2 Construction of an aggregate relation
        2.4.3 The properties of well-defined aggregate
        2.4.4 Maintaining relational invariants in aggregation
        2.4.5 The proposed syntax for generalization and the construction of an example combining aggregation and generalization
        2.4.6 The properties of well-defined generalization
        2.4.7 Maintaining relational invariants in generalization
    2.5 (3,3) Normal Form

3. IMPLEMENTATION..... 46

    3.1 Relational Invariants and Integrity Constraints
    3.2 Additions and Justifications to Smith and Smith's Relational Invariants
    3.3 Methods of Maintaining Relational Invariants
    3.4 Syntax of Maintaining Relational Invariants
    3.5 SUR, SURLY, and a Revised SURLY Syntax
    3.6 Some Considerations in Actual Implementation

4. CONCLUSION..... 68

LIST OF REFERENCE..... 70

VITA..... 72

LIST OF TABLES

TABLE PAGE

1. Column headings of E-R approach..... 19
2. A relation RESERVATION in tabular form..... 28
3. A relation TOURIST in tabular form..... 28
4. A relation FACILITY in tabular form..... 28
5. Maintenance of the relational invariants in aggregation plane during update operations..... 32
6. Maintenance of the relational invariants in generalization plane during update operations..... 37
7. A relation ENROLLMENT in tabular form..... 43
8. The syntax of the subset of SURLY..... 63

LIST OF FIGURES

FIGURE		PAGE
1.	Aggregate relation RESERVATION.....	9
2.	Aggregate and generalization relation RESERVATION..	11
3.	Smith and Smith's syntax of relation FACILITY.....	14
4.	The graphical form from [5] of a relation EMPLOYEE.	15
5.	A relation RA with the hierarchy involving more than two levels.....	16
6.	The relation RA in graphical form from [4].....	20
7.	The relation WORK_FOR.....	22
8.	The proposed graphical form of aggregate relation RESERVATION.....	25
9.	An unnormalized relation RESERVATION.....	39
10.	An normalization relation RESERVATION.....	40
11.	The relation ENROLLMENT.....	44
12.	The relation GRADUATE_IN_STATE.....	44
13.	The relation GRADUATE_OUT_STATE.....	44
14.	The relation UNDER_GRAD_IN_STATE.....	44
15.	The relation UNDER_GRAD_OUT_STATE.....	44
16.	The relation BICYCLE.....	51
17.	The syntax of the subset of SURLY.....	63

CHAPTER I

INTRODUCTION, OVERVIEW, AND ORGANIZATION

1.1 Data Models.

A Data Base Management System (DBMS) is an integrated and shared repository of data. Logically, most DBMS offer a three-level representation independence (via abstractions and mapping functions): an external view, a conceptual level database, and a physical level database. An external view is the way the application programmer chooses to view the data; the conceptual level database is the centralized community view of the database; and the physical level database is the actual implementation of the conceptual database. "Data independence" is the immunity of one level from changes made in the representation of the neighboring level. Data independence is important because it allows the user to deal only with the abstract external view without concern for the detailed community view (conceptual level) or the physical data storage. It enhances the usability of the data, and it also stabilizes the database. Usually, several users' views co-exist in the same DBMS. In order to unify all the user's views and to map them into a physical level, a "data model" is used. There are three data models in wide use today: the relational, the hierarchical, and the network data models. In addition, the entity relationship

data model, a higher-level data model built on top of the relational model, is gaining in importance. The difference among these models is in the way that they allow the user to view and manipulate relations. Some basic terminology used in this paper is introduced and defined briefly in this section as the different models are briefy introduced.

The relational data model describes the real world by entity sets and relationships. An entity is an object which is uniquely distinquishable; a collection of entities comprises an entity set. A relationship is a special type of entity which describes the relationships among entity sets. A relation is a table (a flat one where all fields are "simple" or "atomic"). Relations can represent entity sets or relationships. Relations are viewed as mathematical relations and hence relation theory and set theory can be applied to them. In this approach, rows of tables (relations) are referred to as tuples. An attribute is a function which provides mapping from a relation to a column. A key is a minimal set of attributes that can uniquely identify the tuples in a relation. A relation can have more than one key, with each key being referred to as a candidate key. Typically one primary key is chosed from the set of candidate keys. The primary key is often used to indicate that the primary physical structure of the relation is assigned with respect to the primary key attributes. A domain is a set of values appearing in a given column. The relational approach has the advantages of a sound theoretical foundation, a simple

structure, a uniform operator set, and the ability to achieve a high degree of data independence. It has the disadvantage that it loses some semantic information in modeling the real world because it uses only flat tables.

The network data model is based on graphs. In contrast to the relational model which uses only one primary construct, the relation, the network data model uses two information bearing constructs: the "record type" (like a relation) and the "set" (or link) which represents a 1: many binary relation. Many: many n-ary relations are represented using collections of "sets".

In the hierarchical data model, a database is viewed as a tree structure. The data model is a special case of network approach with the restricted capability that a node may have at most one parent. The criticism of the network and the hierarchical approach is that the data structure (graph or tree) tends to interfere, causing users to model certain data in an unnatural way. The users in these models also need to know certain physical aspects (data structure or access method) of the physical level database.

The entity-relationship data model tries to unify the above three data models. It represents the real world by both relationships and entities and also tries to capture some important semantic information about the real world in the modelling process. It has received considerable attention recently (two international conferences on the entity-

relationship data model have been held), but no commercial systems are based on this data model yet. The approach to aggregation in Smith and Smith is similar to the entity-relationship data model, but the entity-relationship model does not yet incorporate the idea of generalization.

This paper is based on the work of Smith and Smith on database abstractions. The approach of database abstractions is similar to the entity-relationship model in that a logical hierarchy is built on top of a relational data model to capture semantic information that the relational data model has missed.

1.2 Abstractions.

An abstraction is a deliberate omission of certain details which are irrelevant in a certain situation. It is based on information hiding which helps control the complexity of problems for the users. There are many types of abstractions, some of which are: functional abstraction (or implementation abstraction), architectural abstraction, statistical aggregation, behavioral abstraction, component aggregation, and generalization. Functional abstraction is the separation of what a construct does (its external specification) from how it does it (its internal, implementation detail). In higher level programming languages, users are only concerned with constructs such as IF...THEN...ELSE, user defined data types, and files rather than assembly level instructions, machine level data types and pages. Each higher-level

operator is functionally mapped into a schema or organized collection of corresponding machine instructions. The implementation is largely hidden from the user. It is both unnecessary and undesirable for the user to be aware of the implementation detail. For example, no high level language allows users to allocate his own registers. An important subclass of functional abstraction is architectural abstraction, specifically the common 3-level database abstraction. In 3-level database abstraction, external users' views are the abstraction of conceptual database, and the conceptual database is the abstraction of physical database. Certain details in a conceptual database are not of interest to the users. The users at this level are only interested in working with higher-level entities or relations and operators. For example, in a reservation system, the user wishes to think in terms of "reservations, facilities, et cetera" not records or bytes. The conceptual database is then mapped into the physical database.

Statistical aggregation deals with operations like SUM, COUNT, AVERAGE, et cetera, which statistically summarize certain aspects of data.

In behavioral abstraction, an abstract object is defined in terms of its associated abstract operations [6]. In a dynamic environment in which the history of operations is important, behavioral abstraction is very useful. To illustrate behavioral abstraction, for instance, if the sequence

of purchasing certain items is important, one such BUY operation can be uniquely identified by the previous BUY operation; the second BUY can be identified by the first BUY, the third by the second, the last by the one previous to it, and so on. In this way, an item can be identified by operations on it.

Finally, component aggregation and generalization, which are the types of abstraction considered in this paper deal specifically with the conceptual database. Abstraction here allows users to concentrate on details which are relevant to the situation and to ignore those which are not. The method of achieving abstractions of aggregation and generalization is to decompose the conceptual model into a hierarchy of abstractions. Such an abstraction hierarchy has the capability for different users to access the model at different levels of abstraction. For example, the general manager of a company may access global information without needing much detail, while the department managers may access the detailed information concerning respective departments. Just as in 3-level database abstraction, such a hierarchy can enhance the stability of the model by the independence between hierarchical levels. A change in a detail which is ignored at higher levels will leave higher levels unaffected. Component aggregation refers to an abstraction in which a relationship between objects is regarded as a higher level object [4]. It is the "part-whole" abstraction. It is

related to the way objects are formed from components (in the entity-relationship model, entities are formed from attributes, relationships are formed from entities) and also to database normalization theory. Generalization refers to an abstraction in which a set of similar objects is regarded as a generic object [4]. It is related to generalization or class hierarchies (IS-A hierarchies) commonly used in object oriented programming languages like SMALLTALK (August '81 issue of Byte is dedicated to SMALLTALK) [9].

This paper discusses details of component aggregation (hereafter referred to as aggregation) and generalization abstractions in Chapter 2. Chapter 3 discusses an implementation which supports these abstractions.

CHAPTER 2

AGGREGATION AND GENERALIZATION

Aggregation and generalization can enhance the stability of the database as mentioned earlier; they also provide a structual advantage over the regular relational data model structures as they organize the flat files into a controllable hierarchy as will be discussed in Section 2.1.

In this chapter, the notions of Smith and Smith's aggregation and generalization are introduced first. The syntax they use to represent the structures of aggregation and generalization follows. The potential problems associated with their syntax and a proposed solution are discussed. Their relational invariants are introduced along with aggregation and generalization. Finally, normalization is discussed.

2.1 Aggregation.

Aggregation and generalization were proposed by Smith and Smith [3,4] to add a higher order classification scheme to organize a (possibly large) set of relations. Because generalization abstraction can be built on top of aggregation abstraction, the discussion will first concentrate on aggregation abstraction.

Aggregation abstraction is defined by Smith and Smith as "an abstraction in which a relationship between objects is regarded as a higher level object." The main idea is that embedded object can be removed from parent objects.

In this way, it is similar to normalization, which will be discussed in Section 2.5.

