ENHANCE DBMS CAPABILITIES USING SEMANTIC DATA MODELLING APPROACH

A thesis presented to
The Department of Computer Science
of
The Chinese University of Hong Kong
in partial fulfillment of the requirements
for the Degree of Master of Philosophy

By
Yip Wai Man

May 1990

316046

thesis
QA
76.9
D35Y56

TABLE OF CONTENTS

ABSTRACT

ACKNOWLEDGEMENTS

PART I

1 OVERVIEW ON SEMANTIC DATA MODELLING APPROACH ... 1

2 SCOPE OF RESEARCH 4

3 CONCEPTUAL STRUCTURE OF SAM* 7
- 3.1 Concepts and Associations 7
  - 3.1.1 Membership Association 8
  - 3.1.2 Aggregation Association 8
  - 3.1.3 Generalization Association 9
  - 3.1.4 Interaction Association 10
  - 3.1.5 Composition Association 11
  - 3.1.6 Cross-Product Association 12
  - 3.1.7 Summary Association 13
- 3.2 An Example 14
- 3.3 Occurrences 15

PART II

4 SYSTEM OVERVIEW 17
- 4.1 System Objectives 17
  - 4.1.1 Data Level 17
  - 4.1.2 Meta-Data Level 18
- 4.2 System Characteristics 19
- 4.3 Design Considerations 20

5 IMPLEMENTATION CONSIDERATIONS 23
- 5.1 Introduction 23
- 5.2 Data Definition Language for Schema 24
- 5.3 Construction of Directed Acyclic Graph ... 27
- 5.4 Query Manipulation Language 28
  - 5.4.1 Semantic Manipulation Language 29
    - 5.4.1.1 Locate Concepts 30
    - 5.4.1.2 Retrieve Information About Concepts 30
    - 5.4.1.3 Find a Path Between Two Concepts .. 31
  - 5.4.2 Occurrence Manipulation Language 32
- 5.5 Examples 35

6 RESULTS AND DISCUSSIONS 41
- 6.1 Allow Non-Homogeneity of Facts about Entities 41
- 6.2 Field Name is Information 42
- 6.3 Description of Group of Information 43
- 6.4 Explicitly Description of Interaction 43
- 6.5 Information about Entities 44
- 6.6 Automatically Joining Tables 45

6.7 Automatically Union Tables 45
6.8 Automatically Select Tables 46
6.9 Ambiguity 47
6.10 Normalization 47
6.11 Update 50

PART III

7 SCHEMA VERIFICATION 55
7.1 Introduction 55
7.2 Need of Schema Verification 57
7.3 Integrity Constraint Handling Vs Schema Verification 58

8 AUTOMATIC THEOREM PROVING 60
8.1 Overview 60
8.2 A Discussion on Some Automatic Theorem Proving Methods 61
8.2.1 Resolution 61
8.2.2 Natural Deduction 63
8.2.3 Tableau Proof Methods 65
8.2.4 Connection Method 67
8.3 Comparison of Automatic Theorem Proving Methods 70
8.3.1 Proof Procedure 70
8.3.2 Overhead 70
8.3.3 Unification 71
8.3.4 Heuristics 72
8.3.5 Getting Lost 73
8.4 The Choice of Tool for Schema Verification 73

9 IMPROVEMENT OF CONNECTION METHOD 77
9.1 Motivation of Improving Connection Method 77
9.2 Redundancy Handled by the Original Algorithm 78
9.3 Design Philosophy of the Improved Version 82
9.4 Primary Connection Method Algorithm 83
9.5 AND/OR Connection Graph 89
9.6 Graph Traversal Procedure 91
9.7 Elimination Redundancy Using AND/OR Connection Graph 94
9.8 Further Improvement on Graph Traversal ... 96
9.9 Comparison with Original Connection Method Algorithm 97
9.10 Application of Connection Method to Schema Verification 98
9.10.1 Express Constraint in Well Formed Formula 98
9.10.2 Convert Formula into Negation Normal Form 101
9.10.3 Verification 101

PART IV

10 FURTHER DEVELOPMENT 103
10.1 Intelligent Front-End 103
10.2 On Connection Method 104
10.3 Many-Sorted Calculus 104

11 CONCLUSION 107

APPENDICES

A COMPARISON OF SEMANTIC DATA MODELS 110

B CONSTRUCTION OF OCCURRENCES 111

C SYNTAX OF DDL FOR THE SCHEMA 113

D SYNTAX OF SEMANTIC MANIPULATION LANGUAGE 116

E TESTING SCHEMA FOR FUND INVESTMENT DBMS 118

F TESTING SCHEMA FOR STOCK INVESTMENT DBMS 121

G CONNECTION METHOD 124

H COMPARISON BETWEEN RESOLUTION AND
CONNECTION METHOD 128

REFERENCES 132

Abstract

While conventional database management system (DBMS) emphasizes efficiency and reduction of redundancy, a new approach to data modelling, namely semantic data model, is emerged that aims at providing increased expressiveness to the modeler and incorporating a richer set of semantics into the database. In this thesis, we discuss the design and implementation of a prototype of a DBMS, using a semantic data model called SAM*. The work involves a mapping from a data model into a logical organization of an underlying DBMS, and the design and development of two query languages, Semantic Manipulation Language and Occurrence Manipulation Language, to access the database. The results show that, with our system, user can formulate a query in a simple way and understand meta-knowledge of the database through inquire about the schema. Also, we have tackled an issue called schema verification which has not been touched by database communities. The verification determines whether there is contradicting constraints being imposed on the schema. We take an automatic theorem proving approach, and discuss three common provers. We suggest the use of a connection method, and develop an AND/OR connection graph that help to eliminate redundancy without affecting the soundness and completeness of the connection method. Further development using many-sorted calculus is proposed, as our implementation has already constructed a data structure that is ready to be used by the calculus.

Acknowledgements

I would like to thank my supervisor, Dr. Askey C. Yau, for guiding me in carrying out this research, and supporting me in exploring various research areas. I would also like to thank Dr. Y. S. Moon for lending me readings that do stimulate my interest in semantic data modelling and automatic theorem proving.

PART I

CHAPTER 1

OVERVIEW
ON
SEMANTIC DATA MODELLING APPROACH

While conventional database management system (DBMS) emphasizes efficiency and reduction of redundancy, a new approach to data modelling, namely semantic data model, is emerged that aims at providing increased expressiveness to modelers and incorporating a richer set of semantics into the database [PeMa88]. Research in this field is along the direction of capturing the way human perceives objects, events and abstract concepts in the world, hoping to make the accessing process to the information system as "natural" as possible, and requiring the amount of knowledge about internal structures of the information system to be known by the user as little as possible. Since the innovative paper published by Smith and Smith [SmSm77], abstraction mechanism has become an important and necessary feature of semantic data model. Four important abstractions are identified: generalization (is-a relation), aggregation (relationship between low-level types is considered as a higher level type), classification (is-instance-of) and association (is-member-of). Most reported semantic data models have more or less include these abstractions.

Generally speaking, benefits of semantic data model include the following four aspects [PeMa88]:

a) economy of expression: user usually can extract the full range of information from a database with greater ease;
b) integrity maintenance: with mechanisms for defining integrity constraints for inter- or intra records, user can freely manipulate data on a level removed from the low-level record structures;
c) modelling flexibility: semantic data models, through the use of abstractions, permit a user to model and view data on different levels; and
d) modelling efficiency: the database designer, while constructing a particular database schema, does not has to worry about any implementation at low level.

Various semantic data models are reported [HuKi87, PeMa88] and each of them owns different features. A brief description of some popular data models is given in the following paragraphs.

TAXIS [MyBe80] is a language able to model a strongly hierarchical structured data model, having generalization, classification and aggregation abstractions. It can also handle multiple inheritance (an object inherit attributes from two or more abstract objects). A compiler for the language has been developed [NiCh87, NiCh89]. The language is characterized by being incorporated

with specification of semantic integrity constraints, and exception-handling mechanisms.

SDM [HaMc81] is a data model that put emphasis on the classification of *entities* into *classes*. Each entity or class owns *attributes*, and classes are logically related by *interclass connections*. Attributes can be derived from other values in the database. A commercial product bases on the model is developed by Unisys Corporation [JaGu88]. Its data manipulation language resembles SQL [Lans88], but with enhanced capability of recognizing semantic concepts.

DAPLEX [BaLe88, LyVi87, Ship81] is a data definition and manipulation language for a database using functional data modelling approach. A function maps attributes of entity into the values of the attributes. Relationships are implicitly embedded in the functions.

Some researchers focus on the formal description of data model [Abit87], and derive procedures for update propagation in hierarchical structured data model.

CHAPTER 2

SCOPE OF RESEARCH

Although various semantic data models have been proposed, seldom of them test the idea under a practical environment. This project aims at illustrating the enhanced capabilities of a DBMS using semantic data model under a d application environment. We choose investment analysis as the application domain for our database because the complexity of this problem can help to test the power of semantic data model. Requirements of a database for investment analysis are:

a) ability to store a wide variety of data: the database should able to model different types of information, like statistical data, without lowering the ease of accessing the information;

b) ability to cope with change of environment: the amount of information needed in an investment environment is ever changing and expanding, e.g. new projects announced by companies as well as political and economic crisis expand the set of information demanded by analysts; and

c) ability to abstract information: an investment analyst usually view information from different perspective, e.g. the analyst may classify

stocks by the sectors to which they belong, or by their volume of transactions, and then each group can be treated as a whole.

A particular semantic data model is chosen, namely Semantic Association Model (SAM*) [Su83, Su86], because of its

a) richness of pre-defined semantic relationships;
b) ability to specify constraints to check for data addition and deletion automatically;
c) support of the four basic abstractions;
d) allowance of structuring data in a network which is more general than hierarchical structure; and
e) allowance of complex data structures, like time-series for statistical information.

A comparison of other semantic data models with SAM* are given in Appendix A.

In his paper, Su defines SAM* at a conceptual level, and derives a G-relation that maps a SAM* data model into a relational data model. In this thesis, we discuss the design and implementation of constructing a prototype of a DBMS using SAM* as its data model.

The thesis is divided into four parts. In Part II, followed by a brief

description on SAM* in Chapter three, we present the development of a data definition language for a schema written in SAM*, the mapping of the schema into logical organization of an underlying database system, and the development of two query manipulation languages for accessing the database. In Part III, we describe how to use an automatic theorem prover to solve a problem called schema verification, and introduce an AND/OR connection graph to eliminate redundancy in the construction of a proof. In Part IV, we suggest area of further development, and make a conclusion of the thesis.

CHAPTER 3

CONCEPTUAL STRUCTURE OF SAM*

3.1 Concepts and Associations

SAM* [Su83] perceives the world as a collection of *concepts*. Concepts represent physical or abstract items of information. A concept can be

a) a fundamental information unit that is well understood and thus need not be defined; or

b) a more abstract one than that described in (a), and is defined in terms of other concepts.

A concept can own an *association* that itself groups a set of concepts. For example, teachers and students are concepts. Another concept called teach can has an association that groups a teacher and his students. Seven types of associations are pre-defined. In the following sections, we informally discuss each of them in the context of investment analysis. In graphical representation, concept is represented by a circle, with the name of the concept being attached aside. Association is represented by a single capital letter (the first letter of the association) within the circle corresponding to the concept it belongs to.

3.1.1 Membership Association

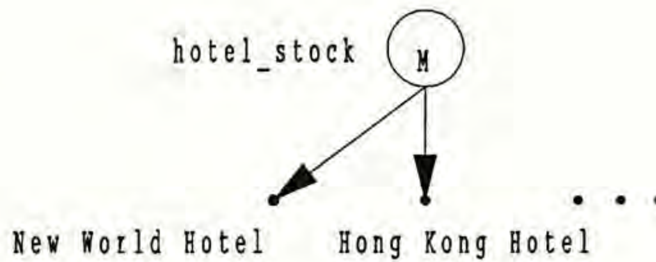

Figure 1 Example on Membership Association

Membership association groups similar concepts together. Optionally, membership constraint is attached to specify the necessary condition(s) to be satisfied by each member. In figure 1, the concept "hotel_stock" groups all stocks having business related to hotel service, where the description can act as the membership constraint. "New World Hotel" and "Hong Kong Hotel" are instances of the concept.

3.1.2 Aggregation Association

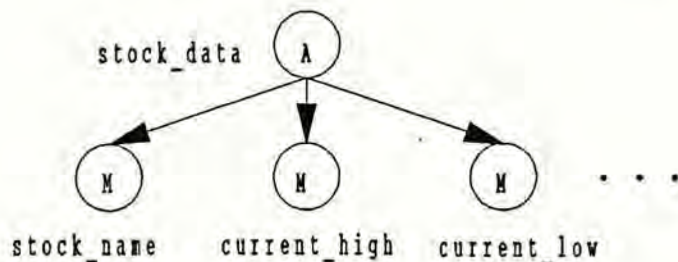

Figure 2 Example on Aggregation Association

A set of concepts can be grouped to describe or characterize another concept. In figure 2, the concept "stock_data" is characterized by a "stock_name", its "current_high", and its "current_low" in the market. One or more of the characterizing concepts should act as identifier(s) that can uniquely identify an instance of the characterized concept. Here, the concept "stock_name" can do the job. Each particular stock name identifies a set of information about that stock.