In graphical form, for example, a RESERVATION system can be organized as shown in Figure 1

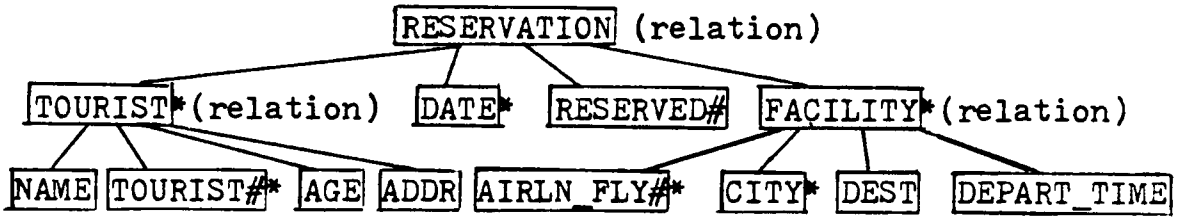

Figure 1. Aggregate relation RESERVATION

where "*" indicates that a certain object is a key or an element of a key and where FACILITY only includes airlines. The same RESERVATION system in the regular relational structure is a flat file (table), RESERVATION (NAME, TOURIST#*, AGE, ADDR, AIRLN_FLY#*, CITY*, DEST, DEPART_TIME); This relation needs to be normalized (details of normalization will be discussed in Section 2.5) into three relations (three flat tables): RESERVATION (TOURIST#*, DATE*, RESERVED#, AIRLN_FLY#*, CITY*); TOURIST (NAME, TOURIST#, AGE, ADDR); FACILITY (AIRLN_FLY#*, CITY*, DEST, DEPART_TIME); Unfortunately, there is no higher level organization to organize the three relations together. In a complex database, it is extremely difficult for the user to see or understand the overall structure of the database. Smith and Smith's method gets around this problem, as we shall see.

2.2 Generalization.

As stated earlier, generalization refers to an abstraction in which a set of similar objects is regarded as a generic object. The idea is very similar to the notion of class and subclass in abstract data types in languages like ADA, SMALLTALK, et cetera. For instance, AIRLINE, HOTEL, and RESTAURANT can be generalized as FACILITY; their differences are ignored here. Aggregate and generic objects in generalization are in fact still relations. The immediate descendants of a generic object in generalization are partitioned into mutually exclusive classes. If relation A is a generalization of relation B, (B is a subclass of A), then this can be denoted as "B isa A", meaning B is a subclass type of A. A mutually exclusive group of generic objects sharing a common parent is called a cluster [4]. For instance, AIRLINE, HOTEL, and RESTAURANT form a cluster since they are mutually exclusive and they also share a common parent, FACILITY. Smith and Smith in [4] give an example of a generic hierarchy with multiple clustering where a generic object vehicle based on medium (land vehicle, water vehicle....) and based on propulsion category (man powered, motorized vehicle....).

Now, the structure of reservation is redesigned by adding generalization to the diagram as shown in the next page (Figure 2). FACILITY is the generalization of AIRLINE, HOTEL, and RESTAURANT; TOURIST is the generalization of OVERSEAS_TOURIST and DOMESTIC_TOURIST. OVERSEAS_FLYING is the aggregation of AIRLINE and OVERSEAS_TOURIST.

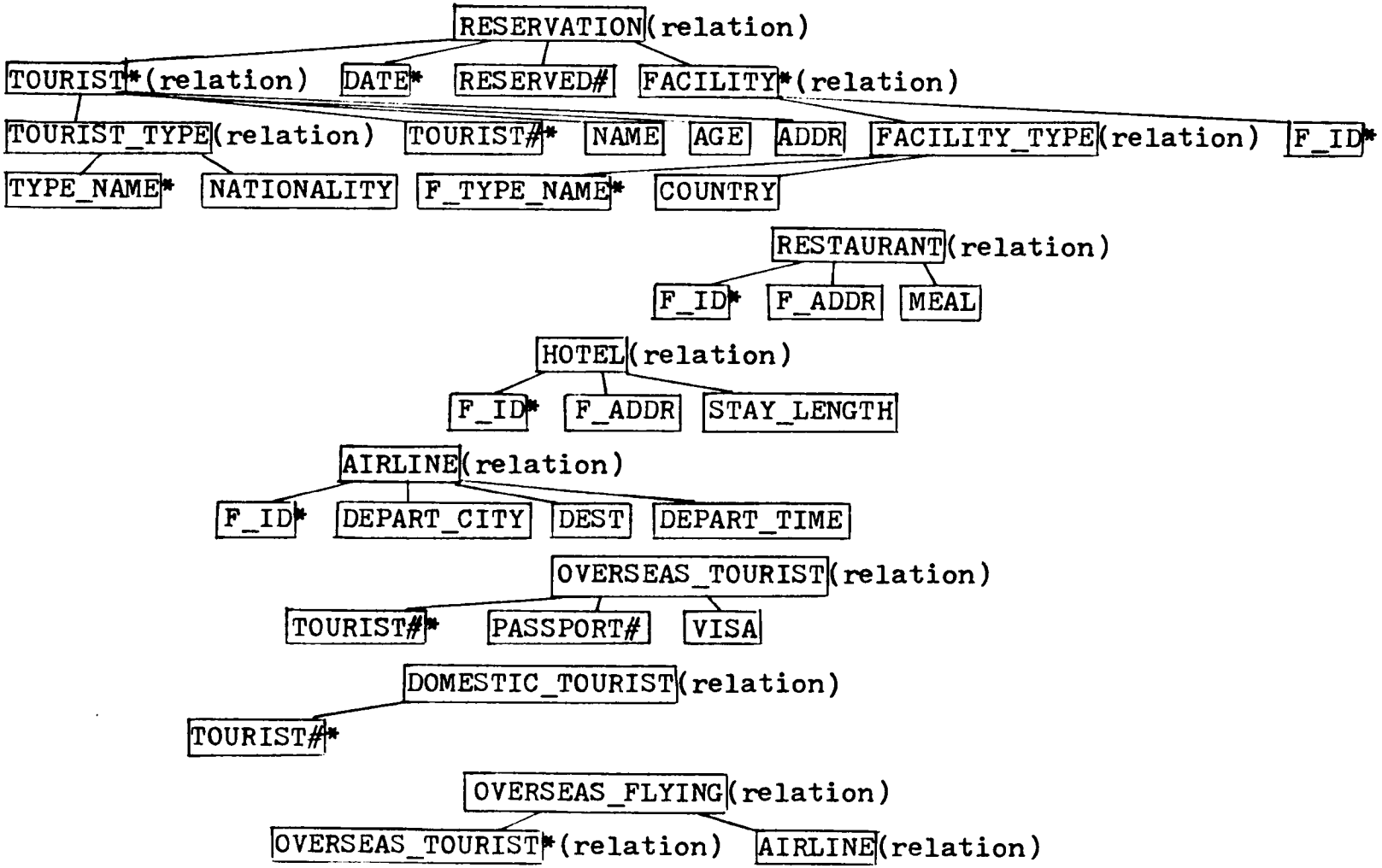

Figure 2. Aggregate and generalization relation RESERVATION

2.3 Smith and Smith's Syntax for Aggregation and Generalization and the Problems Associated With the Syntax.

This section is to point out that Smith and Smith's graphical notation and linear syntax from [5] suffer from the confusion of assigning names in certain situation and there are four methods of solving this problem including Smith and Smith's syntax in [4]. The advantages and disadvantages of each method are to be discussed including two problems in Smith and Smith's syntax in [4].

Aggregation and generalization were proposed by Smith and Smith [3, 4] to add a higher order classification scheme to organize a (possibly large) set of relations. In these papers both a graphical representation and a syntatic form to describe abstractions are used. Smith and Smith's syntax is PASCAL-like. An example of the above FACILITY relation is as follows:

```
var facility
   generic
       f_type_name = (airline, hotel, restaurant)
   of
   aggregate [f_id]
       f_id:  facility_id#;
       f_type_name:  KEY facility_type;
end
```

The relation name "facility" is followed by its generic subtypes airline, hotel, and restaurant, all of which are "sub-types" or "sub-classes" of facility, then by its aggregate attributes. The key of facility is f_id and is represented in brackets. An attribute is followed by either the name of an atomic domain (where domain specification will be discussed further in this section and Section 2.4) or a relation name. If it is a relation name (such as facility_type in the above example), then this attribute (set) is a key of this relation (such as f_type_name); the relation name (facility_type) preceded by the word "KEY". In Smith and Smith's papers, key raising is the approach in representing hierarchy. In Smith and Smith when there is a relational hierarchy, the key of the lower-level relation is used as an attribute of the higher-level relation. It is called the key-raising method. Smith and Smith suggest that all object names in a relation definition must be natural language nouns. This includes both aggregate names and generic object names in generalization. Smith and Smith maintain that this method of modeling the real world will make it easier for users to interact with the database. However, this criterion is at best a guide since complex relationships used nounphrases and nounphrases can encode anything in the real world situation.

In graphical form the above example can be shown as in Figure 3.

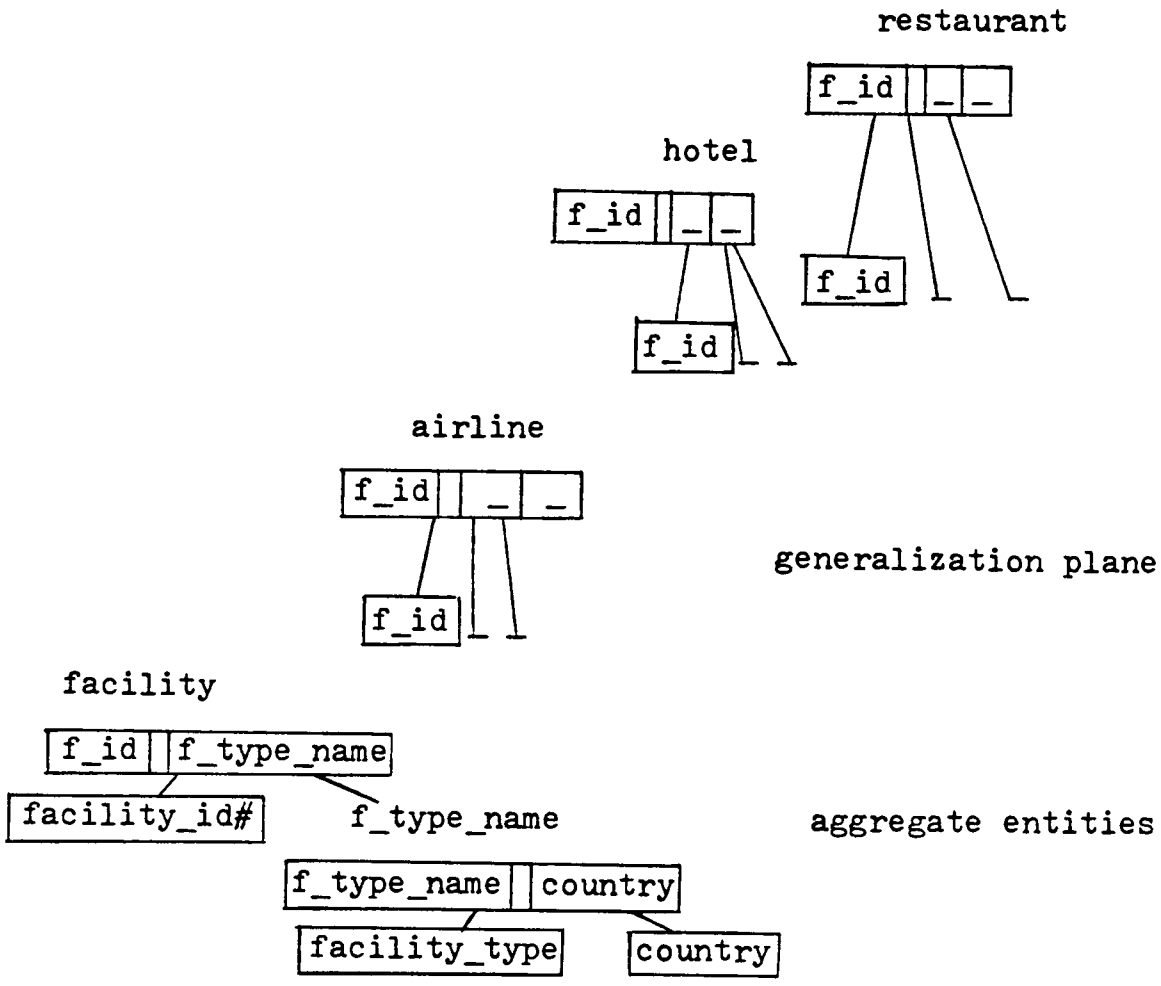

Figure 3. Smith and Smith's syntax of relation FACILITY

In the graphical notation, within each relation, key attributes are separated from non-key attributes by a double bar.

In [5], Smith and Smith propose a different linear syntax which does not distinguish attributes and domains. Both the syntax from [5] and the graphical representation suffer from a confusion of how to assign names. Consider an example as shown in Figure 4

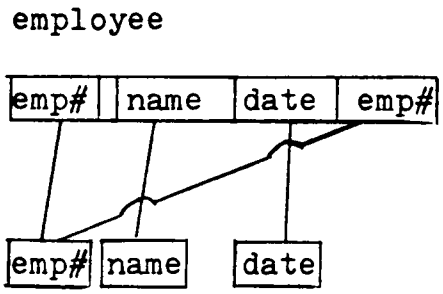

Figure 4. The graphical form from [5] of a relation EMPLOYEE

The syntactical form of the relation EMPLOYEE is as follows:

```
def employee
   object[emp#]
      emp#; /*of employee*/
      name;
      date;
      emp#; /*of boss*/
end
```

Clearly, we wish to distinguish between emp# domain used as employee #'s and as the boss id's. But neither of these representations handle this problem. The problem is that there is no way to distinguish the different roles played by objects from the same domain.

A potential problem is associated with the hierarchy involving more than two levels of relations in Smith and Smith's syntax. For example, the following relation (Figure 5) RA has attribute A1 and relation RB as its components. RB has components of relation RC and attribute A2; RC has attributes A31 and A32. A1 is the key of RA, RC is the key of RB, and A31 is the key of RC.

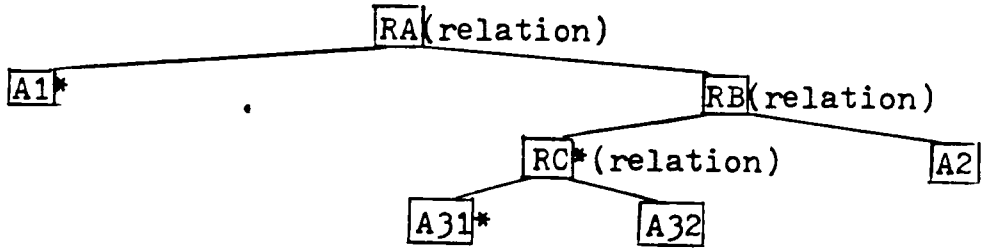

Figure 5. A relation RA with the hierarchy involving more than two levels

There are several alternative methods of describing this hierarchy as considered below.

(1) Simple attributes.

Declare all attributes such as

```
RELATION RA (*A1:     CHAR 10,
             A31:     CHAR  7,
             A32:     CHAR  6,
             A2:      CHAR  5);
```

This notation is easy to implement, but hierarchy structure is lost here. Furthermore, the relation is in 1NF and suffers from undesirable insert, delete, or modify anomolies when the embedded component RB or RC are operated on.

(2) Using KEY_OF.

```
RELATION RA (*A1:     CHAR 10,
             RB:      key of RB);
```

This abstract key method, reference [Thompson, personal communication], has the property of data independence--key_of dependent relation can be changed without affecting the higher-level definition. Also, the hierarchical structure of the data is maintained without introducing possibly many key attributes raised from the lower level attributes or relations to specify its tuples as would in the following methods (3) and (4). The problem with this notation is that the underlying object can not be directly specified. It is also

not clear how to write the syntax of the insert command or what the form of the print should be.

(3) Dot notation method. Dot notation can be used indicating next lower level in the hierarchy. The relation might look like as follows:

```
RELATION RA (A1:          CHAR 10,
             RB.RC.A31:   CHAR 5);
```

A problem could arise if the user forgets to specify part of the key. For example, if the correct relation is

```
RELATION RA (*A1:          CHAR 10,
             RB.RC.A31:    CHAR  5,
             RB.RC.A32:    CHAR  6);
```

but the user specifies the relation as

```
RELATION RA (*A1:          CHAR 10,
             RB.RC.A31:    CHAR  5);
```

If this happens, then this relation is not specified correctly. However, it is felt that this disadvantage is outweighted by the advantage

that the hierarchical structure is preserved in this notation.

Two points should be made about the dot notation convention. First, it is very similar to the convention of hierarchical naming of columns and collection of columns in the entity-relationship approach as shown in Table 1.

Table 1. Column headings of E-R approach

WOEKFOR

BOSS SSN	MAID SSN	TIME
-	-	-
-	-	-

The difference is that role-path names are substituted for complex column headings. The second point is the hierarchy is still present in the attribute names in the dot notation. BOSS.SSN and MAID.SSN show the logical hierarchy in the names. FACILITY.AIRLN_FLY# and FACILITY.CITY show that these attributes come from a common underlying object. A straight forward extension, not pursued in this paper, is to allow the user to have abbreviating

conventions for long attribute names. Thus in a name like x.y, x can be omitted if no other z.y is used in the context of the relation; x.z or x.y can be abbreviated to x to capture the x entity without forcing the user to distinguish all of x's key attributes. This is exceedingly useful where there is no role confusion. An attribute can be identified by the relation name and enough of the access path to avoid ambighity. A similar mechanism exists for referencing PL/I aggregate names.

(4) Strictly following Smith and Smith's syntax in [4], the graphical form of the relation should look like as shown in Figure 6

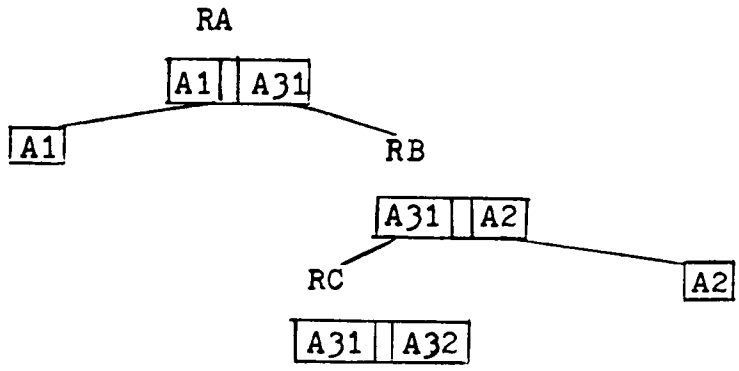

Figure 6. The relation RA in graphical form from [4]

The syntactical form from [4] of relation RA is as follows:

```
var RA
  aggregate[A1]
     A1: fixed;
     A-RB: key RB;
end

var RB
   aggregate [A-RC]
      A-RC:  KEY RC;
      A2:    fixed;
end

var RC
   aggregate [A31]
      A31: fixed;
      A32: fixed;
end
```

This linear syntax has 2 advantages--it captures the idea of imposing a hierarchy on a collection of relations and it appears to be representationally adequate since it uses a discipline of naming to distinguish between attribute role and underlying domain/relation by using local attribute names such as A-RB in RA and A-RC in RB.

There are two problems with Smith and Smith's

representationally adequate syntax for aggregation. First, the user is not constrained in his choice of names for attributes. Since he is locally naming domains and relations, he can distinguish their various roles, but he can use whatever names he wants to. Second, when the aggregate objects are viewed as relations, there is nothing in the attribute names to group attributes that are raised from lower level entities, so that they are treated as object identifiers. So attributes AIRLN_FLY# and CITY in RESERVATION are not treated togather as FACILITY identifiers (Figure 1).

The problem existing in method (2) as the underlying object can not be directly specified, also exists in this method aside from those mentioned above.

Let's consider another relation example such as WORK_FOR between bosses and maids as shown in Figure 7.