3.1.3 Generalization Association

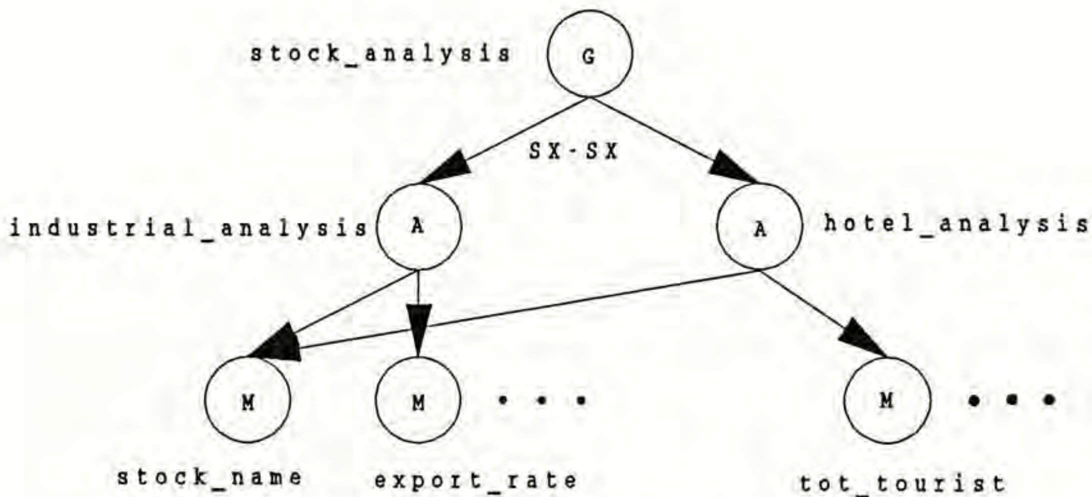

Figure 3 Example on Generalization Association

Concepts can be grouped by their generic nature to form a more general concept. As shown in figure 3, we can analyze stocks according to the sectors they belong to, like industrial and hotel sector. These two perspective on stocks analysis are generalized into a generic concept "stock_analysis". Four types of constraints can be specified to describe the relationship between any two sets of

instances of component concepts: set equal (SE), set intersection (SI), set exclusive (SX), and set subset (SS). Refer to the example again, the "SX-SX" means "industrial_analysis" and "hotel_analysis" represent two entirely different sets of stocks.

3.1.4 Interaction Association

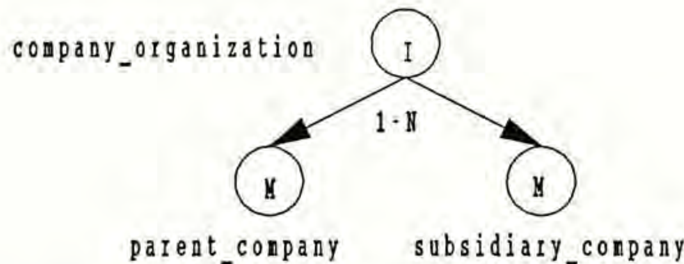

Figure 4 Example on Interaction Association

Interaction association describes facts or events among concepts. In figure 4, "company_organization" tells that a parent company can hold more than one company. Like ER model, there are three types of constraints to describe the mapping between component concepts under an interaction association: one-to-one (1-1), one-to-many (1-N), and many-to-many (N-M). With this association, we can specify relationship (as described in the I node) between two concepts.

3.1.5 Composition Association

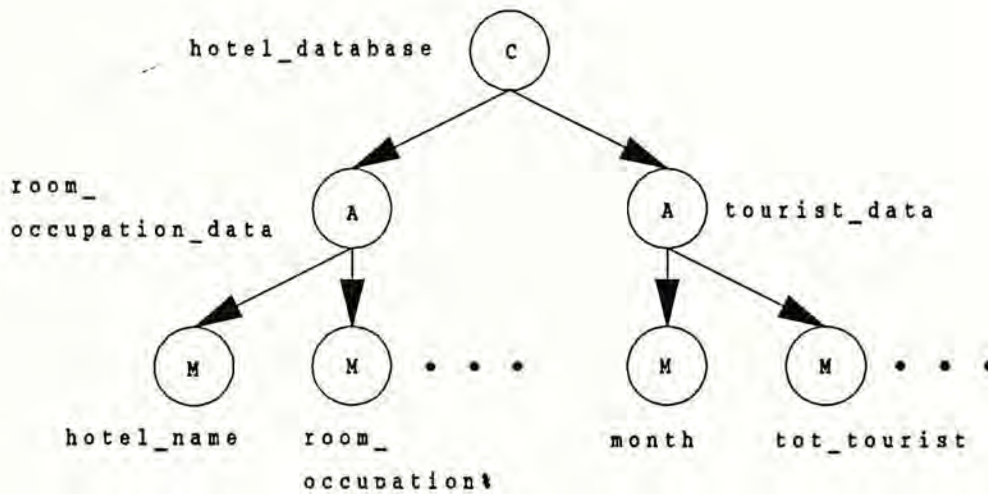

Figure 5 Example on Composition Association

Composition association collects concepts that are physical parts or functional components of another larger part or component. In figure 5, "hotel_database" composes of all information about room occupation percentage, and all information about number of tourists in Hong Kong. The components are not interacting, like those in interaction association, nor they are characterizing or characterized concepts, like those in aggregation association. Composition association has the semantic meaning of "is a part of" rather than "is a" as for generalization association.

Having a composition association to group and identify a set of entities, we can further describe this set using aggregation association. As shown in figure 6, "information_source" describes the source of all information in "hotel_database".

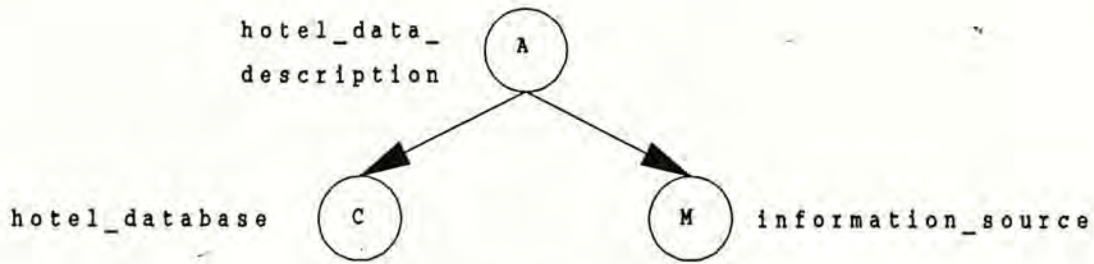

Figure 6 Example on Group Description

3.1.6 Cross-Product Association

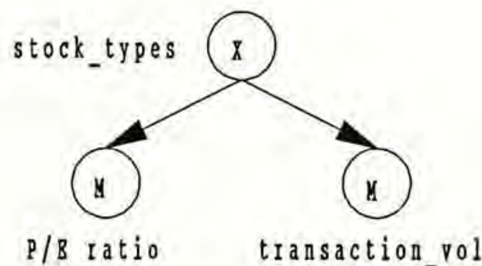

Figure 7 Example on Cross-Product Association

Cross product association takes a cross product on a group of concepts to form another concept. Unlike aggregation or interaction association, each cross product defines a category of entities rather than a particular entity. The data model in figure 7 can be used to define different sets of stocks depending on their P/E ratio and transaction volume. For example, active prospective stocks are those having high P/E ratio and high transaction volume; while inactive prospective stocks are those having high P/E ratio and low transaction volume. Through summary association (to be described next), a set of concepts can be used to describe each category so defined.

3.1.7 Summary Association

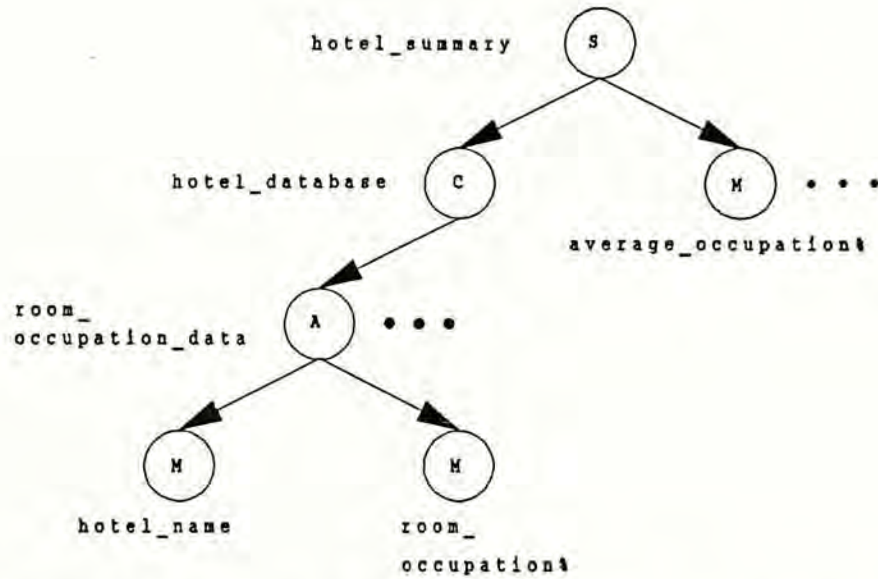

Figure 8 Example on Summary Association

Summary association collects a group of concepts to summarize or characterize a set of entities defined by a cross product or composition association. In figure 8, the concept "average_occupation%" calculates the average of "room_occupation%" of all instances of "hotel_name". Procedure can be attached to "average_occupation%", indicating how the summarizing data is derived.

3.2 An Example

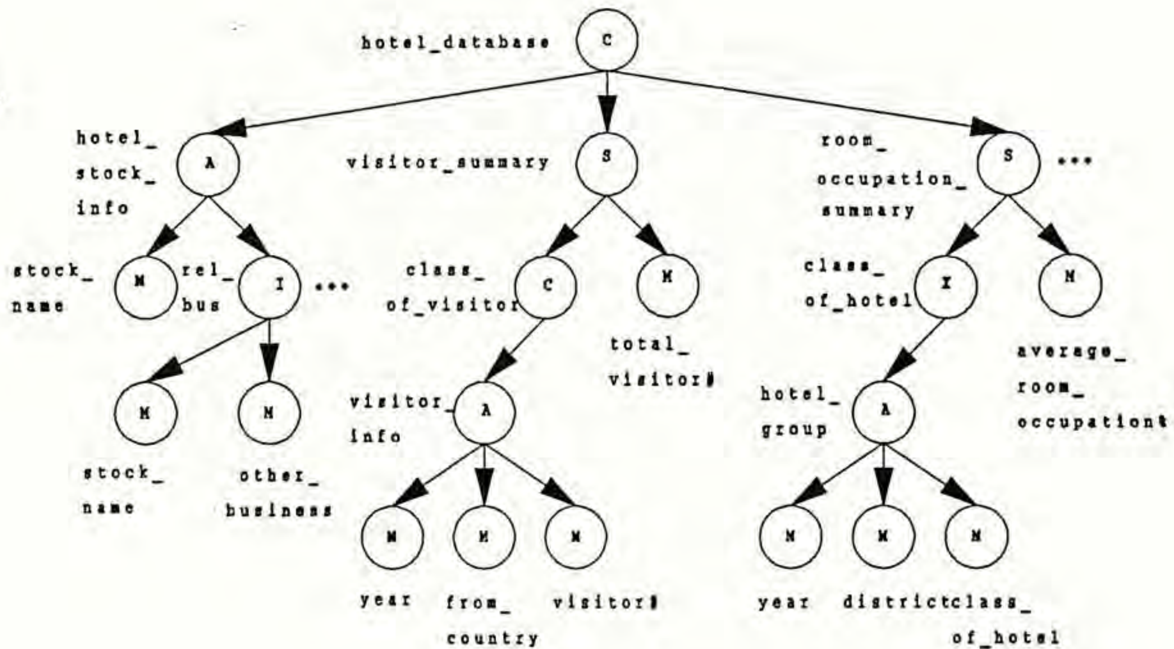

Figure 9 An Example on Using SAM*

Figure 9 illustrates a data model for analyzing stocks belonging to hotel sector. It models information of each hotel sector stock, as well as summarizing data about visitors in Hong Kong and room occupation of hotels. Each hotel sector stock is identified by its stock name, and may involve in other businesses. There is data about the number of visitors coming to Hong Kong in each year. The total of these numbers is stored under the concept "total_visitor#". Also, by cross product association, "class_of_hotel" defines several categories of hotels by the districts they locate and their classes, with each of the categories being characterized by a summarizing data "average_room_occupation%".

3.3 Occurrences

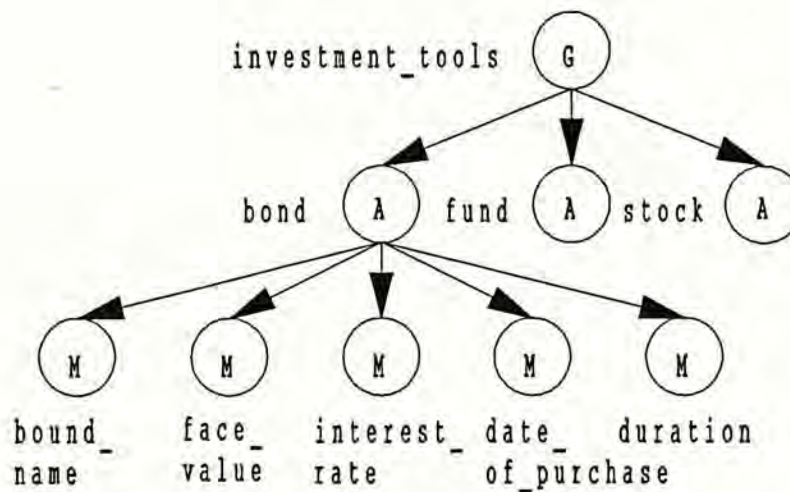

Figure 10 Example of a SAM* data model

Each concept owns *occurrences*, instances (as termed before) of the concept. Consider an example as shown in figure 10, "bond" is a kind of "investment_tools" that is characterized by its "bond_name", "face_value", "interest_rate", "date_of_purchase" and "duration". An occurrence of "bond" can be any particular bond like Fixed Interest Rate Substitute Transaction offered by MTR with face value of HK$ 500,00, interest rate at 8%, date of purchase in May 1992, and duration of 5 years.