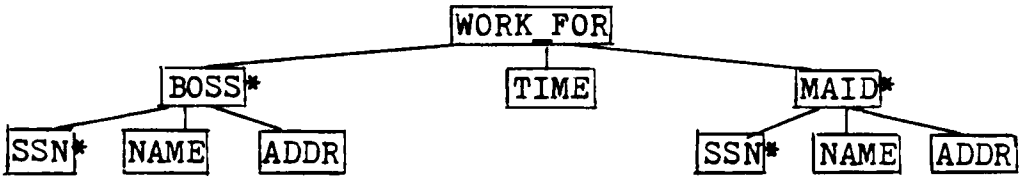

Figure 7. The relation WORK_FOR

(BOSS, MAID) is the key of relation WORK_FOR. When SSN is the key for respective relation BOSS or MAID, there are duplicate keys for relation WORK_FOR if Smith and Smith's graphical notation and linear syntax in [5] is used as discussed at the beginning of this section. In dealing with this problem, using method (3), dot notation, we can represent the relation as:

```
RELATION WORK_FOR (*BOSS.SSN:     CHAR 11,
                   TIME:          NUM  10,
                  *MAID.SSN:      CHAR 11);
```

In this representation, the local attribute names, BOSS.SSN and MAID.SSN, are treated with a systematic method, and it also avoids the domain specification in the syntax. If an improved syntax is used as described in next section, we can simply tell that BOSS.SSN and MAID.SSN are drawn from the same domain, SSN but with different attribute names. In this case, we need to declare DOMAIN SSN CHAR 11.

Both method (2) and (3) provide improvements over method (4). Method (3) provides the easy way of creating local variable names. Method (3) in principle is consistent with Smith and Smith's key raising method and preserves hierarchy structure in this representation of relation hierarchies. Method (2) is more consistent with Smith and Smith's syntax, but in actual implementation, method (3)

reflects the hierarchical structure better. Therefore, method (3) is the best choice even though it has redundancy in its notation in that the lower level key is redundently specified in each level of relations such as AIRLN_FLY# and CITY in the above example.

2.4 The Proposed Syntax.

2.4.1 The proposed syntax for aggregation. It should be noted that some of the notations used for aggregation in this section are not in final form and will be refined later in the paper after discussions of generalization.

A reservation system can be described as shown in Figure 8.

The syntactical form of relation RESERVATION is as follows:

```
RELATION RESERVATION(*TOURIST.TOURIST#:      CHAR 11,
                     *DATE:                  CHAR 8,
                      RESERVED#:             CHAR 2,
                     *FACILITY.AIRLN_FLY#:   CHAR 6,
                     *FACILITY.CITY:         CHAR 10);
```

PL/I-like syntax is used here instead of Smith and Smith's PASCAL-like syntax because of consistent consideration as the underlying DBMS uses PL/I-like syntax. The "*" symbol is placed in front of attributes to indicate that it is an

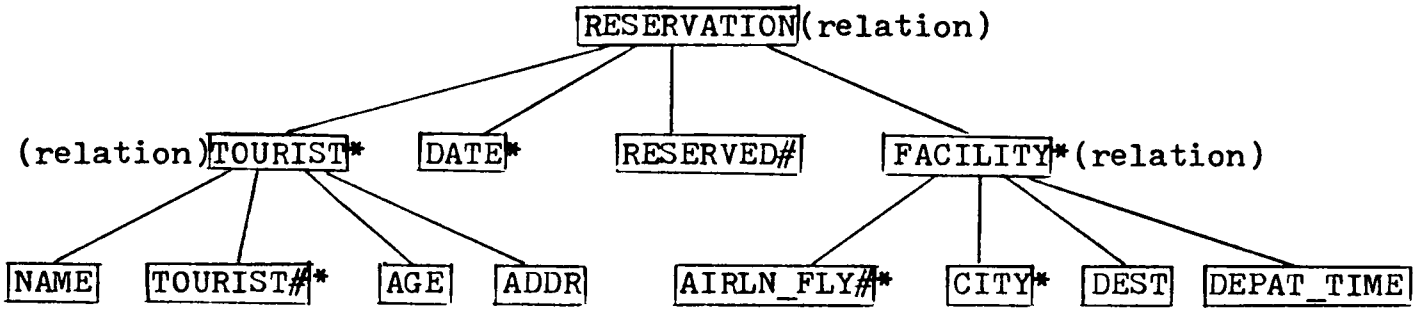

Figure 8. The proposed graphical form of aggregate relation RESERVATION

element of a primary key. In this example, the primary key of RESERVATION is the attribute set [TOURIST#, DATE, AIRLIN_FLY#, CITY]. In this notation, an attribute is followed by a colon ":" and then either by a relation or a simple domain. The BNF notation is presented in Section 3.5.

One possible improvement in the syntax is to specify the above relation RESERVATION as

```
RELATION RESERVATION(*TOURIST.TOURIST#:     TOURIST,
                     *DATE:                 DATE,
                      RESERVED#:            RESERVED#
                     *FACILITY.AIRLN_FLY#:  FACILITY
                     *FACILITY.CITY:        CITY);
```

and

```
DOMAIN TOURIST#     CHAR 11
DOMAIN DATE         CHAR  8
DOMAIN RESERVED#    CHAR  2
DOMAIN CITY         CHAR 10;
```

In this way, the user does not need to specify the constraint of domain in the relation definition when he specifies the relation.

2.4.2 Construction of an aggregate relation. Based on the discussion so far, an aggregate relation RESERVATION can be built as follows.

```
RELATION RESERVATION (*TOURIST.TOURIST#:      CHAR 11,
                      *DATE:                  CHAR  8,
                       RESERVED#:             CHAR  2,
                      *FACILITY.AIRLN_FLY#:   CHAR  6,
                      *FACILITY.CITY:         CHAR 10);
RELATION TOURIST      ( NAME:                 CHAR 20,
                      *TOURIST#:              CHAR 11,
                       AGE:                   NUM   3,
                       ADDR:                  CHAR 50);
RELATION FACILITY     (*AIRLN_FLY#:           CHAR  6,
                      *CITY:                  CHAR 10,
                       DEST:                  CHAR 10,
                       DEPART_TIME:           CHAR  7);
```

In tabular form, it can be shown as in Table 2, 3, and 4.

Table 2. A relation RESERVATION in tabular form

RESERVATION

TOURIST.TOURIST#	DATE	RESERVED#	FACILITY.AIRLN_FLY#	FACILITY.CITY
487-20-1199	7-8-81	2	EA211	CHICAGO
487-20-1199	7-8-81	2	DA101	KNOXVILLE
523-15-4231	8-5-80	3	EA111	KNOXVILLE

Table 3. A relation TOURIST in tabular form

TOURIST

TOURIST.NAME	TOURIST.TOURIST#	TOURIST.AGE	TOURIST.ADDR
C.J.SMITH	487-20-1199	50	118,17TH ST,KNOX
RANDY JONES	533-15-4231	65	20,8TH ST.KNOX
ANN WILLIAMS	321-01-1155	30	1199 RODERICK RD.KNOX
TONY LYNN	623-17-1180	18	155 BRIDGEWATER,KNOX

Table 4. A relation FACILITY in tabular form

FACILITY

FACILITY.AIRLN_FLY#	FACILITY.CITY	FACILITY.DEST	FACILITY.DEPART_TIME
EA211	CHICAGO	LOS ANGELES	11:30
DA101	KNOX	CHICAGO	8:15PM
EA211	KNOX	DALLAS	9:05AM
AA693	KNOX	ATLANTA	12:15Am

(1) Here, RESERVATION, TOURIST, and FACILITY are all natural language nouns; and RESERVATION, TOURIST, and FACILITY express aggregation abstractions about the real world.

(2) TOURIST.TOURIST#, DATE, RESERVED#, FACILITY.ARILN_FLY#, and FACILITY.CITY are attributes of RESERVATION; TOURIST.NAME, TOURIST.TOURIST#, TOURIST.AGE, and TOURIST.ADDR are attributes of TOURIST; FACILITY.AIRLN_FLY#, FACILITY.CITY, FACILITY.DEST, and DEPART_TIME are attributes of FACILITY.

(3) FACILITY.AIRLN_FLY# and FACILITY.CITY determine each FACILITY; TOURIST.TOURIST# which is a social security number or an assigned ssn-like number determines TOURIST. TOURIST.TOURIST#, the primary key of relation RESERVATION, DATE, and FACILITY.AIRLN_FLY# and FACILITY.CITY, the primary key of relation FACILITY, determine each RESERVATION.

The (1) above satisfies the condition that all object names in a relation definition be natural language nouns [4]. The (2) and (3) satisfies the following two conditions in 2.4.3. Therefore, it can be said that RESERVATION is well defined.

2.4.3 The properties of well-defined aggregate. The syntax of well-defined aggregates was introduced in the

previous section. In this brief section, the semantic properties of aggregates are considered. Smith and Smith state that a well-defined aggregate relation must satisfy the following properties [3]:

(1) All tuples in a relation will have unique keys, and

(2) If the types of component s of tuple t in relation r is key r' then t.s is the key of some tuple in relation r'.

The second condition is related to the key-raising property in that a tuple in an aggregate parent must refer to a corresponding tuple in any aggregate children via the key. These two conditions are called aggregate relational invariants by Smith and Smith.

To maintain the relational invariants, certain restriction must be placed on update operations. Directly taken from Smith and Smith [3], the following table is a set of restrictions in update operations. INSERT(R,t), DELETE (R,k), and MODIFY(R,k,t) are added to the table to clarify the relationships between parents and children. r refers to a relation, k is the key of r, and t is a tuple of r.

UPDATE OPERATION	/ RESTRICTION
insert(r,t)	1) the key of t contains no blanks (nulls) and is unique within r. 2) all tuples referenced (via their keys) by t exist.
delete(r,k)	2) the tuple with key k is not referenced by a tuple in another relation.
modify(r,k,t)	1) the components forming the key k are not

modified.
2) all tuples referenced by t exist.

The two conditions mentioned above and the three conditions to be mentioned in Section 2.4.6 were called "relational invariants" by Smith and Smith because they must be satisfied by every relation in a relational hierarchy irrespective of the kind of abstract object the relation represents. The relational invariants are thought of as constraints that a relation must maintain to represent an "abstract object".

The invariant constraints are true before any update operation (INSERT, DELETE, and MODIFY) and true after the operation is completed. When a user updates a relation, a side effect may be to update the related relations paR, caR, pgR, cgR (paR: parent(s) in the aggregation plane of R; caR: child(ren) in the aggregation plane of R; pgR: parent (s) in the generalization plane of R; cgR: child(ren) in the generalization plane of R). In this way, a single user initiated update operation can trigger additional update operations which propagate along both aggregation and generalization planes.

2.4.4 Maintaining relational invariants in aggregation. The following table (Table 5) is taken from Smith and Smith [4] and summarizes the methods they suggest for maintaining the relational invariants during update operations. The numerical numbers (1 and 2) in the table indicating

Table 5. Maintenance of the relational invariants in aggregation plane during update operations

Invariant	possible viclation after insertion, deletion, and modification	method for correcting violation
INSERT(R, t)		
1	t occurs twice in R.	delete either occurence of t
2	t references a non-existent tuple in some caR relation R'	insert a tuple in R' with the appropriate key values and blanks elsewhere
DELETE(R, k)		
1	none	
2	a tuple in some paR relation references a non-existent R-tuple.	a) modify the tuple so that the reference is replaced by a blank (only possible if reference is not part of tuple's key). b) delete the tuple.
MODIFY(R, k,t)		
1	a tuple in some paR relation references a non-existent R-tuple.	modify the tuple so that it references t.
2	(key domain modification) t references a non-existent tuple in scme caR relation R'.	insert a tuple in R' with the appropriate key values and blanks elsewhere.

correspondence with the above aggregation properties. One addition to the table is the label of update operations of INSERT (R, t), DELETE (R, t), and MODIFY (R, k, t). Three other changes are necessary in MODIFY: a) the order of (1) and (2) is switched for the consistency with the numerical numbers (1) and (2) in above properties, b) the place of "key domain modification" should be put to the first possible violation condition, and c) the first possible violation condition should be (2) to replace "none"

2.4.5 The proposed syntax for generalization and the construction of an example combining aggregation and generalization. After presenting the syntax for aggregation in Section 2.4.1, in this section, the syntax for generalization only needs to be added on top of the aggregation syntax. To add generalization, a list of parameters enclosed in parentheses is added. The parameters are the entities that are generalized into this relation. Using generic structure, we can combine both aggregation and generalization into one structure as follows:

```
RELATION RESERVATION (*TOURIST.TOURIST#:    CHAR 11,
                      *DATE:                CHAR  8,
                       RESERVED#:           CHAR  2,
                      *FACILITY.F_ID:       CHAR 15);
RELATION TOURIST (*TOURIST#:                CHAR 11,
                  NAME:                     CHAR 20,
                  AGE:                      NUM   3,
```

```
                    ADDR:                    CHAR 50,
                    TOURIST_TYPE.TYPE_NAME:  CHAR 21)
                   (OVERSEAS_TOURIST, DOMESTIC_TOURIST);
```

The second list of parameters here indicates generalization.

```
RELATION FACILITY (FACILITY_TYPE.F_TYPE_NAME:   CHAR 10,
                   *F_ID:                       CHAR 15)
                  (AIRLINE, HOTEL, RESTAURANT);
```

In the relation FACILITY above, F_ID is the key and can be used to identify a particular AIRLINE, HOTEL, or RESTAURANT. For instance, Sheraton_Greenway_Phoenix is the key (F_ID) of a tuple.

```
RELATION AIRLINE (*F_ID:         CHAR 15,
                  DEPART_CITY:   CHAR 10,
                  DEST:          CHAR 10,
                  DEPART_TIME:   CHAR  7);

RELATION HOTEL (*F_ID:           CHAR 15,
                F_ADDR:          CHAR 50,
                STAY_LENGTH:     CHAR  3);

RELATION RESTAURANT (*F_ID:      CHAR 15,
                     F_ADDR:     CHAR 50,
                     MEAL:       CHAR 6);
```

```
RELATION TOURIST_TYPE (*TYPE_NAME:     CHAR 21,
                       NATIONALITY:    CHAR 15);
RELATION FACILITY_TYPE (*F_TYPE_NAME:  CHAR 10,
                        COUNTRY:       CHAR 15);
RELATION OVERSEAS_FLYING (*OVERSEAS_TOURIST.TOURIST#: CHAR 11,
                          *AIRLINE.F_ID:     CHAR 15);
RELATION OVERSEAS_TOURIST (*TOURIST#  CHAR 11,
                           PASSPORT#: CHAR 12,
                           VISA:      CHAR 50);
RELATION DOMESTIC_TOURIST (*TOURIST#:  CHAR 11);
```

Here, the generic objects AIRLINE, RESTAURANT, HOTEL, DOMESTIC_TOURIST, and OVERSEAS_TOURIST are declared just as the regular objects. AIRLINE, HOTEL, and RESTAURANT are associated with FACILITY; FACILITY is their common parent. The generic structure is also true among the relationships DOMESTIC_TOURIST, OVERSEAS_TOURIST and TOURIST. The relation TOURIST_TYPE describes OVERSEAS_TOURIST and DOMESTIC_TOURIST when they are thought of as generic objects. From the relation TOURIST we can refer to the subclass in which an individual TOURIST belongs by way of TYPE_NAME. The similar structure also holds in the relation FACILITY.

2.4.6 The properties of well-defined generalization. There are three semantic conditions necessary for a relation definition to specify a generalization [4].

(1) The objects in the generalization plane have a common parent; each object is a subclass of the common parent. For example, FACILITY is the common parent of AIRLINE, HOTEL, or RESTAURANT.

(2) Each individual object in a generalization plane contains all individuals in the common parent classified as belonging to this object. For instance, relation HOTEL contains motel, inn, et cetera, classified as public places providing lodging and possibly board and other services.

(3) Clusters contain mutually exclusive classes.

We say that a relation is well-defined if its definition satisfies the above three semantic requirements and the two requirements in aggregation.

2.4.7 Maintaining relational invariants in generalization. In the following table, an image domain is simply the domain in a relation which contains the name of a descendant relation [4]. For example, the domain which contains airline, hotel, and restaurant as its values of relation FACILITY is the image domain for the decendant relations AIRLINE, HOTEL, and RESTAURANT.

As in the table for maintaining relational aggregation, a table describes how Smith and Smith [4] suggest that update operations in maintaining relational invariants in generalization. Here the numerical number (1, 2, or 3) refers to the number in the above section, 2.4.6. R refers to a relation,

Table 6. Maintenance of the relational invariants in generalization plane during update operations.

Invariant	possible violation after insertion, deletion, and modification.	method for correcting violation.
INSERT(R, t)		
1	t does not have a parent image in some pgR relation R'.	if a tuple with key k already occurs in R' then modify this tuple so that it becomes the parent image of t, otherwise insert a parent image in R'(use blanks where values are unknown).
2	t does not have a required child image in some cgR relation R'.	insert a child image in R' (use blanks where values are unknown).
3	NONE.	
DELETE(R, k)		
1	a tuple in some cgR relation has no parent image in R.	delete the tuple
2	a tuple in some pgR relation does not have a required child image in R	a) modify the tuple by replacing its image domain value by a blank (only part of tuple's key) b) delete the tuple
3	none	
MODIFY(R, k, t)		
1	a) t does not have a parent image in some pgR relation R'.	modify t's old parent image so that it becomes t's (new) parent image.

Table 6. (Continued)

Invariant	possible violation after insertion, deletion, and modification.	method for correcting violation.
	b) a tuple in some cgR relation R' has no parent image in R.	if image domain for R' is modified then delete the tuple, otherwise modify the tuple so that t becomes its parent image.
2	a) t does not have a required child image in some cgR relation R'.	if image domain for R' is modified then insert a child image in R' (use blanks where values are unknown), otherwise modify t's old child image in R' so that it becomes t's (new) child image.
	b) a tuple in some pgR relation does not have a required child image in R.	modify the tuple so that t becomes its child image.
3	none.	

t refers to a tuple of R, and k refers to the key of t.

2.5 (3,3) Normal Form

The normalization of relations to avoid certain anomalies in update operations is important in the logical construction of a database. Suppose we have a relation as shown in Figure 9.

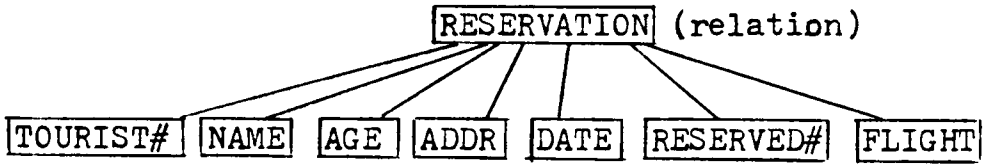

Figure 9. An unnormalized relation RESERVATION

There is some undesirable redundant information stored in this relation: TOURIST#, NAME, AGE, and ADDR are repeatedly stored for each reservation the same tourist made. So-called anomalies arise in connection with basic operations: insert, delete, and modify. Insertion anomalies could happen when we wish to record a tourist's address while he does not currently have a reservation, using this RESERVATION system as an example; because we can not put null values for key elements (TOURIST#, DATE, FLIGHT). Deletion anomalies could happen as the inverse of the above example in that should we delete the last reservation, we also delete the address and other information that we wish to retain about the tourist should that reservation be the only one left for that tourist. Modification anomalies happen because of redundancy in the

database; a tourist's ADDR is changed in one reservation tuple but not changed in other tuples. The above problems could be solved if we normalize the relation as shown in Figure 10

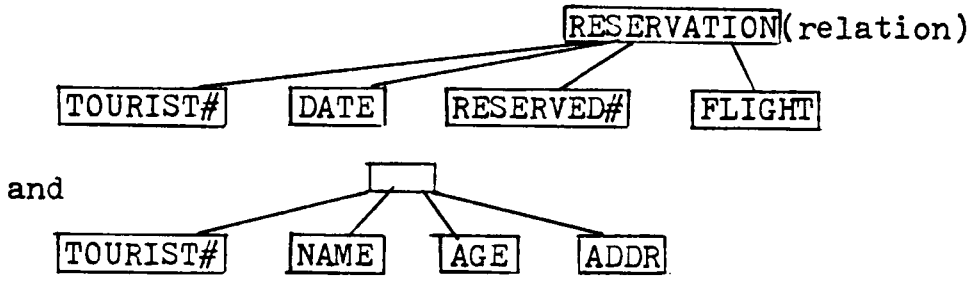

Figure 10. An normalization relation RESERVATION

Normalization theory is well developed in dealing with anomaly problems in aggregate abstraction. Certain normal forms are considered important and useful [such as first normal form, third normal form, and Boyce-Codd normal form.]