Concepts of different association types have occurrences of different structures. For example, when the concept "investment_tools" generalizes "bond", "stock" and "fund", the three particular investment tools are all the same to the concept "investment_tools". Thus, the set of occurrences of "investment_tools" is

taken as a union of all these particular types of investment tools.

occurrences of investment tools =

{occurrences of bonds U occurrences of stocks U

occurrences of fund}

where U stands for the set operation union. Suppose "fund" generalizes "fixed_rate_fund" and "variable_rate_fund", then the above set can be further expanded by replacing occurrences of fund with occurrences of "fixed_rate_fund" and "variable_rate_fund". Generally, occurrences of a concept are constructed by applying union and/or cross-product operations on the occurrences of the less abstract concepts. Appendix B presents the construction of occurrences in a recursive manner.

For convenient purpose, the more abstract concept (e.g. investment tools, relative to bond) is called parent concept and the more specific concept (e.g. bond, relative to investment tools) is called child concept. In SAM*, a parent concept may has more than one child concept and vice versa, allowing a network of concepts to be constructed. A concept is said to be a root if it does not has any parent concept. The data model allows the existence of multiple roots.

PART II

CHAPTER 4

SYSTEM OVERVIEW

4.1 System Objectives

We have built a prototype of a DBMS using SAM* as the conceptual data modelling tool. The overall objective is to facilitate and enhance the access of information from the database system.

4.1.1 Data Level

At the data level, it is aimed at relieving user of memorizing details of logical organization of data within the database system, and at the same time offering user a flexible method to formulate his requirement.

In conventional DBMS, there are usually two approaches for user to access a database. One approach requires user to learn a query language as well as understand the logical structure of the database system, then user can use a query manipulation language, like SQL, to formulate query. This method is flexible enough to retrieve a wide variety of information, but burden the user with

explicitly addressing the logical structure of the underlying database. For example, to use SQL, user must know the names of tables and their corresponding field names. An expert user of the database system may prefer this approach. The other approach allows user to rely on application programs to access the database system. This relieves the user of understanding the logical structure at the expense of a low degree of flexibility in formulating a query. Whenever new request arises, the application program must be modified, usually through an intermediator, which may take a lot of time. This approach may be suitable for novice user or a routine application of database system.

In our system, we develop a new query manipulation language that only requires the user to has a picture on a conceptual data model (which is expected to be familiar with the user) and understand the abstraction mechanisms employed by the data model. Query from user is pre-processed by the system, when necessary, and is translated into data manipulation language for the underlying database system. The user needs not care about the logical structure of the database system, regardless it is relational, hierarchical or network model based.

4.1.2 Meta-data Level

By meta-data level, we mean data that describe data in the database, i.e. a schema. At this level, it is aimed at allowing user to perceive a complete picture

of the information being stored in the database system. This is done by providing facilities to user to inquire about the schema. For example, suppose a user finds that there is no information about interest rate of a fund. Then, by traversing the data model, say through the child concept of "fund", the user can reveal other relevant information offered by the system.

User can manipulate the system easier because user and system share a common perception on the application domain. This common vehicle is built by explicitly expressing semantic in the data model. Thus, the power of the system lies on the comprehensiveness of the model being built. Similar to the case in rule-based expert system where its power lies on the completeness of the set of rules.

4.2 System Characteristics

Allowance of raising query at meta-level is, to the author's best knowledge, a special characteristics of our system when compare with other implementations [BaLe88, JaGu88, LyVi87, Tsur84]. Though SAM* has been discussed in several literatures [HuKi87, PeMa88, Su83, Su86], no implementation of this model is reported. Thus, our work is useful in illustrating the practical aspect of this data model. The prerequisite for a user to work with our system is that the user should know the semantic meaning of the seven built-in associations of SAM* and the use of two query manipulation languages, each for the data level and meta-data

香港中文大學圖書館藏書

level data manipulation. As entities and their relationships as described in the schema are expected to be easily grasped by the user, it should take not long time for user to get used to the system.

The system is designed in such a way that user can access the database system either interactively or through a host language with embedded query languages. To facilitate the query formulation in interactively mode, the following measures are employed to make the syntactic structure of query as simple as possible:

1. the system has list processing ability, thus reduce the number of dummy variables being used. Variable owns a list of pointers pointing to concepts. Thus, an operation acts upon such variable is equivalent to executing an iterative statement with each iteration one of the concepts is processed by the operation.
2. pre-process (to be specific is automatically join and/or union tables, and qualify field names) query from user, hence, lessen user to concern with details of query formulation.

4.3 Design Considerations

As a database system, the system will be frequently inquired of information about occurrences of concepts, which could be a voluminous amount of data. Thus, occurrences of concepts are prefer to be stored in a DBMS which can well-

organize the data, and hence allow easy retrieval of them as needed. The data model make no assumption on which type of DBMS should be used. In this system, a relational DBMS is chosen because of its popularity and ease of use.

The underlying relational database should be constructed in such a way that the manipulation of query can be facilitated. Since query usually focus on concepts, we use a relational base table[1] [Date86] to store occurrences of a concept. However, not every concept should own a base table for three reasons.

Firstly, information of occurrences of a concept (usually the membership association) may not be able to exist independently. For example, current high price characterizes a stock through aggregation association. It is meaningful to talk about occurrences of the concept having that aggregation association, e.g. HongKong Bank has current high price of HK$ 12.5. However, it is meaningless to talk about occurrences of current high price without referring to a stock. In fact, a base table should be created for concept that represents entities rather than an attributes.

Secondly, on observed that occurrence of a more abstract concept can be expressed in terms of that of less abstract concepts, only the occurrences of "primitive" concept is physically stored in a base table. Occurrence of non-

---

1 A base table is an autonomous, named table. It is "really exist", in the sense that, for each row of a base table, there really is something physically stored.

primitive concept is constructed from the primitive one, using the algorithm stated in Appendix B. This can reduce not only the number of base tables to be maintained but also the amount of duplicated data. Database administrator takes the responsibility of deciding which concept is to be a primitive one. Usually, primitive concept is the one with association type of aggregation, interaction, or summary.

Thirdly, some association types, typically the composition and cross-product associations, are used to define a group of entities, and that group is to be manipulated as a whole. Under this situation, there is no need to assign a base table for concepts of these association types.

CHAPTER 5

IMPLEMENTATION CONSIDERATIONS

5.1 Introduction

To illustrate the ability of a DBMS using semantic data modelling approach, a prototype of a database system has already been built with effectiveness as the primary concern. The implementation involves four parts:

a) development of a data definition language for schema using SAM*;
b) construction of a schema compiler that build necessary data structure at run-time of the system;
c) design of query manipulation languages, and development of language interpreter for user to access the database; and
d) map the schema into a logical organization of an underlying database system, preparing the DBMS for data manipulation.

The underlying relational database system is built using ORACLE[2] that runs on an IBM-AT[3]. As ORACLE runs on a wide range of computer systems,

[2] ORACLE is a trademark of Oracle Corporation.

[3] IBM is a trademark of International Business Machines Corporation.

our system can be port to mini or mainframe as required. We use Microsoft C[4] to develop the schema compiler and the query manipulation language interpreter. Also, it acts as the host language with which to access the database. We have investigated the possibility of using VAX Rdb as the underlying database system, and access it through VAX Pascal or C. Nevertheless, VAX Rdb does not provide join operation on relations, thus, this alternative is discarded.

5.2 Data Definition Language for Schema

Schema defines characteristics of and relationships among concepts using a data definition language (DDL). The syntax of the DDL for SAM* is presented in Appendix C. A schema consists a set of statements. Each of them describes an association and has the following information: concept name corresponds to the association, its child concepts, and integrity constraints imposed on the association. For example, the schema corresponds to the data model as shown figure 11 is presented in table 1. The name within square bracket is the field names that are used in the underlying DBMS. Figure 12 shows the base tables for the sample schema.

To simplify the implementation, we have put restrictions on the use of associations. Yet, these restrictions will not affect the expressiveness of the data

[4] Microsoft is a trademark of Microsoft Corporation.

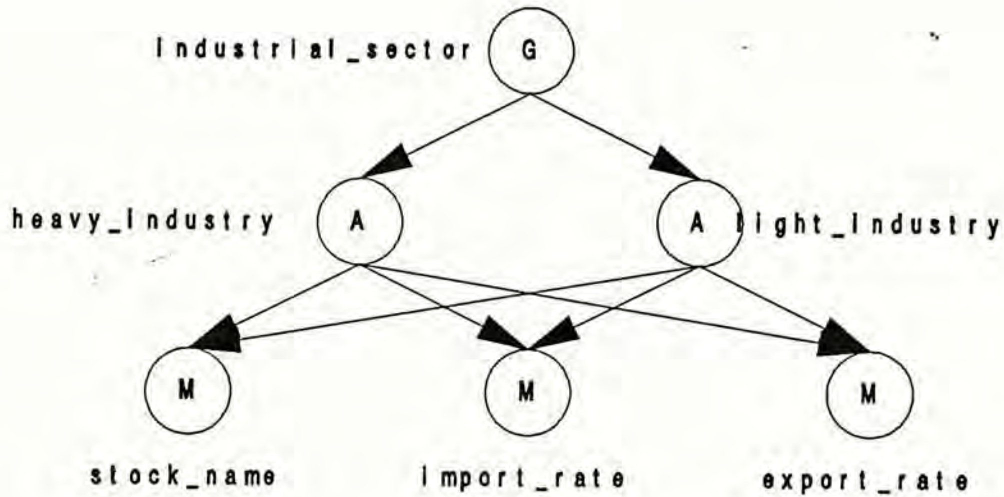

Figure 11 A Sample Data Model

```
generalization :: industrial_sector, root,
     heavy_industry, light_industry;
     with sx-sx;
     under constraint1;

aggregation :: heavy_industry, base,
     [stock_name key, import_rate, export_rate]
     stock_name key, import_rate, export_rate;
     under constraint2;

aggregation :: light_industry, base,
     [stock_name key, import_rate, export_rate]
     stock_name key, import_rate, export_rate;
     under constraint3;
```

Table 1 A Sample Schema

heavy_industry

stock_name	import_rate	export_rate
⋮	⋮	⋮

key : stock_name

light_industry

stock_name	import_rate	export_rate
⋮	⋮	⋮

key : stock_name

Figure 12 Base Table for the Sample Data Model

model.

We require concepts being generalized should have a common unique key to identify their occurrences. This is reasonable as generalization association pick out generic nature of its components, and we just require this generic nature of each component concept should, syntactically, take a common form.

Each interaction association should not describe two or more independent interactions. In case a concept do involve two or more independent interactions, we may assign a new interaction associated concept to each of the interactions, and group them by a generalization association.

Through a concept having cross-product association, user can only retrieve classification of categories, not occurrences of the child concepts of that concept. The same apply to concept having composition association which group dissimilar concepts and itself is treated as a single class. To retrieve information, user should use other associations to describe the child concepts.

Values in the summarizing concepts are stored, rather than being derived from other values in the database. Thus, whenever the data to be summarized is changed, the summarizing concept should be updated.

The schema is mapped into a logical organization of an underlying DBMS

in two phrases. Firstly, the characteristics of and relationships among concepts are mapped into a directed acyclic graph. Secondly, an underlying DBMS is prepared to store occurrences of concepts. A framework of the mapping is shown in figure 13.

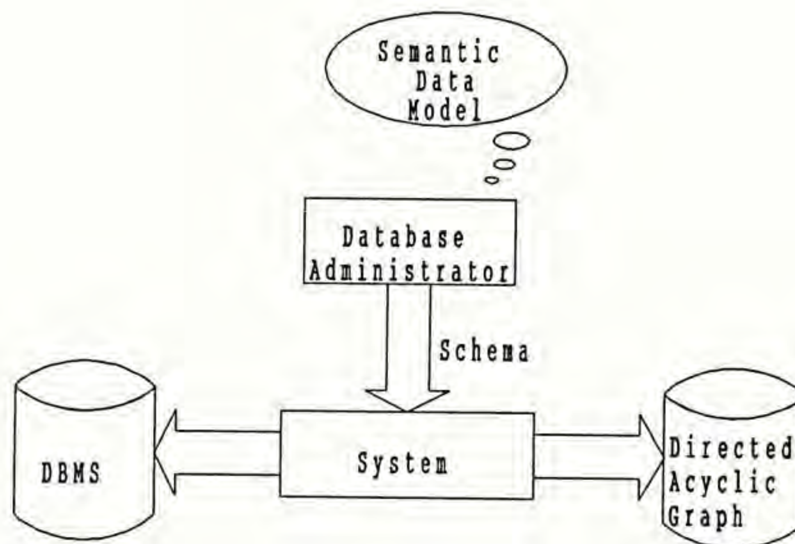

Figure 13 Mapping Schema Into Logical Organization

5.3 Construction of Directed Acyclic Graph

A directed acyclic graph is represented by a set of vertices and edges. Acyclic means that there is no cycle in the graph. From now on, in Part II, graph refers to a directed acyclic one. A vertex is assigned to each concept in the schema. Edge is connected from vertex of a parent concept to vertex of its child concepts, as described in the schema. Each vertex owns a record having the following information:

a) name: name of the concept as shown in the schema;

b) asso: association type of the concept;

c) parent: a pointer pointing to a list of vertices corresponding to parent concepts of the concept, or is a null pointer if the concept has no parent;

d) child: a pointer pointing to a list of vertices corresponding to child concepts of the concept, or is a null pointer if the concept has no child;

e) root: indicates if the concept is a root;

f) constraints: for generalization association, user should specify set relationships between component concepts: set subset, set exclusive, set equal, and set intersection. For interaction association, user should specify the mapping relationships: one-to-one, one-to-many, and many-to-many. User can impose other constraints as needed;

g) base: indicates if the concept has assigned a base table;

h) key: if the concept owns a base table, key is a pointer pointing to a list of child concepts that form the primary key of the base table, as required by relational data model; otherwise, key is a null pointer.

5.4 Query Manipulation Language

Two query manipulation languages are developed: Semantic Manipulation Language (SML) to access the graph and Occurrence Manipulation Language

(OML) to retrieve occurrences of concepts. User can toggle between the use of these two languages. In either mode, the corresponding language interpreter is invoked. A framework of the query manipulation process is shown in figure 14.