A relation is said to be in a certain normal form if it satisfies certain constraints. If one attribute A uniquely determines another attribute B in a relation, we say that there is a "functional dependency" (or FD) of B on A or $A \rightarrow B$. To be in 1NF (first normal form), the relation should satisfy the constraints that it contains atomic values only. The values in each column can not be further decomposed. To be in 2NF (second normal form), it should be in 1NF and every non-key attributes should be fully dependent on the primary key. A full functional dependency is such that

$A \rightarrow B$, (A functionally determines B) but B is not functionally dependent on any proper subset of A. The process of getting 2NF from 1NF is also known as removal of partial dependencies. To be in 3NF (third normal form), the relation should be in 2NF and transitive dependencies among non-key attributes should be removed. If A is the key and $A \rightarrow B$ and $B \rightarrow C$, and B is not a subset of A then there is a transitive dependency and the relation should be split into two. Normalization is the process for converting a relation in 1NF to be in 3NF or BCNF. Ullman [8] shows that every relation in 1NF can be converted to be in 3NF (which has certain nice properties, losslessness and preservation of dependency, not discussed here). "3NF", then, is a normal form that forces out most of update anomalies and normalization is the processes of removing embedded aggregates from an object.

In addition to maintaining the relational invariants, to insure that the aforementioned anomalies do not happen, the relations should be put in (3,3)NF [5]. The first subscript in (3,3)NF refers to 3NF and the process of deriving 3NF is the same as Smith and Smith's removing embedded aggregates and declaring them as independent aggregates. To derive (3,3)NF, not only certain attributes are factored out as additional components but also categories of the object are factored out. (second subscript of (3,3)NF) Factoring out a component is the removal of partial

dependencies and transitive dependencies in aggregation. Factoring out categories is the removal of functional dependencies from subsets of an object in generalization. In this way, some details of the object are moved to lower levels of abstractions. Normalization algorithms are well known for 3NF. Apparently, no known algorithms exist for normalization in the generalization plane. This paper does not introduce such algorithms but stops short at presenting examples of deompositions into (3,3)NF.

Examples of normalization.

A. Factoring out components.

Suppose the definition of FACILITY is

```
RELATION FACILITY (*FLY#:          CHAR 6,
                   *CITY:          CHAR 10,
                   AIRLN:          CHAR 5,
                   DEPART_TIME:    CHAR 7,
                   TERMINAL:       CHAR 3);
```

and there is a FD TERMINAL → AIRLN. After factoring out a component, the new definition will appear as:

```
RELATION FACILITY (*FLY#:          CHAR 6,
                   *CITY:          CHAR 10,
                    DEPART_TIME:   CHAR 7,
                    TERMINAL_ASSIGNMENT.T#:   CHAR 6);
```

```
RELATION TERMINAL_ASSIGNMENT (*T#:      CHAR 6,
                              AIRLIN:   CHAR 5);
```

Now the relation is in 3NF.

B. Factoring out categories.

Suppose we have the table as shown in Table 7.

Table 7. A relation ENROLLMENT in tabular form

ENROLLMENT

STUDENT#	CLASSIFICATION	RESIDENCY	ENROLL_HOUR	CHARGE
778-15-6950	GRADUATE	IN-STATE	3	90
698-75-1100	UNDER_GRAD	IN-STATE	15	180
123-01-2778	GRADUATE	OUT-STATE	9	420
770-15-2350	GRADUATE	IN-STATE	9	250
693-78-9327	UNDER-GRAD	OUT-STATE	15	450

There is no FD from ENROLL_HOUR to CHARGE. However, within the subsets of (GRADUATE, IN-STATE), (UNDER-GRAD, IN-STATE), (GRADUATE, OUT-STATE), and (UNDER-GRAD, OUT-STATE) there is a FD from ENROLL-HOUR to CHARGE. Therefore, we can factor out categories and get the following relations as shown in Figure 11, 12, 13, 14 and 15.

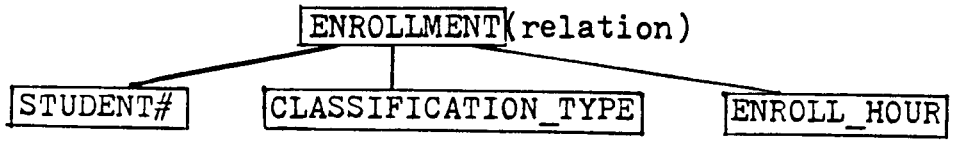

Figure 11. The relation ENROLLMENT

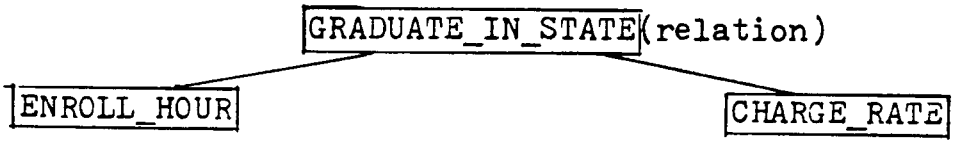

Figure 12. The relation GRADUATE_IN_STATE

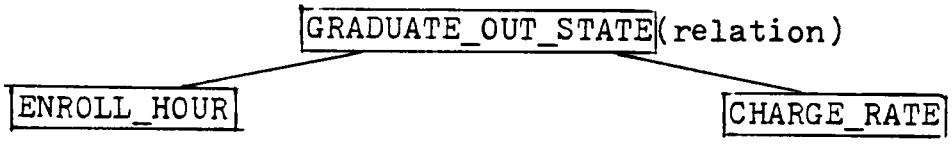

Figure 13. The relation GRADUATE_OUT_STATE

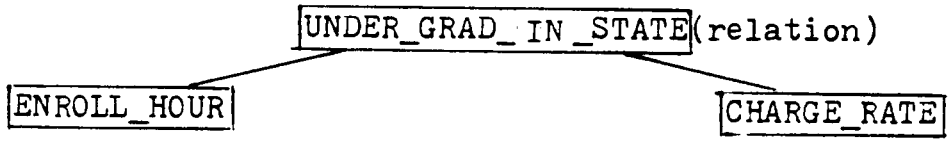

Figure 14. The relation UNDER_GRAD_OUT_STATE

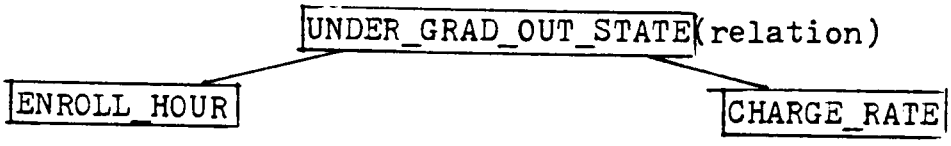

Figure 15. The relation UNDER_GRAD_OUT_STATE

The relations in their syntactical form are as follows:

```
RELATION ENROLLMENT (*STUDENT#:              CHAR 11,
                     CLASSIFICATION_TYPE:    CHAR 20,
                     ENROLL_HOUR:            NUM 2);
                    (GRADUATE_IN_STATE, GRADUATE_OUT_
                     STATE, UNDER_GRAD_IN_STATE, UNDER_
                     GRAD_OUT_STATE);

RELATION GRADUATE_IN_STATE (*ENROLL_HOUR:    NUM 2,
                            CHARGE_RATE:     NUM 3);

RELATION GRADUATE_OUT_STATE (*ENROLL_HOUR:   NUM 2,
                             CHARGE_RATE:    NUM 3);

RELATION UNDER_GRAD_IN_STATE (*ENROLL_HOUR:  NUM 2,
                              CHARGE_RATE:   NUM 3);

RELATION UNDER_GRAD_OUT_STATE (*ENROLL_HOUR: NUM 2,
                               CHARGE_RATE:  NUM 3);
```

CHAPTER 3

IMPLEMENTATION

This chapter is to have further discussions on relational invariants. Smith and Smith's tables for maintaining relational invariants will be examined and methods of maintaining the relational invariants and justified additions will be discussed. Finally, the actual implementation is to be briefly discussed and the syntax in implementation is presented.

3.1 Relational Invariants and Integrity Constraints.

In this chapter, Smith and Smith's "relational invariants" are first discussed. The purpose is to point out that the so called "invariants" are a type of integrity constraints and furthermore in an implementation environment, alternative semantics may be involved. Therefore, relational invariant will be treated as integrity constraints in this chapter.

A closely related concept to Smith and Smith's notion of relational invariants is the notion of integrity constraints. An integrity constraint is a constraint expressing some restriction on legal states of the database. Specification of domains, relations, and functional dependencies all are kinds of integrity constraints. The difference between integrity constraints and Smith and Smith's invariants is

that relational invariants reflect the well-formedness rules of a relation or a collection of relations and often leave out the particular semantics associated with the particular relation (that is a relation where EMPLOYEE → AGE and where AGE < 70). That is Smith and Smith's invariants are integrity constraints that are universal, applying to all objects regardless of their particular semantics. However, the methods of maintaining invariants is to perform update operations on one or more paR, caR, pgR, or cgR relations and these actions may cause further violations and need more update operations. From this we can clearly see that constraints are violated until all the update operations are done successfully. Therefore, we can look at the whole update operations as a transaction, where a transaction is a single execution of an operation which may comprise several atomic operations. A user INSERT "I" is treated as a "transaction" consisting of a (possibly large) number of system atomic inserts "i". If the transaction is unsuccessful, all the completed update operations in the transaction need to be undone to restore to the original state. In this sense, it is also similar to deferred constraints, where a deferred constraint is a contraint which may be enforced after a certain point but not continuously. Furthermore, from a case by case examination of the invariant tables. it is evident that semantics are also involved in

the interpretation of the invariants. To sum it up, so-called invariants actually are types of integrity constraints, and therefore they will be treated as maintaining integrity constraints in this paper.

3.2 Additions and Justifications to Smith and Smith's Relational Invariants.

Each invariant will be examined in this section using the reservation system example as used throughout this paper. The purpose is to interpret these rules and to point out certain undesirable features and to generate a broader class of interpretations. Case by case examinations are needed instead of taking the whole set of relational invariants as the fixed rules as we go through the examples in this section; that is semantic consideration is necessary. The point is that the five relational invariants (2 in aggregation and 3 in generalization) are not rejected as universal constraints for maintaining well-formed aggregation and generalization hierarchies. But it is pointed out that the methods needed to maintain the invariants may be considerably more complex than indicated by Smith and Smith; in fact, they may require idiosyncratic specification. The right hand column of the Smith and Smith's tables in Section 2.4.3 and 2.4.7 are being questioned. It is not the only way to maintain the constraints, as we show by example below. In the section below, potential alternative semantic side effects for the operators

of insert, delete, and modify in both the aggregate and the generalization planes are considered. As before (1), (2), and (3) refer to the invariants as the number indicated in the two tables of maintaining invariants and r, t, and k refer to the relation, tuple, and key respectively.