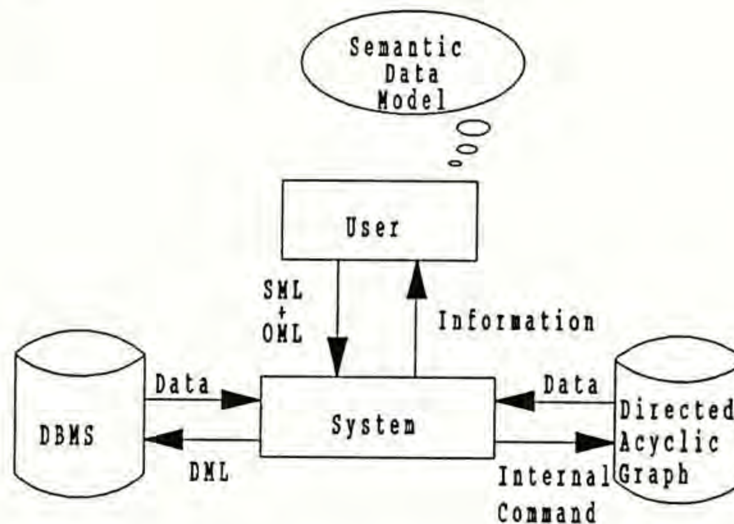

Figure 14 Query Manipulation Process

5.4.1 Semantic Manipulation Language

SML is for revealing information about the graph. Basic operations upon the graph fall into three categories: locate concepts, retrieve information of concepts, and find paths between two concepts. Syntax of the SML is given in Appendix D.

5.4.1.1 Locate Concepts

format: variable = find_node(restriction template)

The function returns a pointer pointing to a list of one or more vertices satisfying the requirement as specified in the restriction template. The returned pointer is assigned into the variable on the left of the equal sign. There are five types of restriction templates referring information about a concept: name, parent concept, child concept, association type, key identifier(s) of the concept. For example, refer back to the data model as shown in figure 11,

v = find_node(child = "stock_name")

returns a pointer to v, pointing to a list of vertices having "stock_name" as their child. The system returns a pointer pointing to {"heavy_industry", "light_industry"}.

5.4.1.2 Retrieve Information About Concepts

format: ? projection template(V)

Given a pointer pointing to a list of vertices V or a name of a concept, the function returns information, as specified in projection template, about the vertices or the concept respectively. For example,

? asso(v)

where v is a variable having a pointer pointing to a list of vertices, will return the association types of the vertices one by one. Suppose v is the variable as returned in the above find_node command, then the system returns the following

information:

The association types are:

1. aggregation

2. aggregation

Internally, the function is executed by an iterative statement as follows:

for each element v_i in v do

print association type of v_i

5.4.1.3 Find a Path Between Two Concepts

format: find_path(v_1, v_2)

v_1 and v_2 are pointers. Each of them points to one vertex, or is name of concept. The function returns all paths between them. For example,

find_path("industrial_sector", "stock_name")

returns intermediate concept names along a path from the concept "industrial_sector" to the concept "stock_name". Thus, the system responds as:

Path(s) through them include(s):

1. "industrial_sector", "heavy_industry", "stock_name";

2. "industrial_sector", "light_industry", "stock_name";

Semantically, this function finds out the way a concept is related to another.

5.4.2 Occurrence Manipulation Language

User can request information about occurrences of concepts using query in the following format,

select	child concepts
from	parent concepts
where	restriction template

The child concept in the select clause needs not be an immediate child of the parent concept in the from clause, but it should be a less abstract concept than the parent concept. The system displays occurrences of the child concepts that satisfy the restriction template as specified in the where clause.

For the simplest case, the parent concept owns a base table and the concepts addressed in both select and where clauses have corresponding field names in the base table. Then, the query can be directly executed by ORACLE. For example, refers to the data model as shown in figure 11, and consider this query, "Displays the stock names and export rates of those light industry stocks whose export rate is greater than 10%.",

display	stock_name, export_rate
from	light_industry
where	export_rate > 10%

As light industry owns a base table, this query can be directly executed.

For other cases, the system will automatically collect relevant base tables, perform join and/or union operations, and qualify field name with its base table. The central idea of the algorithm of query pre-processing runs in four steps. Firstly, collect those base tables whose field names are addressed in either the select or where clause, with their corresponding concepts are child of the concept mentioned in the from clause. Secondly, for any two base tables, say B_1 and B_n, from the pool of base tables collected, pick out a set of base tables, say B_2, B_3, ..., B_{n-1}, such that B_1 and B_2 has a common attribute, so does B_2 and B_3, ..., as well as B_{n-1} and B_n. Also, B_1, B_2, ..., B_n should lay on a path from B_1 to B_n. Thirdly, perform join operation to join all the base tables collected. Finally, attributes are qualified, and a query in ORACLE format is formulated and is executed by ORACLE. In case the concept addressed in the from clause has generalization association, union operation is performed to group the base tables collected, instead of joining them.

Consider another example, "Display the stock names and their export rate for all stocks belonging to industrial sector and has export rate greater than 10%.",

```
display     stock_name, export_rate
from        industrial_sector
where       export_rate > 10%
```

Here, as industrial sector does not has a base table, and "industrial_sector" has a generalization association, the system translates it into

```
select    stock_name, export_rate
from      heavy_industry
where     export_rate > 10%
union
select    stock_name, export_rate
from      light_industry
where     export_rate > 10%
```

With these extension, user can focus on what he want without caring about the details of how to formulate a query.

5.5 Examples

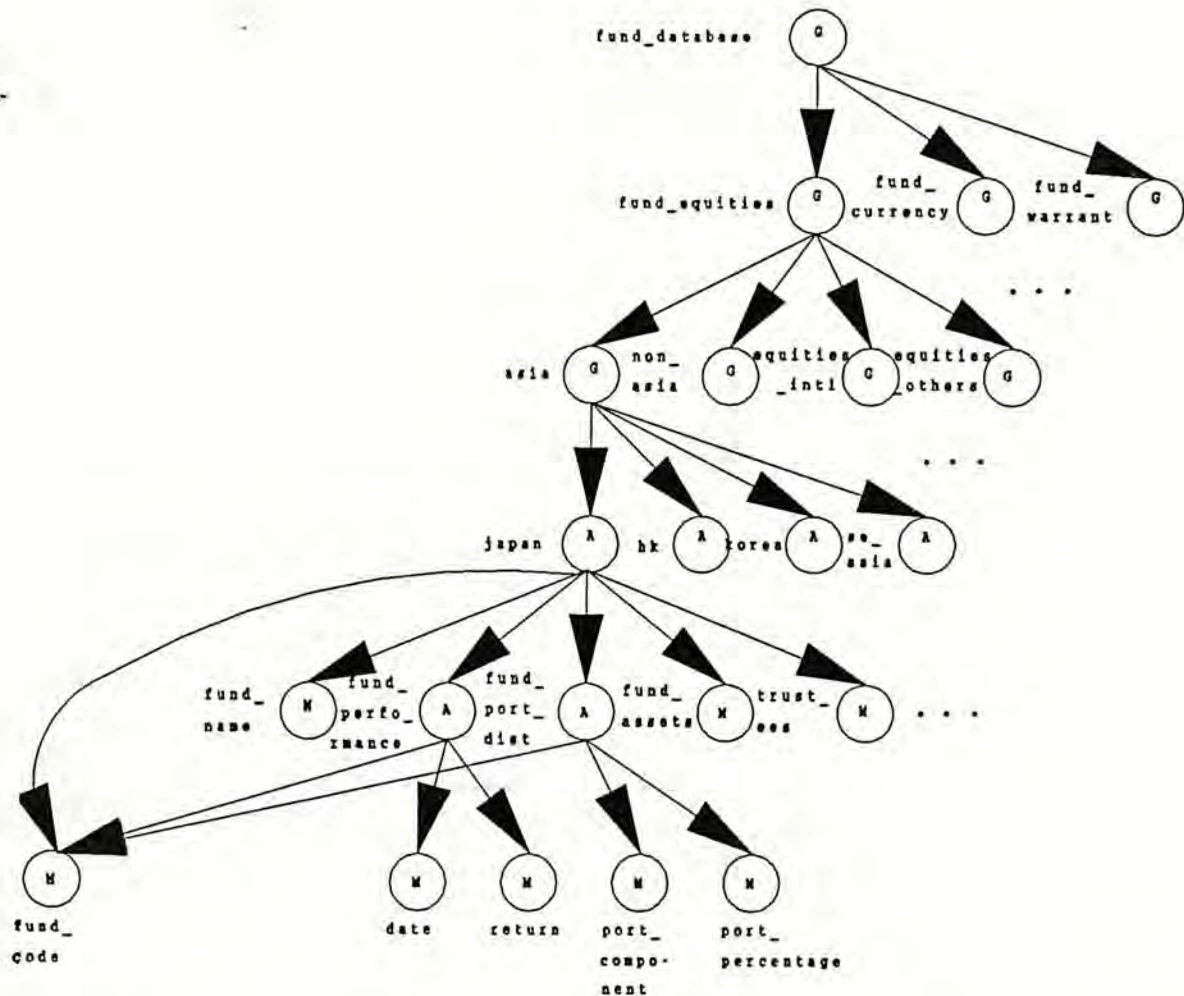

Figure 15 A Partial Data Model for Fund Investment

Example 1

Example 1 and 2 refer to the data model as shown in figure 15. It describes a model for fund investment. The full model is shown in Appendix E which is based on a year book about fund information [Hon89a, Hon89b]. The database for funds classifies funds into three categories: equities based, currency

based, and warrant based. Each of them is further divided into finer classes. Thus, forming a hierarchy of generalization associations. A fund is then characterized by a set of attributes, like those describing japanese funds.

Consider this query, "Display the fund names and their assets for those japanese equities fund with return on May greater than 20%.",

```
select      fund_name fund_assets
from        japan
where       (date = 'May') and (return > 20)
```

which is translated by the system into

```
select      japan.fund_assets, japan.fund_name
from        fund_performance, japan
where       ((fund_performance.date = 'May') AND
            (fund_performance.return > 20)) AND
            fund_performance.fund_code = japan.fund_code
```

Occurrences of "fund_name" is in the table japan, while that of "return" is in the table fund_performance. Thus, these two tables are joined by the system to obtain the answer.

Example 2

Consider this query, "Display the fund names and their annual charges of those asia funds that have initial charges smaller than 5%.",

```
select      fund_name annual_charge
from        asia
where       initial_charge < 5%
```

which will be translated into

```
select      fund_name, annual_charge
from        japan
where       initial_charge < 5%
union
select      fund_name, annual_charge
from        hk
where       initial_charge < 5%
union
select      fund_name, annual_charge
from        korea
where       initial_charge < 5%
union
```

```
select      fund_name, annual_charge
from        se_asia
where       initial_charge < 5%
```

As Asia funds consists Japan, HongKong, Korea, and South East Asia funds, the answer is obtained by taking a union on all funds from all these countries.

Example 3

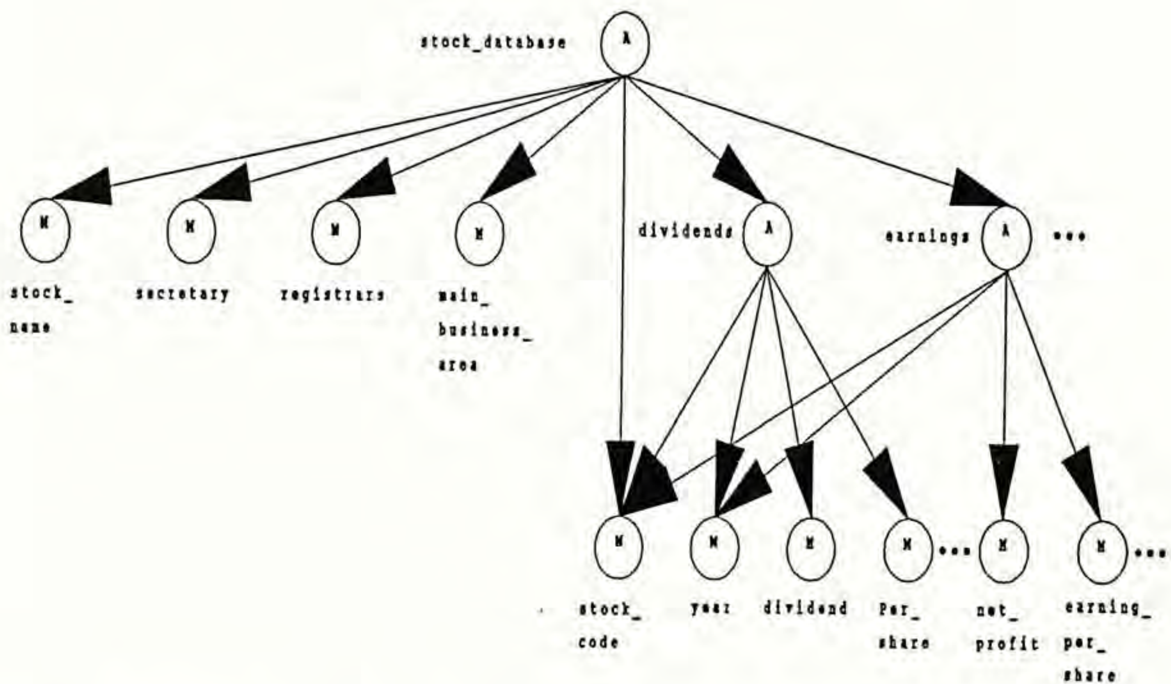

Figure 16 A Partial Data Model for Stock Investment

Example 3 refers to a data model as shown in figure 16. It models a stock investment analysis database. Each stock is identified by its code, and is characterized by a set of attributes in membership or aggregation associations. It illustrates the power of the use of hierarchy of aggregation associations. A full

data model is shown in Appendix F.

Consider this query, "Display the names, business activities and earning per share in 1989 for those stocks that have earning per share greater than 1.",

```
select      stock_name activities
            earning_per_share
            net_profit
from        stock_database
where       earning_per_share > 1 and
            dividends.year = 1989 and
            earnings.year = 1989
```

will be translated into

```
select      stock_database.stock_name,
            business_activities.activities,
            dividends.earning_per_share,
            earnings.net_profit
from        stock_database, business_activities,
            dividends, earnings
where       (earnings.earning_per_share > 1 AND
            dividends.year = 1989 AND
            earnings.year = 1989) AND
            earnings.stock_code = stock_database.stock_code AND
            dividends.stock_code = stock_database.stock_code AND
```

business_activities.stock_code =

stock_database.stock_code

In this example, information of a stock is scattered in several tables. However, the user only need to specify the concept he concerns (e.g. "activities", "earning_per_share", and "net_profit") without knowing exactly the relations (i.e. "business_activities", "dividends", and "earnings") that the concept belongs to. An exceptional case is when addressing concepts having multiple parents. In this case, the concept must be unambiguously qualified. This can be avoided with sophisticated programming technique that can produce required qualification inferred from relevant concepts.

The above dialogue can not happen in conventional database system if the user do not know the logical organization of the underlying database system.

CHAPTER 6

RESULTS AND DISCUSSIONS

In this chapter, we discuss benefits gain from using SAM* as the data model over relational data model. Limitations of record-based data model has been criticized by Kent [79]. Although we map SAM* on a relational database, we has overcome several drawbacks of relational data model with the help of the semantic knowledge embed within the SAM* schema.

6.1 Allow Non-Homogeneity of Facts about Entities

In relational database, every record of a table must be structurally the same, i.e. the types and number of attributes of every record of a table must be the same. Semantically, each record represents an entity, and similar entities should be put in the same table for ease of reference. However, similar entities may has different set of characterizing attributes to describe them. In this case, a single table cannot be used to represent all of them. For example, manager and messenger are both employee. Suppose only manager can own a company car, then it is meaningless to assign an attribute about company car to messenger. Thus, we cannot use one table to store all of the information about employee.

There are usually two approaches in solving this problem. One is to include all relevant attributes under the same roof, and the other is to use separate table for each set of characterizing attributes. The former introduces null fields in records, while the later scatters information about similar entities over several tables which puts burden on user to re-organize them at retrieval time.

Using SAM*, this situation can be modelled by generalization association. Entities belonging to a generic type are grouped under a generalization association. Each subtype has its own concept, and can be further characterized by its own set of attributes. Thus, all entities can be accessed at the generalized concept, and at the same time information about particular entities can be retrieved from the constituting concepts.

6.2 Field Name is Information

Field name is a place holder in conventional relational database. User cannot raise query about field names itself (e.g. "Which table has this field name?"), nor the system returns field name as answer (e.g. "What are the common field names of two tables?"). In our system, through the directed acyclic graph corresponding to the schema, user can not only inquire about child concepts of a concept but also query any relevant information about the interested concept, thus enriching one's knowledge about available information stored in the database.

6.3 Description of Group of Information

In a tuple, nonkey attributes describe the key one. The key attribute usually represent a single entity. There is no direct method in describing a group of entities. Encoding may be used so that a value in the key field do represent a pre-defined set of entities, but this should introduce additional mechanism to guarantee the correspondence. In SAM*, summary association serves this suppose. Under this association, we can attach two types of concepts: one is a composition or cross-product association which defines a set of entities, and the other is an aggregation association or simply a set of membership association, describing the set of entities. This is useful not only for statistical but also qualitative description. For example, one can use summary association to describe the source of information for a set of entities.

6.4 Explicitly Description of Interaction

In relational database, a table has two semantic functions: it groups attributes that are interacting one and the others, and groups attributes to describe another attributes. The system cannot differentiate between these two functions. In case, there is only one semantic function in a tuple, we can use the table name to represent this relationship, indicating the semantic of the table. However, when more than one semantic function appears within a table, we cannot explicitly expressing every semantic function of the table.

In SAM*, we explicitly use interaction and aggregation associations to reveal such relationships, allowing user to understand what information is stored in the database. Also, we can use a generalization association over interaction or aggregation association to solve the problem that a key attribute has more than one depending attribute.

6.5 Information about Entities

Due to normalization, a table will be decomposed into smaller tables, having information about an entities being stored in separate tables. However, for relational database, schema does not provide any details on the relationship between decomposed table of an entities. Problem may arise when an entity is processed (insert, deleted, or modified) which may leave the database in an inconsistent state. Integrity constraints may help but they are usually hidden in the underlying integrity constraint handling mechanism, so that most users will not know these valuable information.

In our system, relevant information about an entity can be easily retrieved from the graph, thus, user can completely inquiry about an entity, and can has more confident in manipulating entities.

6.6 Automatically Joining Tables

Our system can automatically join two or more tables when this operation is implied by the query. For example, refer to figure 15, user can ask this query, "Display the date formed of those japanese funds whose annual return is greater than 20%"

```
select      date_formed
from        japan
where       return > = 20
```

Here, the occurrences of concepts "date_formed" and "return" are in two separate tables (the former is under the concept "japan", and the latter under "fund_performance"). Thus, the two tables must be joined in order to produce the answer. Our system does this for the user, and produces this query,

```
select      japan.date_formed
from        japan, fund_performance
where       (fund_performance.return > = 20) and
            (japan.find_code = fund_performance.fund_code)
```

the second condition in the "where" clause is necessary for joining two relations where "fund_code" is the common primary key for both tables.

6.7 Automatically Union Tables

Our system can automatically take a union operation on tables when

necessary. Suppose, currency fund generalizes both US funds and Japan funds. We can use a single query to retrieve all currency fund information from the generic concept "fund_currency" as in

```
select      fund_name
from        fund_currency
```

This is automatically translated into

```
select      fund_name
from        fund_us
union
select      fund_name
from        fund_japanese
```

6.8 Automatically Select Tables

Our system is able to select necessary tables to form a valid query, and qualify attribute. For example, refer to figure 16 and consider this query,

```
select      net_profit
from        stock_database
where       stock_code = 15
```

Here, the user does not require to know which table does the occurrences of "net_profit" reside. Instead, it is sufficient to specify a more abstract concept it belongs to, i.e. "stock_database". The system produces the following query,

```
select      net_profit
from        earnings
where       stock_code = 15
```

This capability is of particular help in case a lot of concepts are in many different tables.

6.9 Ambiguity

When addressing a concept that has more than one parent concept, user is required to specify which concept he refers to. For example, consider figure 16 and the following query, "Display the net profit for the stock with code equal to 15 in 1989". Here, the concept "year" will be addressed. As year has more than one parent ("dividends", "earnings", and etc.), the user must qualify "year" with the parent concept that he refers to. Hence, the query should be raised as,

```
select      net_profit
from        stock_database
where       stock_code = 15 and earnings.year = 1989
```

6.10 Normalization

Normalization is a process in logical design of a relational database system that aims at avoiding various update anomalies. Usually, normalization is fulfilled by decomposing the original table into smaller tables. The use of SAM* as a data

model does not affect the process of normalization, since SAM* does not impose any constraint on the composition of the child concepts of a concept, hence the grouping of field names of a table.

Consider the Boyce/Codd Normal Form (BCNF) which require that every determinant[5] is a candidate key[6] [Date 86]. A table, called SSP (supplier number-supplier name-part number relation), is shown in figure 17.

S#	SNAME	P#
S1	Smith	P1
S1	Smith	P2
S1	Smith	P3
S1	Smith	P4
.	.	.

Figure 17 A Table not in BCNF

SSP is not in BCNF because both S# and SNAME are determinants (P# depends on S# and SNAME) but are not candidate keys (there are duplication for S# and SNAME in SSP) for the relation. To model this information and have the resulting tables satisfying BCNF, one can decompose SSP into two tables: one has supplier number and supplier name as attributes, and the other has supplier number (or supplier name) and part number as attributes. This situation can be modelled in SAM* as in figure 18. The concepts "supplier_part" and "supplier"

[5] A determinant is any attribute on which some other attribute is fully functionally dependent.

[6] The set of attributes $K = (A_i, A_j, \ldots, A_k)$ of R is said to be a candidate key of R if and only if it satisfies the following two time-independent properties: 1) uniqueness. At any given time, no two distinct tuples of R have the same value for A_i, the same value for A_j, ..., and the same value for A_k. 2) minimality. None of A_i, A_j, ..., A_k can be discarded from K without destroying the uniqueness property.

own base tables. The base table of the former can has supplier number (or supplier name) and part number as attributes; while that of the latter has supplier number and supplier name as attributes.

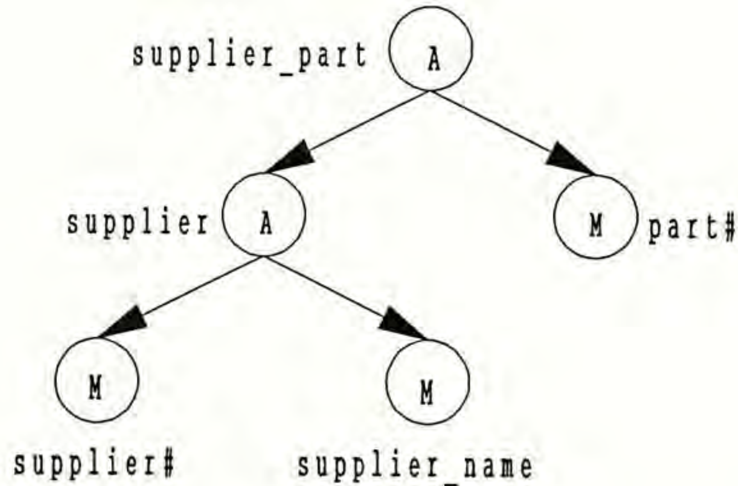

Figure 18 An Example on Handling Normalization

Alternatively, the schema can be modelled as in figure 19, where the concepts "supplier" and "supplier_part" own base tables, with the former having supplier number and supplier name as attributes, and the latter having supplier name and part number as attributes. The resulting tables are also in BCNF.

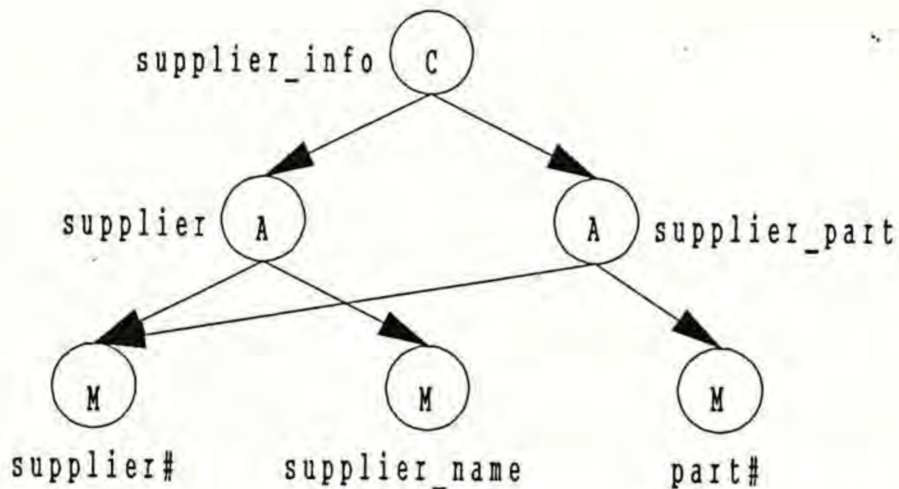

Figure 19 Another Example on Handling Normalization

6.11 Update

Update in a database using semantic data model should be carried out with special care, or else it will lead the database into an inconsistent state. By update, we mean insert, modify, and delete of entities and/or some of their attributes values. In this section, we discuss update handling mechanisms offered by various implementations of semantic data models.

DAPLEX leaves the responsibility of maintaining the consistency to user. User can specify the operation to be done (specified in a so-called *perform* clause) when a particular update is made (specified in *using* clause). As commented in [Ship81],

> It is important to point out that this facility merely allows users to provide the illusion of derived data updating. The system does not, for example, validate the PERFORM ... USING statement to ensure that it results in the intended derived data update.

Even in recent implementation of DAPLEX [BaLe88, LyVi87], there is no mechanism reported for handling update.

IFO [Abit87] takes a theoretical approach to analyze update propagation. An *Update Propagation Graph* is constructed, and any update at the leave node is then propagated upward to the root of the graph. The discussion is under a restricted environment: 1) only is-a (specialization and generalization) relations are considered; 2) there is no cycle in the update propagation graph; and 3) there is no functional relation in the graph. The third restriction makes the discussion of little value in practical case, where an update to an object will usually affect other object that is functionally depends on the former one. For example, consider a schema describing the situation of a course enrolls students. Here, enroll is a functional relation mapping a course to a set of students. When a course is deleted, so does the students who have taken that course. But, IFO cannot handle this common phenomenon.

The implementation of SDM by [JaGu88] takes a similar approach to that of IFO, where it only considers update propagation along subclass or superclass path starting from the entity to be updated. When an entity is inserted, all of its ancestors will be inserted with information indicating the existence of the new entity. When an entity is deleted, all its subclass instances are deleted while its superclass instances are not affected. For modification, immediate and inherited attributes of a class can be changed, but the effect is not automatically

propagated.

An implementation of GEM semantic data model is reported in [Tsur84]. The system allows user to formulate query to append, modify, and delete entities. However, it will reject any attempt to delete a entity that still has been referenced by other entity. Thus, it is also the user's responsibility to perform all the chaining operations induced from an update.