Aggregate.

Insert (R, t)

(1) There may be no side effect to the aggregate parent or child. Relational invariants are not violated before and after insertion. This is a relational invariant inherited from the relational model.

(2) There may be different side effects depending on two possible interpretations: the open and the closed world interpretations. In the closed world interpretation, certain relations are fixed and update operations which are side effects of other operations can not affect the content of these relations. For instance, in relation FACILITY, all the facilities are recorded in this relation, and it is a fixed set of facilities. In the open world interpretation, there is no such restriction. It assumes that if a child tuple is not present, a side effect should correct this situation (Smith and Smith do this). The method of correcting this is to insert a corresponding child tuple with key

value and nulls elsewhere. Taking relation RESERVATION for example, if a tuple RESERVATION is inserted and there is no corresponding child in FACILITY, in this case, a tuple <F_ID(key), null>, where nulls are for unknown attribute F_TYPE_NAME, needs to be added. (A semantics for null values is beyond the scope of this paper and will not be dealt with further.) To insert a tuple in the closed world interpretation, if no corresponding child exists, the insertion should be rejected. The assumption here is that all the facilities should have been already recorded since it is a closed world, (using the reservation system example) and therefore the insertion must be an illegal one.

Adding a tuple should not have any side effects to its parents in aggregation. For example, adding a tuple to FACILITY or TOURIST should not affect the parent RESERVATION (although in the case of TOURIST, it can be argued that you never add tourists unless they have reservations).

Delete (R, k)

(2) For the deletion of a parent in aggregation, when a RESERVATION is deleted, the information about a tourist may need to be kept or may not, and therefore an automatic decision should not be adopted

here. If a tuple in RESERVATION is to be deleted, it depends on the semantics that is adopted to decide to delete or keep a tuple in TOURIST. In general, if the parent relation represents an object, deleting it may often require deleting its aggregate parts. The following relation as shown in Figure 16 is a good example.

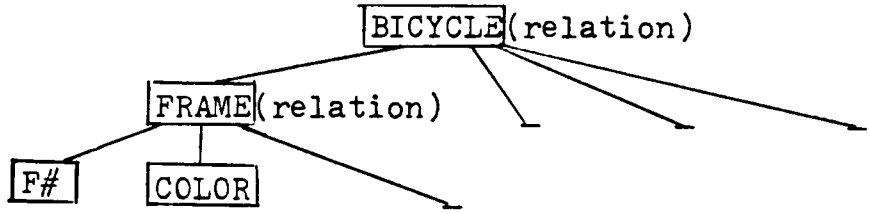

Figure 16. The relation BICYCLE

But if it represents a relationship, we may or may not wish to keep the underlying objects such as in RESERVATION example (we deleted the tourist but kept the facility). Deleting of a parent tuple is not covered by Smith and Smith's tables.

If a tuple in TOURIST is deleted, the parent RESERVATION needs to be deleted because TOURIST is part of the key in RESERVATION. If a tuple in TOURIST_TYPE is deleted, which is not a key of TOURIST, the tuples in TOURIST and RESERVATION still need to be kept; the attribute TOURIST_TYPE

is replaced in parent relation TOURIST with a null value. This is the same semantics as adopted by Smith and Smith for this case.

Modify (R, k, t)

(1) and (2) For (2), a key modification should be treated as delete followed by insert; the DELETE and INSERT operations should be used.
of key-raising property, the modification of non-key attributes should not affect parent relations. For the same reason, modifying elements in a tuple should be treated as a delete and followed by an input if the elements is part of its parent tuple's key. Smith and Smith treat key values not differently from other attributes.

We allow some nulls but a null in a foreign key domain is not propagated to the corresponding relation in which it is in the key domain. Smith and Smith do not consider this case.

Generalization.

Insert (R, t)

(1) If a tuple in AIRLINE is inserted and there is no parent image in FACILITY, then the tuple in FACILITY with the same key needs to be either modified or a tuple in FACILITY with the key F_ID and null in FACILITY_TYPE needs to be added. But in fact, a null in FACILITY_TYPE is not needed

because AIRLINE can be simply put as its value in the example.

(2) If a tuple in FACILITY is inserted and there is no child image in AIRLINE (or HOTEL or RESTAURANT), then the key F_ID will be inserted and nulls are inserted elsewhere. Somehow if the inserted tuple does not exist in F_TYPE_NAME, then this insertion either needs to be rejected or automatically make this insertion or leave the user the choice of alternative. Here, the open world interpretation and manual method is adopted leaving the user the choice of making sure that the insertion is correct. By "manual method", it means that the operation is not done automatically; it is by the choice of the user.

Delete (R, k)

(1) If a tuple in FACILITY is deleted and leaves a tuple in AIRLINE (or HOTEL or RESTAURANT) without parent image in FACILITY, then there is no purpose of keeping that tuple in AIRLINE.

(2) If a tuple in AIRLINE (or HOTEL or RESTAURANT) is deleted and leaves a tuple in FACILITY without child image, then for the same token there is no purpose in keeping that tuple in pgR. It seems that 2(a) of Smith and Smith's table is unlikely to happen because the relation FACILITY

and its children are linked via key. If a tuple in AIRLINE is deleted and leaves a tuple in FACILITY without child image, in this case, unless the raised key from AIRLINE is not part of the key of FACILITY, the image domain value can not be replaced by a null value.

Modify (R, k, t)

Again as in aggregation, modification of key values should be handled by the system as delete followed by insert.

For modify in generalization, the possible violation condition of 1(a) of relational invariants tables is the same as 2(b) and 1(b) is the same as 2(a); one is from the parent's point of view and the other is from the child's point of view. This is apparent from the underlying invariant properties.

For the possible violation condition of 1(a) or 2(b), if the tuple in AIRLINE (or HOTEL or RESTAURANT) is modified, the parent tuple in FACILITY needs to be modified, but again because of key raising this needs to be treated as delete and then followed by insert.

For the possible violation condition of 1(b) or 2(a), if a tuple in FACILITY is modified and F_TYPE_NAME gets changed, which is the image

domain of AIRLINE (or HOTEL or RESTAURANT), then the tuple in AIRLINE (or HOTEL or RESTAURANT) needs to be deleted and then a new tuple with the modified value is added into a different child relation. For example, if a tuple <hotel,12345> is changed to <restaurant,12345>, that hotel must be deleted in relation HOTEL and then will be added to relation RESTAURANT. Other changes in FACILITY should not affect AIRLINE or HOTEL or RESTAURANT.

The set of ways of maintaining constraints proposed by Smith and Smith has the property of simplicity, but this set can not cover many semantic situations as discussed above. The additions to the invariants here complicated the whole process but are essential for capturing the semantic information which is the purpose of adding abstractions to relational databases. It should also be pointed out that rather arbitrary transaction can be invoked which use system insert, delete, or modify arbitrarily and which leave the DB in a state satisfying the invariant integrity constraints. The user insert, delete, or modify commands are actually only commonly used instances of such transactions. For example, if for some reason, the DB is not in a state of integrity and to correct the problem it is only necessary to insert or modify or delete one tuple in a certain relation, then it may be desirable to invoke an atomic

INSERT, DELETE, or MODIFY which has no propagation side effects.

3.3 Method of Maintaining Relational Invariants.

Methods of maintaining relational invariants can be implemented in several ways as follows:

a. The manual method. Each time when there is an update operation which in turn triggers another update operation, the automatically triggered operations are not performed; instead, at that point the system will let the user decide what actions to take by querying the user. This method gives the user the ultimate responsibility, but it may create inconsistencies in the database by allowing this flexbility.

b. Triggered functions. A triggered function is a set of code with one or more program statements. It is tied with a relation and an operation. When a relation is operated on (such as an execution of INSERT on FACILITY), then this piece of code is executed to cause further update operations (side effects). Triggered functions can be built into the system; each trigger can be associated with an individual relation and operation. For example, on relation AIRLINE, we can have a trigger function.

```
TRIGGER AIRLINE ON INSERT
INSERT FACILITY facility_type.f_type_name = AIRLINE
               f_id = airline.f_id;
END TRIGGER.
```

In this example, if that particular tuple of AIRLINE does not exist, this new AIRLINE tuple will be inserted; otherwise, this trigger creates a duplicate tuple which will not be inserted by the system. Another important point is that if triggers are tied with sets of events, then such an approach tends to lessen the degree of cluttering the system with triggered operations. If the triggers are tied too closely with individual relations, then this approach tends to clutter the system with triggers associated with relations. This is a drawback in triggered function approach. Another disadvantage this approach shares with the triggered approach is that the user can access these triggers and in turn an inconsistent database may be created. On the other hand, if the user decides that there are exceptions to the invariants that the system does not support, he can easily change the trigger functions. This certainly offers flexbility. An alternative to the user accessable triggers is to have system triggers that users can not alter. The system trigger approach can be fully automatic

triggers or partially automatic triggers and partial manually controlled triggers.

c. All the triggered update operations are specified in the relational descriptors. This approach is more consistent with the overall database design without cluttering the database with all the trigfered functions. On top of this, the flexibility can be offered by specifying whether certain operations are automatic or manual. This approach is adopted by this paper as will be described below.

3.4 Syntax of Maintaining Relational Invariants.

As mentioned above, approach (c) is the method chosen for actual implementation. The syntax after adding triggered operations to it will look like the example below. For a BNF notation, see Section 3.5.

```
RELATION RESERVATION (*TOURIST.TOURIST#:    CHAR 11,
                      *DATE:                CHAR  8,
                       RESERVED#:           CHAR  2,
                      *FACILITY.F_ID:       CHAR 15)
                    ( )
                      (INSERT    TOURIST    AUTOMATIC,
                       DELETE    TOURIST    MANUAL,
                       MODIFY    FACILITY   REJECT,
                       MODIFY    TOURIST    REJECT);
```

Here, AUTOMATIC means that Smith and Smith like update will be carried out and the propagation will be continued. MANUAL means that at this point, the user can decide if future propagaration is needed interacting with the system. REJECT means that this update operation should be rejected. INSERT, DELETE, and MODIFY facilitate the further propagation. Without specifying the above update operations means that no need of further propagation on the relation that follows the update operator. REJECT functions a little different from AUTOMATIC and MANUAL. When encountering a REJECT, which only used along with MODIFY, the system checks the attributes of the previous relation, such as RESERVATION in the above example, against the propagated relation name, such as FACILITY, and if they match, the update should be rejected.

As RESERVATION is not a generalization of other entities, the parentheses "()" contain null values in it; otherwise, they should contain the entities, which are generalized by RESERVATION, in them. As entities in both aggregation and generalization planes are treated equally, if propagations involve generalization entities, they will be treated just as the aggregation entities in the above example. In this example, if an insertion is made on RESERVATION, then the system checks the INSERT in the propagated operations. The system finds that TOURIST

should be inserted and then further propagations are carried on. In this case a tuple in TOURIST will be inserted with a key value and nulls elsewhere. When DELETEing a RESERVATION, the information about a tourist may still need to be kept and consequently the manual operation is needed here. A requirement is necessary in that the system needs to check to see if this RESERVATION is the only reservation and the last one in this reservation. For MODIFY on RESERVATION, if FACILITY or TOURIST is the target, then this MODIFY should be rejected.

In actual implementation, one crucial point is that if the propagation of triggered operations generates a cycle (loop), the operations should be stopped. This can and should be built into the system with a method such as storing each involved relation and check against each of these relations each time when there is a further triggered operation. The problem of looping is further discussed in section 3.6.

3.5 SUR, SURLY, and a Revised SURLY Syntax.

To demonstrate the semantics of database abstractions, an actual implementation of the system was written in SITBOL. The relational database system SUR (Single User Relational database management system) was modified in its suntax to reflect the structure of abstractions. Some features of SUR which are not relevant to the purpose of

this paper are not implemented. More specifically, only a subset of SUR is implemented; exclusions are query oriented operations and secondary indices. While needed, these features can be modularly added to the implementation, but the rationale is that DELETE, INSERT, and MODIFY are the only operations affected by abstractions.

The following quotes from [7] best describe SUR. "SUR a Single User Relational database management system, is designed to offer relational database facilities to a host language--PL/I, SNOBOL, LISP, PASCAL,...on a large, mini, or micro computer. SURLY, the SUR language facility, consists of a data definition language (DDL) and a relationally complete data manipulation language (DML)."

"SURLY is a relational algebra language with operators such as JOIN, PROJECT, SELECT, UNION, SET_DIFFERENCE as its query oriented operators. It is designed to be extensible so that after a core of basic commands and utility routines have been implemented, new commands can be added to SURLY in a modular way. Extensions can include routines such as VIEW, TRIGGER, INTEGRITY, RETRIEVE (relational calculus operator), hierarchical formatted print, LINK, DAY FILE, UNDO, DUMP, et cetera. The major restriction is that relations should never be too large to fit in a program's address space."

Because of single user environment, security constraints are not supported.

The syntax of the subset of SURLY is listed in Table 8, with "**" in the left hand column to indicate modifications added for the purposes of this paper.

3.6 Some Considerations in Actual Implementation.

When the program interprets relational descriptors it sets up links between aggregate children and parents, and between generalization children and parents for side effect propagation purposes. One single INSERT, DELETE, or MODIFY command triggers a series of such commands without looping infinitely (see below). A primary key is also specified and built into a relational descriptor. For accessing tuples via key, a hash index is built and maintained. Some important considerations are discussed below.

(1) There is an individual hash table for each relation. Because of the key-raising property, two tuples in two different relations may have the same key and so a key is only unique in a certain relation. Consequently, a hash table is needed for each relation.

(2) Nulls and blanks. The positional order of the elements in a tuple is important in the implementation. Trailing nulls need not be represented explicitly; other nulls must be replaced by a

METASYMBOLS

::= means "is defined as"

{x} repeat x zero or more times

[x] x is optional

x|y either x or y

x||y concatenate x to y

<x> x{[,]x} e.g. x x x... or x, x, x...

```
  surly_input     ::= {command}
  command         ::= RELATION relationname
                              (<[*]attribname format>)
                              ([<relationname>])
**                            ([<property>]);|
                      DESTROY <relationname>;|
                      INSERT relationname tuple;|
                      DELETE relationname [WHERE (qualifier)];|
**                    MODIFY relationname [WHERE (qualifier)]
                             TO <attrib = value>;|
                      INPUT {relationname {tuple;}} END_INPUT;|
                      PRINT relationname;
**format          ::= :|| CHAR length|NUM length
  length          ::= an integer
  relationname    ::= identifier
**attribname      ::= attrib|attrib||<.attrib>
```

Figure 8. The syntax of the subset of SURLY

```
attrib          ::= identifier
identifier      ::= letter || {letter| rumber|_}
value           ::= character string varying
qualifier       ::= compare|
                    compare AND qualifier|
                    compare OR qualifier
compare         ::= attribute relop value
relop           ::= = | < | <= | >= | > |¬=
comment         ::= /* comment */
**property      ::= INSERT relationname method|
                    DELETE relationname method|
                    MODIFY relationname method
**method        ::= AUTOMATIC|MANUAL|REJECT
```

Figure 8. (Continued)

distinguished symbol such as "?" which has the effect of specifying the presence of null or unknown values.

(3) Backup and recovery. Two methods could be used. The first method is to process one INSERT, DELETE, or MODIFY command and the triggered commands follow the original command one after another. If the whole process can not be completed, that is one of them should be rejected before completion, then the system needs to trace back to undo all the completed commands. Some method of backup and recovery is required in this first approach. The second method is to look at all the triggered operations as a transaction. If it is a committed transaction (the transaction can be completed successfully) then the system processes all the commands; otherwise an uncommitted transaction should not be allowed to enter changes into the database. In doing this, a preprocessor is needed to concatenate all the propagated commands to form a new command string. This paper adopts the second method. Whenever there is an INSERT, DELETE, or MODIFY, the preprocessor checks through the relation's definition and determines if this operation can be performed. If it can be

performed, the preprocessor goes further on propagation and concatenates this operation to the previous one until no more propagation is needed. If there is a REJECT for a certain operation, then this transaction is an uncommitted one. In such a way, a transaction is a committed one or an uncommitted one can be determined.

(4) Avoiding infinite cycle. The easiest way of terminating looping is to set a pointer for each propagated relation initially to null. When first propagating through a certain relation, we set the relation's pointer to one. Each time before propagating through a certain relation, the system checks the pointer's value of this relation to decide if further propagation is needed. If the pointer's value of this relation is null, then this relation has not been propagated through yet, and propagation can go through this relation and pointer of this relation is set to 1. In this way, when encountering a 1 value of a pointer of certain relation, further propagation is meaningless and the propagation should be terminated.

(5) Manual method in propagation. A special feature supported by the system is the manual method in propagation. This method allows the user to

decide if certain INSERT, DELETE, or MODIFY should occur in contrast to the automatic method. To the system, certain operations could either occur or not occur; to the user, he interacts with the system to decide if the operation should occur or not depending on the circumstances that he faces. This is a very useful feature that allows much flexilility to the user.

CHAPTER 4

CONCLUSION

Smith and Smith introduced a high level (hierarchical) organization to a set of relations so the user can organize his view conceptually. More specifically, they introduced the following ideas:

(1) Aggregation which is very much like 3NF (dealing with removing embedded aggregates),

(2) generalization which seems to have no correlative structure in normalization,

(3) a general notation for orthogonally represented aggregation and generalization hierarchy,

(4) a PASCAL-like notation for describing abstracted objects, and

(5) semantics for maintaining a set of invariant relationships which keep relations updated in a uniform way.

This thesis adds to the work of Smith and Smith in the following ways:

(1) Several alternative syntaxes were considered and their properties examined, a PL/I-like syntax was chosen for compatability with SURLY,

(2) the user is provided with alternative sets of rules for maintaining invariant conditions that support abstractions in the database, and

(3) an implementation is discussed, which allows the user to build aggregation and generalization hierarchies while preserving the property that relations are still tables, and which allows for propagated update operations to be performed automatically or manually in an interactive environment.

LIST OF REFERENCES

LIST OF REFERENCES

1. Chen, P. P. S. "The Entity-relationship model--toward a unified view of data." ACM Transactions on Database Systems, March 1976.

2. Date, C. J. An Introduction to Database Systems. Third Edition. Reading, MA: Addison-Wesley, 1981.

3. Smith, John Miles and Smith, Diane C. P. "Database Abstractions: Aggregation." Communications of ACM, June 1977, pp. 405-13.

4. Smith, John Miles and Smith, Diane C. P. "Database Abstractions: Aggregation and Generalization." ACM Transactions on Database Systems, June 1977, pp. 105-33.

5. Smith, John Miles. "A Normal Form for Abstraction Syntax," in Conference on Very Large Data Bases, 1978, pp. 156-62.

6. Smith, John Miles and Smith, Diane C. P. "Integrated Specification for Abstract Systems." Salt Lake City, 197-

7. Thompson, Craig W. "SUR: a Single User Relational DBMS." Knoxville, 1980.

8. Ullman, Jeffrey D. Principles of Database Systems. Potomac, MD: Computer Science Press, c.1980.

9. Byte, Byte Publications, N.H.: Peterborough, v.1, 1976-

VITA

Benjamin Y. Lin was born in China on March 21, 1947. He grew up in Taiwan. He received his elementary and high school education in Tainan, Taiwan. In September 1967, he entered National Chung Hsing University, and in June 1971 he received a Bachelor of Law degree. He was in military service from 1971 to 1973 as a second lieutenant in Taiwan.

In May 1975, he received a Master of Library Science degree from George Peabody College for Teachers, Nashville, Tennessee.

After working as a catalog librarian from 1975 to 1978 at Louisiana College, Pineville, Louisiana, he entered the Graduate School of The University of Tennessee, Knoxville, in September 1978. He was a graduate teaching assistant from 1980 to 1981. He received the Master of Science degree with a major in Computer Science in June 1982.

The author works for Honeywell--Large Information Systems Division since August 1981. His current research interest is in logical database design.