In SAM*, update propagation can be done with the help of the graph. The direction of propagation (upward or downward in the graph) depends on the operation and the association of the concepts being involved. For aggregation association, the existence of an occurrence of a child concept depends on that of its parent concept. Thus, an addition of an occurrence of a child concept should not be performed before the addition of the corresponding occurrence of its parent concept; while a deletion of an occurrence of a parent concept must has its corresponding occurrences of child concepts being deleted too. For interaction association, if an interaction exist, then there should has corresponding occurrences of its child concepts. Thus, an addition of an occurrence of an interaction must be done after the addition of the corresponding occurrences of its child concepts. An deletion of an occurrence of a child concept should has the deletion of the corresponding occurrence of its parent concept. To illustrate the situation, refer to the data model in figure 16, a delete of a stock, say with stock code equal to 15, can be specified as,

```
delete    entity
from      stock_database
where     stock_code = 15
```

Since "stock_database" owns an aggregation association, the deletion of an occurrence of this concept (as the query request to delete an entity of stock_database) must be done after the deletion of corresponding occurrences of its child concept. The system is expected to translate it into several queries with each of them the stock with stock code of 15 should be deleted from the occurrences of a child concept of stock_database. Two of these queries, which are in SQL form, are:

```
delete
from      dividends
where     stock_code = 15
and
delete
from      earnings
where     stock_code = 15
```

Only after these operations are done should the occurrence, the stock with code 15, of "stock_database" be deleted,

```
delete
from      stock_database
where     stock_code = 15
```

Besides manipulation of entity as a whole, the user should be allowed to perform update at a concept without propagation. Suppose the user just want to delete information about earning of a stock, then this query

```
delete
from        earnings
where       stock_code = 15
```

should not induce any other operation.

From the above discussion, we can get the idea that there is no simple solution for performing a consistent update. The problem is that a single update may induce other operations. If the update is done automatically by the system, then the induction of other operations can only solely based the data model. Thus, not only the model itself must completely represent the application domain, but also the system must make no error in interpreting the data model. For flexibility reasons, the system should offer user the ability to control update propagation.

PART III

CHAPTER 7

SCHEMA VERIFICATION

7.1 Introduction

Schema is a description of an application environment. It structurally defines characteristics of and relationships between entities that are relevant to the application. Usually, schema is equipped with constraints to restrict possible instances of entities. Constraints are the means through which user or database administrator can specify semantic restrictions onto the database.

In conventional database system, schema is of rather static nature. Seldom literatures address the problem of re-design of schema. The situation can be acceptable in a very stable application environment. However, there do have circumstances under which schema is subjected to modification. For example,

a) incompleteness of the schema itself;
b) incompatibility between user specified constraints and those imposed implicitly by the underlying data model;
c) changing of the application environment, e.g. a schema for stock investment analyzing database should be updated if a new investment

tool is introduced to the market; and/or

d) incompatibility between requirement specified by users under a multi-user environment, where each user has his own interest in a particular aspect of the database.

The situation as described in point (b) becomes obvious when working with a database using semantic data modelling approach. With enriched set of semantic constructs to describe data as well as operators to operate on them, the data model usually impose constraints to guarantee consistent usage of these additional facilities.

Once the schema is to be revised, we will concern the following questions: "Is the resulting schema still a valid description of the real-world data?" or "Will user imposes extra requirements that are inconsistent to the existing ones?".

The author would like to term this problem as schema verification which is defined as a process that determines the existence of contradicting constraints within a schema, assuming every requirement imposed by users is expressed as constraints. Serious modification must rely on the database administrator to re-design the whole schema from the beginning. However, in case the change is of small size, the change happens from time to time, and/or the schema is of large scale, the process becomes increasing tedious and difficult. The solution through re-design the whole database from the beginning seems to be time-consuming and expensive. In this part, we proposal a mechanical method to help the database

administrator to perform schema verification. It should be emphasized that the discussion does not restrict to database using SAM*. In fact, the result can be applied to any database system that face the problem of having to change the schema.

7.2 Need of Schema Verification

As the schema grows more and more d, there may has problem of inconsistency. Consider the schema as shown in figure 20. At one time, the user perceives "industrial_sector" and "hotel_sector" are two mutually exclusive concepts, as indicated by the "SX-SX" constraint. However, at the other time, the user thinks that "hotel" is a kind of "light_industry", as well as "hotel_sector". This contradicts to the set exclusion constraint. The result is that it is forbidden to add any data under the concept "hotel". If the user does not aware of such a fault in the schema and try to retrieve occurrence of the concept "hotel", which is empty, then the user may wrongly conclude that there is no hotel business in Hong Kong!

Such mistake occurs when schema designer has inconsistent view on the concepts. The situation is not uncommon if the schema is developed by more than one designer or the schema is modified by a third party, say user. This shortcoming can be amended, if contradicting constraints within the schema can be checked out before the database is used, which is the job of schema verification.

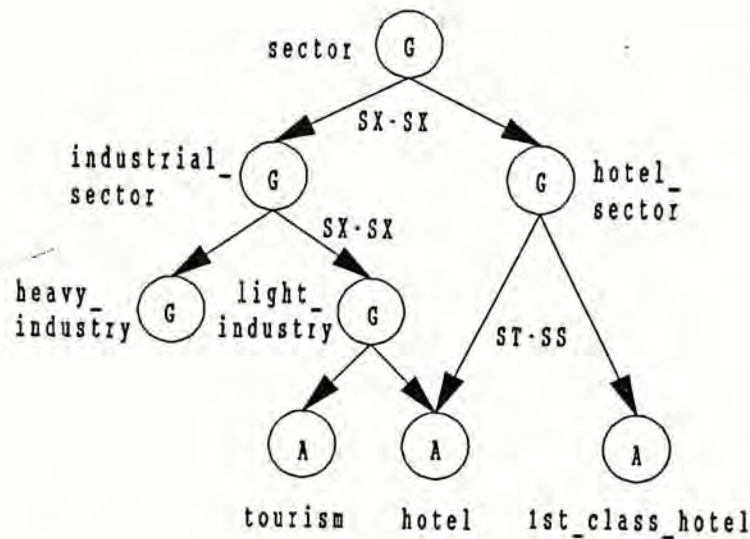

Figure 20 Example of an Inconsistent Data Model

7.3 Integrity Constraint Handling VS Schema Verification

Another field of research in database communities that relates to schema verification is under the title of integrity constraint handling. Integrity constraint characterizes valid database states, and impose restrictions on the possible state transitions of a database [GaMi84]. The very first issue of both problems is to represent constraints. Research has been done for expressing integrity constraints in relational database [Nico82] as well as deductive database [Ling87]. The result can be directly applied to schema verification.

The two issues are different in their very goals. Integrity constraints checking intend to derive mechanism to efficiently enforce constraints on the database during data manipulation. While in the schema verification, we try to check if the constraints themselves are contradicting to one another. Thus, the

former is carried out during the access of the database, while the latter is done at the compile time of the schema. In this sense, schema verification works at a meta-level.

In fact, the two issues maintain consistency of a database system from different perspective, and should be treated as a pair of complementary issues.

CHAPTER 8

AUTOMATIC THEOREM PROVING

8.1 Overview

Automatic theorem proving is a promising approach to schema verification. The main goal of research in automated theorem proving is to build programs that are effective in finding or helping to find proofs of theorems from mathematics and other fields of applications [BlHe85]. Stated in terms of automatic theorem proving, schema verification is try to find a proof for the conjectural theorem that new constraints (or conclusion) logically follow existing constraints (or premises).

A desirable prover is both effective and efficient. Soundness and completeness are one of the measures of effectiveness. A prover is sound if whenever it finds a proof for a theorem, the proof is valid for the theorem. It is complete if whenever a theorem is valid, a proof can be found by the prover to prove the theorem. In general, provers are sound but incomplete. A prover can be sound if only valid inference rules and premises are used. Incompleteness of provers follows from the Church's Thesis [Jeff89]. However, there is proof

procedure that is refutation complete, i.e. the prover guarantees to find a proof for an invalid theorem [GeNi87].

8.2 A Discussion on Some Automatic Theorem Proving Methods

In this section, four provers are discussed so that we can choose a suitable one to solve the problem of schema verification.

8.2.1 Resolution

Resolution [Robi66, Robi79, Love78] is the most well studied automated deduction procedure. It is sound and refutation complete. The resolution procedure is shown in table 2 [GeNi87]. Before the procedure is invoked, input formulas have to be converted into conjunctive normal forms, then, the conclusion is negated and added to the premises. The procedure is based on a single rule of reference known as modus ponens, i.e. (p -> q) and p logically imply q.

```
Procedure Resolution (Delta)
     Repeat if Termination(Delta) then
          Return(success);
          Phi := Choose(Delta);
          Psi := Choose(Delta);
          Chi := Resolvent(Phi, Psi);
          Delta := Concatenate(Delta, Chi);
     end;
```

Table 2 Resolution Procedure

Consider the following problem: given two premises, every manager is an employee and every employee has a mail box, we want to know if every manager has a mail box logically follows from the two premises. The problem is formulated as shown in table 3.

```
predicate:
          manager(x) : x is a manager
          employee(x) : x is a employee
          mail_box : x owns a mail box
premises:
          forall x (manager(x) -> employee(x))
          forall x (employee(x) -> mail_box(x))
conclusion:
          forall x (manager(x) -> mail_box(x))
```

Table 3 Axioms for 'mail-box' Problem

Once the conclusion is negated and all the clauses are converted into conjunctive normal forms, we perform the resolution procedure as depicted in figure 21.

The success of the procedure depends on the 'clever' choose of parent clauses to be resolved. Various strategies for improving performance of resolution are proposed [GeNi87, Rich83]. Some of them aim at eliminating useless clauses, e.g. pure-literal, tautology and subsumption elimination. Others restrict the candidate set of clauses for resolution, e.g. set of support, unit, input and linear resolution.

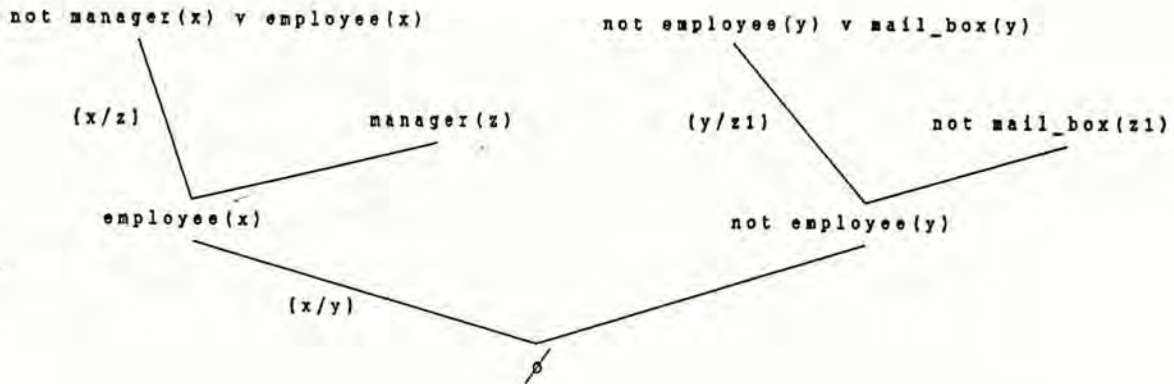

Figure 21 An Example of Resolution

8.2.2 Natural Deduction

Natural deduction [BlHe85] describes a collective prover that attempts to produce a 'natural' proof understandable to human, usually without prior converting formulas into any normal form. Gentzen's NK, as mentioned in [Bibe86], is one of the earliest development of natural deduction which simulates mathematician's reasoning. Natural deduction usually involves a wide variety of inference rules, and able to retain useful pragmatically information encode in the connectives, hence ease the use of heuristics to direct the prove procedure. In addition, as user can grasp each step in a proof, he can head the proof procedure to fruitful direction, resulting in a so-called 'interactive' prover [MiFe86]. Inference rules of a natural deduction system can be classified as introduction and exploitation rules [Fros86]. Some of them are shown as follows:

香港中文大學圖書館藏書

Introduction Rules:

from A and B infer A & B

from A infer A v B

from A |- {} infer -A

from A |- B infer A -> B

Exploitation Rules:

from A & B infer A

from A v B, A |- C, B |- C infer C

from A and -A infer {}

from A, A -> B infer B

where {} stands for a logically false formula. Starting from a supposition (or assumption), a proof is constructed by using these rules and the given premises.

A proof for the 'mail box' problem using natural deduction is shown as follows:

(a)	manager(x)	supposition
(b)	manager(x) -> employee(x)	given
(c)	employee(x)	by ->-exploitation of (a) and (b)
(d)	employee(x) -> mail_box(x)	given
(e)	mail_box(x)	by ->-exploitation of (c) and (d)
(f)	manager(x) |- mail_box(x)	from (a) to (e)
(g)	manager(x) -> mail_box(x)	by ->-introduction with (f)

It can be seen that the proof is easily understood, but the successful construction of a proof depends on the clever choice of supposition and inference rules.

8.2.3 Tableau Proof Methods

Tableau proof method is similar to natural deduction in the sense that it makes use of a wide variety of inference rules and the proof is expect to be easily followed by user. The difference is that a tableau proof is constructed from the negation of the theorem to be proved. As an illustration, consider a tableau-based theorem prover as described in [OpSu88]. It has four types of inference rules:

Alpha type:

```
- - M      M & N      -(M v N)     -(M = > N)
------     ---------  -----------  ---------------
 M          M          -M            M
            N          -N            -N
```

Beta type:

```
M v N      M = > N      -(M & N)       (M < = > N)        -(M <=>N)
--------   -----------  -------------  ----------------   ---------
M | N      -M | N       -M | -N         M | -N             M | -N
            N | -M                     -N | M
```

Universal Instantiation (UI):

```
forall x P(x)        - exist x P(x)
---------------      -----------------
    P(a)                 - P(a)
```

Existential Instantiation (EI):

```
exist x P(x)         - forall x P(x)
--------------       ------------------
    P(a)                 - P(a)
```

where M, N are formulas

P is predicate symbol

On the way to find a proof, it selects a premise upon which a rule of inference is applied. The premise is split into conjuncts by alpha type rules, and into disjuncts by beta type rules. EI and UI eliminate the universal and existential variables respectively by substituting a constant term into the variables. The proof can be visualized as a tree-like structure where each branch is a disjunct. A

branch is fail if it resolves to empty clause with one of its ancestors (M is an ancestor of N if N is produced by a series of decompositions form M). The procedure terminates when there is at least one truth-value assignment to all the atomic formulas of the premises. This proof procedure is shown to be refutation complete. The system can be incorporated with a set of heuristics to improve the proof. A proof of the 'mail box' problem for this prover is shown in figure 22.

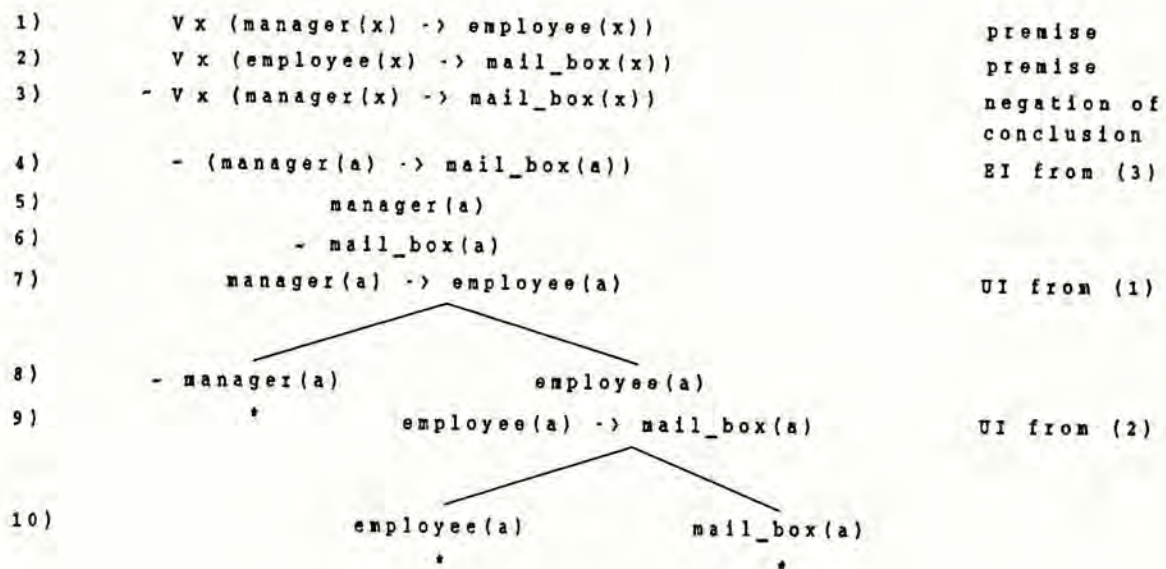

Figure 22 An Example of Tableau Proof Method

8.2.4 Connection Method

Connection method [Bibe81, Bib82a, Bib82b, Bibe83, HoBi82, Andr81] implicitly converts the input formula into disjunctive normal form (or conjunctive normal form), and concludes the original formula is invalid (valid) by showing that every disjunct (conjunct) is false (true). In this section, we focus the

discussion on the view of implicit conversion to disjunctive normal form. The case for conjunctive normal form is similar. Each disjunct through the input formula is called a path. A path is defined as follows [Stic86]:

a) a path through a formula that is a single atomic formula consists of that single atomic formula;

b) any path through one of the disjuncts is a path through the disjunction; and

c) any concatenation of paths through all of the conjuncts is a path through the conjunction.

To facilitate the distinction of disjunction and conjunction, all negations should be drawn inward to atomic formulas. A theorem logically follows from a set of premises if every path passing through the set of premises and the negation of the theorem contains at least one complementary pair of atomic formulas.

Consider the 'mail box' problem again. The input formulas are the following conjuncts,

-manager(x), employee(x);

-employee(x), mail_box(x);

manager(x); and

-mail_box(x).

There are four paths through these formulas:

a) -manager(x), -employee(x), manager(x), -mail_box(x);

b) -(manager(x)), mail_box(x), manager(x), -(mail_box(x));

c) employee(x), -employee(x), manager(x), -mail_box(x); and

d) employee(x), mail_box(x), manager(x), -mail_box(x).

As each path contains a complementary pair of atomic formulas (indicated in the underlined formulas), the conclusion logically follows from the premises.

In the course of proving, there may have paths having common pair of complementary atomic formulas (or connection), like those pairs in clauses (b) and (d) as well as in clauses (a) and (b) in above. The method has an improved version that avoid repetition of connections. This can be depicted in figure 23. The formula is placed in a form of matrix, where formulas being placed in a column is disjuncts, and those in a row is conjuncts. Here, construction of three connections is sufficient.

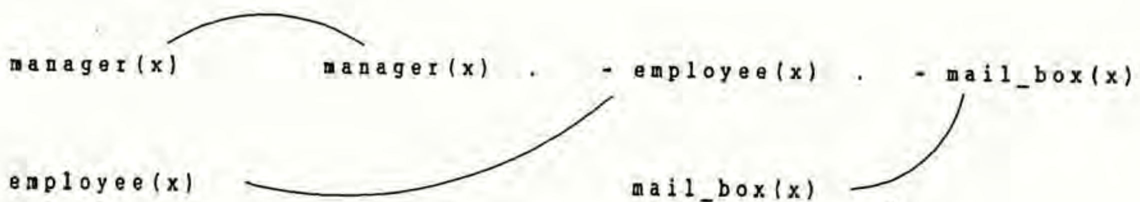

Figure 23 An Example of Connection Method

8.3 Comparison of Automated Theorem Proving Methods

The four methods are effectively the same, namely, they are sound and refutation complete. Thus, the comparison focuses at their efficiency.

8.3.1 Proof Procedure

With regard to the structure of proof procedure, both resolution, natural deduction and tableau proof method are simpler and more elegant than connection method. Their proving strategies can be changed by simply imposing heuristics as rules without modifying the proof procedure itself. For connection method, the path searching strategies are incorporated into the proof procedure, making the modification of the procedure more difficult.

8.3.2 Overhead

Overhead of a proof procedure refers to the amount of effort spent in the preparation steps. Resolution has the highest overhead as it should convert the input formula into conjunctive normal form. For example, a formula like (a & b & c & d) v (e & f & g & h) should be converted into 16 conjuncts, (a v e) & (a v f) & ... & (d v h). As connection method needs to identify the conjuncts and disjuncts in order to find a path, the input formula should be in negation normal form, a less restrictive form than CNF, where argument of negation is atomic

formula only. Thus, every negation should be drawn inward to atomic formulas. Tableau proof method only require negating the theorem to be proved, and does not required any transformation of the formulas, so it takes nearly no overhead at all. Natural deduction make no requirement on the form taken by the input formulas, nor any negation of formula. Thus, it has least overhead.

8.3.3 Unification

During unification, special attention should be paid to the case where existential variables appear within the scope of universal variables. For example, the statement "Everyone has a mother" is expressed as the following formula,

forall x exist y (mother(y, x))

y appears within the scope of x. While the instant that x takes can be arbitrary, the choice of y depends on x. The provers have different treatments to this case.

In resolution, such existential variables are replaced by a skolem function, optionally with argument of the universal variable upon which it depends. The above formula can be expressed as,

forall x (mother(M(x), x))

where M is a skolem function that returns the mother of its argument, x. The construction of skolem function requires extra storage.

In tableau proof method, the problem is handled by instantiating the

existential variable with a new instant that does not appear before the step. Thus,

forall x exist y (mother(y, x))

=> exist y (mother(y, b))

=> mother(a, b)

indicating that a particular instance a is mother of b.

For connection method, the problem is handled by avoiding substitution of dominated variable into dominating one. In the above example, y is dominated by x, thus, it is forbidden to substitute y into x.

8.3.4 Heuristics

Heuristics direct the proof procedure to its goal. Resolution leaves the choose of a promising parent clauses to be resolved totally to heuristics, i.e. the success of the procedure heavily depends on the heuristics. In natural deduction or tableau proof method, the heuristics is more in the sense that it not only has to determines the parent premise but also the rule of inference to be applied at each step. Although there is no proposal of using heuristics in connection method, heuristics can be helpful in choosing a clause from the input matrix and a sub-clause from a clause.

8.3.5 Getting Lost

'Getting lost' is a term given by Wos [88] to describe the situation where a reasoning program does not adequately direct a proof, pursuing unprofitable path one and another. Resolution, natural deduction and tableau proof method generate intermediate clauses and make the pool of parent clauses expanding. On the other hand, connection method is quite systematic in the sense that it exhaustively searches, in a depth-first manner, all paths at any time, does not expand the set of input clauses, and, in certain extend, does not repeat steps that has been taken.

For example, consider this formula {L1 v L4}, {-L1 v L2}, {-L2 v L3}, {-L4 v L5}, {-L1}, {-L5}. Using resolution, in the best case, three steps are needed; while in the worst case, 18 steps are taken. In fact, there are 12 different resolvents can be generated by using 18 resolution steps. Heuristics is heavily depended in order to arrive at shorter proof. Using connection method, the procedure can be terminated by locating 3 connections in best case, and locating 5 connection at the worst case.

8.4 The Choice of Tool for Schema Verification

Contrast to resolution, connection method has the following advantages: 1) require a moderate conversion of input formulas; 2) allow easy manipulation

of existential variable within universal variables; and 3) seems to be more systematic or directive in construction of a proof. Bibel has proved that connection method can simulate the model elimination [Love78], a highly restricted form of linear resolution[7], at the same number of steps and lesser storage space [Bib82a]. For example, consider this formula {-L1 v L3 v L4}, {L1 v L2 v L4}, {-L2 v L1}, {-L4}, {-L3}. Figure 24 shows the six resolvents obtained by resolving among input clauses (by the requirement of linear resolution, the first resolvent must be obtained by resolving two input clauses). The pair of number below each resolvent represents the maximum and minimum number of resolution steps required to obtain an empty clause using that resolvent.

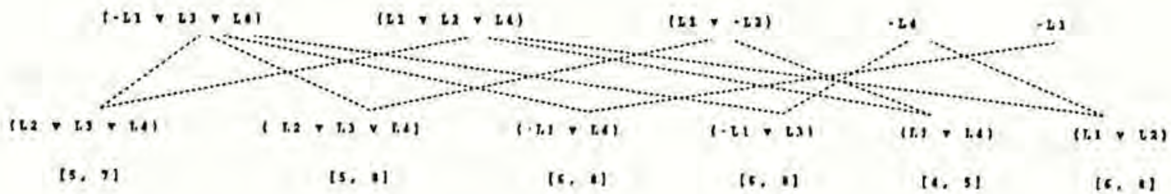

Figure 24 Example on Linear Resolution

A proof using eight steps is shown in figure 25. It can be seen that step (e) and (h) repeat step (c), and step (g) repeats step (b). While using connection method, four or six steps are sufficient to report the complementarity of these

---

7 Linear resolution requires at least one of the parent clauses to each resolution operation must be either an input clause or ancestor clause of the other parent

clauses, as shown in figure 26a and 26b respectively. If redundancy elimination mechanism is used, five connections are needed in the case of figure 26b. This is because the avoidance of repetition has been taken account at the second chance of step 5 of connection method algorithm (see Appendix G).

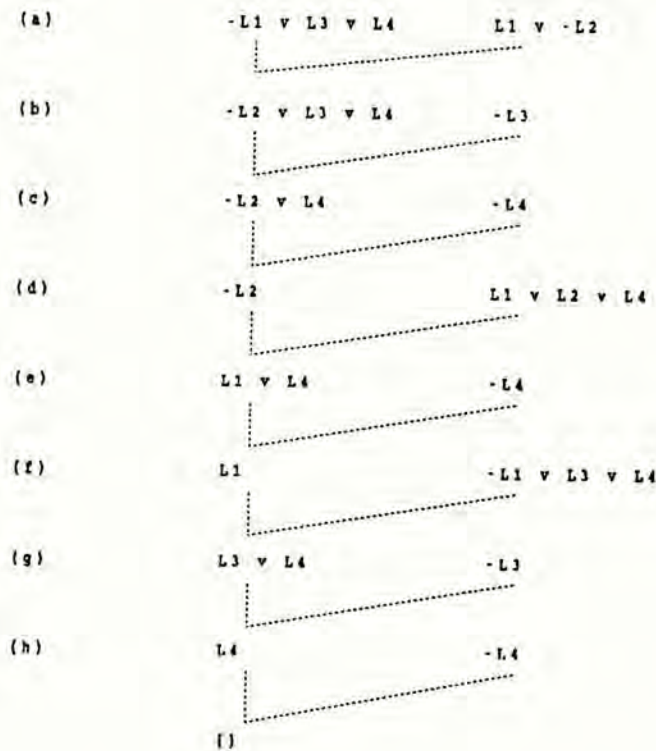

Figure 25 A Proof Using Linear Resolution

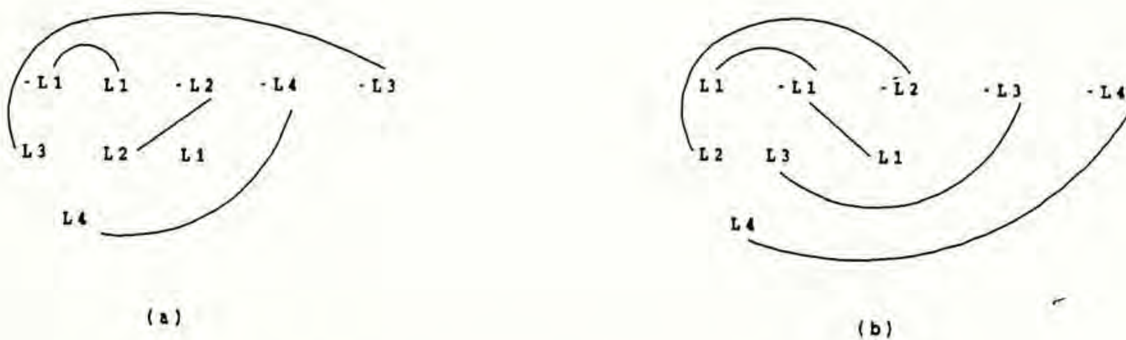

Figure 26 A Proof Using Connection Method

Contrast to natural deduction method or tableau proof method, connection method does not heavily depend on heuristics, and a totally mechanical procedure can be developed for it. It seems that connection method is efficient in handling a wide range of problems. On the other hand, natural deduction, though seems to be less efficient than connection method, will be useful for hard problem where user's involvement is needed to construct the proof.

CHAPTER 9

IMPROVEMENT OF CONNECTION METHOD

9.1 Motivation of Improving to Connection Method

On top of the connection method algorithm, Bibel attempts to build a redundancy elimination mechanism to avoid repetition of previously considered paths. However, the combined version is very d. His approach confuses the control flow of and data structures used by the two purposes. For example, a stack, named "WAIT" in the combined version, has five different functions. Besides being primarily used to store up any path has not yet been examined, the stack is also used to store information for handling redundancy elimination. Stack elements for the former purpose is labeled as "sg", while that for the latter one as "sc", "lm", "dm", and "PRSG". This makes the algorithm hard to read and maintain. More seriously, the resulting algorithm is not guaranteed to be correct, as Bibel comments in [HoBi82] that,

> "It is rather obvious, that the preliminary version of COMPLEMENTARY has the correct I/O-behaviour (able to return 'complementary' if all paths through an input matrix are complementary, and 'non-complementary' otherwise). This is no longer clear, when we add the mechanisms (for redundancy elimination)...".

Another problem is due to the ad hoc approach to handle the redundancy

elimination. Once a kind of redundancies is identified, a definition for its occurrence is formulated, and then the algorithm is modified to cater for this particular type. The process repeats whenever a new type of redundancies is discovered. This progressive approach inevitably lengthens the time of development as well as gives no insight on solving the problem completely. Also, potentially fault may arise due to incompatibility between redundancy definitions. An example is to be discussed later. Bibel also gives the following remarks [HoBi82]:

> "... the addition of another mechanism interferes with them (the previously added mechanisms) thus yielding an inconsistency".

The main problem of this approach is that it lacks a global view on the situation, thus does not able to produce a complete solution. Motivated by these drawbacks of the existing system, we take a general systematic and correct approach to solve the problem. In this chapter, we discuss the result.

9.2 Redundancy Handled by the Original Algorithm

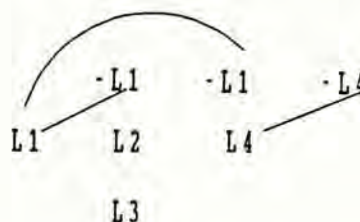

Figure 27 Example on the First type of Redundancies

In [HoBi82], three types of redundancies are identified. The first one is defined as follows:

"... the subgoal must be to the right of the left end of any connection with right end to the right of the subgoal."

The situation can be depicted from an example as shown in figure 27. Showing the complementarity of paths starting with L1 and L3 is the same as that starting with L1 and L2. Thus, the subgoal L3 can be eliminated. This satisfies the above definition because L3, is to the right of the left end (L1 of the first clause) of any connection (the one between L1 and -L4) with right end (-L4 of the third clause) to the right of L3. Bibel [Bib82a, Bib82b, HoBi82] uses a set SC to explicitly record the right coordinates of all elements of ACT which are the left ends of such connections, and reject any subgoal whose coordinate is greater than every element in SC.

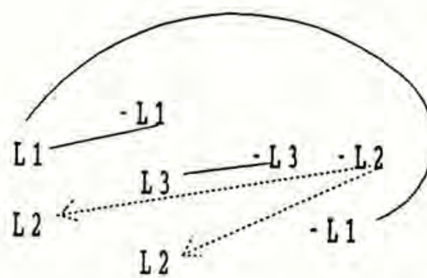

Figure 28 Example on the Second Type of Redundancies

The second kind of redundancies is described as follows:

"... exactly those subgoals are redundant which are to the right of the rightmost left end among all connections with a right end below or to the right of the origin of the dotted arrow."

For example, consider the formula as shown in figure 28. Dotted arrow represents a connection between a suspicious redundant subgoal and its complement. L2 in the second clause is redundant while that in the first one is not. This is because there is only one connection (L1 to -L1 in the third clause) with a right end (-L1) below the origin (-L2 in the third clause) of the dotted arrow (from -L2 to L2 in the first or second clause), and only L2 in the second clause is to the right of the left end (L1) of this mentioned connection. L2 in the first clause is not redundant since it is not to the right, rather below, of that connection. In the original algorithm, PRSG is used to tackle this particular type of redundancies by storing the complement of any potentially redundant subgoal (like -L2 in the above example).

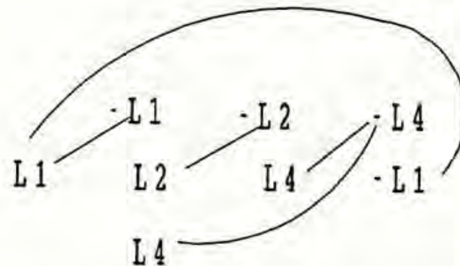

Figure 29 Example on the Third Type of Redundancies

A third kind of redundancies is termed as factorization. The idea can be depicted by figure 29. Just after identifying the connection between L2 and -L2, we can ignore exploring the subgoal L4 in the third clause because there is another L4 in the second clause which has been push into the stack, and thus is expected to be considered later.

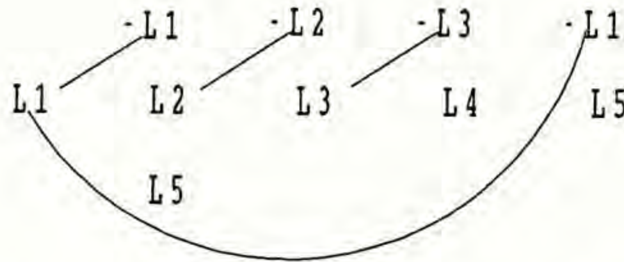

Figure 30 Example on Problem with Factorization

However factorization together with the first type of redundancies elimination mechanism will spoil the correctness of the algorithm. For example, consider figure 30. By factorization, L5 in the fifth clause will be ignored. Nevertheless, the L5 in the second clause is also eliminated due to the first type of redundancies (L5 is to the right of a connection passes through it). Thus, the original algorithm returns complementary for this formula, which in fact is not. Bibel admits no final solution is found for this problem except sophisticated bookkeeping, which further complicates the algorithm. It is interesting to notice that fail in handling factorization makes connection method cannot definitely simulate resolution. The argument is presented in Appendix H. In the coming

section, we will discuss how our proposed approach overcome a part of this problem.

9.3 Design Philosophy of the Improved Version

It is the clumsy definition and ad hoc treatment of redundancy that motivate the author to derive a general, reliable and non-interference approach. By general, we mean we can handle as many redundancies as possible. By reliable, we mean the mechanism do not draw wrong decision on eliminating a redundancy. By non-interference, we mean the mechanism do not affect the normal operation of the connection method algorithm.

To fulfil the first design principle, we start with defining redundancy in a general sense. We interpret redundancy as repeating useful effort that have been done before. In our problem, useful effort refers to steps showing a path is complementarity, i.e. having at least one connection. Thus, our approach is to develop a structure to store up any connection found. Then, by referring to this structure, we can determine whether an attempt is redundant or not. Reliable is guaranteed if we record all and only useful information and properly referring to the structure. Non-interference is fulfilled if the mechanism acts like a consultant device and is invoked on request such that it does not alter the control flow of the connection method algorithm. Also, the mechanism should not share any data structure with the original algorithm.

Our work is done as follows: firstly we remove the redundancy elimination mechanism developed by Bibel from the connection method algorithm. The resulting algorithm, which simply detects complementarity of formulas, is called Primary Connection Method. Then, our redundancy elimination mechanism, which is developed independently, is attached to this simplified version. Basically our mechanism is to construct an AND/OR connection graph to store up the useful effort as mentioned, and develop a procedure to traverse the graph to identify redundant attempt. The mechanism can be further polished to cater for some particular redundancies, without spoiling the design philosophy mentioned.

9.4 Primary Connection Method Algorithm

From the full version of the connection method algorithm, as shown in Appendix G, we extract the necessary portion for checking complementarity of an arbitrary matrix. We called this the primary connection method algorithm, as shown below. Statements in bold face is used to handle redundancies as to be explained later.

```
1:    ACT := NIL;

2:    choose a clause C from M;
      M := M \ C;
      forall literals K s.t. (-K in C) and (K in ACT) do {
          C := C \ -K;
          [call CONNECT(K, -K);]
          }
      if C = NIL then goto 8;

3:    choose a matrix M' from C;
      C := C \ M';
      if C < > NIL then
          call push_wait(C, ACT, M));
      if M' is not a literal then {
          M := M' U M;
          goto 2;
          }
      L := M';
      ACT := ACT U {L};
      [if Chk_Compl(ACT) is complementary then
          if C < > NIL then goto 3 else goto 8;]

4:    if M = NIL then return ("non-complementary");

5:    if there is no clause C in M s.t. -L in C then
          if there is no clause C in M s.t. -K in C for some K in ACT then {
             while WAIT < > NIL do pop(WAIT);
             goto 1;
             }
          else
             choose C from M s.t. -K in C for some K in ACT;
      else
          choose C from M s.t. -L in C;
      M := M \ C;

6:    if -L in C then
          choose a matrix M' in C s.t. -L in M'
      else
          choose a matrix M' in C s.t. -K in M' for some K in ACT;
      C := C \ M';
      forall literals K s.t. (-K in C) and (K in ACT) do {
          C := C \ -K;
          [call CONNECT(K, -K);]
          }
```

```
7:    if M' is a literal then {
          [call CONNECT(M', -M');]
          if C <> NIL then goto 3 else goto 8;
      if C <> NIL then
          call push_wait(C, ACT, M);
      if -L in M' then
          choose a clause C from M' s.t. -L in C;
      else
          choose a clause C from M' s.t. -K in C for some K in ACT;
      M := (M' \ C) U M;
      goto 6;

8:    if WAIT = NIL then return ("complementary");

9:    (C, ACT, M) := value of top of WAIT;
      pop(WAIT);
      goto 3;
```

Remarks: Care should be taken in choosing a M' in step 3. If C at that time have more than one clause, M' should be taken to include all the clauses.

Table 4 Primary Connection Method Algorithm

The primary version contracts to nearly half its original size, in terms of number of statements, which indeed help a lot in grasping the main control flow. For ease of comparison with original algorithm, we remain to use goto in the algorithm. Through out this discussion, we restrict the discussion on checking complementarity of formula of propositional logic. For first order logic, the procedure is similar except unification is required to act upon pairs of complementary atomic formulas as well as those clauses they belong to whenever a connection is found.

Before explaining the algorithm, three terms are defined. A *literal* has truth value of either true or false under an interpretation. A *clause* is any combination of literals, negation, connectives (and, or, imply), and parentheses. A *matrix* is a conjunction of clauses. In this part, a literal is represented as a single capital letter (usually K and L) optionally followed by a number, the or connective as "v", the and connective as "&", and negation as "-".

The primary connection method algorithm mainly consists three parts. The first part, step 2 and 3, aims at choosing a literal (L) from a clause (C) in the matrix (M), whose clauses has not been considered yet. The second part, from step 5 to 7, locates a complementary literal to L. The third part, step 9, restores the system to a state that is described in a subgoal in the stack, and continues to proceed from that state.

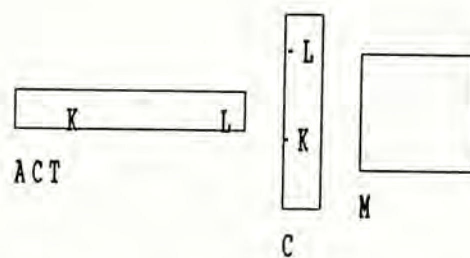

Figure 31 Data Structure in Connection Method Algorithm

Relationship between main data structures used in the algorithm can be depicted as in figure 31. ACT is a list of literals, whose complementarity has been found or is currently under consideration. L is the latest literal chosen in the first part. C is a clause chosen in the second part because it has a literal complementary to L. M is the remaining matrix whose clauses have not yet been visited. At this snapshot, we are going to show that any path having ACT as its leading portion and passing through the clause C is complementarity. Therefore, if there is any literal, -K, in C, such that K occur in ACT, we can discard -K from C for considering complementarity of paths having ACT and through C. This is catered in the forall loop in step 2 as well as that in step 6. In case -L does not appear in clause C, the algorithm takes a second chance that chooses a clause that has a literal complementary to an element in ACT. This is done in the second if statement of step 5. If the second chance fail, it means that the set of literals in ACT is non-complementary. The process starts from beginning, and treats M as if it is the original matrix. A stack WAIT is used to maintained information of other paths to be considered later.

To illustrate how the algorithm works, consider the following input formula, (L1 v L3) & (-L1 v L3) & -L3, it is represented graphically as,

L1	-L1	-L3
L3	L3	

The paths to be considered is as follows:

(a) L1, -L1, -L3;

(b) L1, L3, -L3;

(c) L3, -L1, -L3; and

(d) L3, L3, -L3.

Suppose L1 of {L1, L3} is chosen after step 3. This means that we are going to show complementarity of any path starting with L1, i.e. path (a) and (b). Information about the remaining paths (i.e. path (c) and (d)) should, then, be stored up in somewhere, here the stack WAIT, as a sub-goal to be considered later. In step 5, the clause {-L1, L3} is chosen. At this time, Act is {L1}, C is the chosen clause, M is {-L3}. After establishment of connection between L1 and -L1, the algorithm goes back to step 3, chooses L3 of the second clause as the next literal, and finds a connection with the third clause {-L3}. Now, all paths beginning with L1 have proved to be complementary. Next, we pop out the sub-goal in WAIT, and consider paths having the L3 of the first clause as the first literal, i.e. path (c) and (d). The whole process ends with the discovery of the last connection between L3 of the first clause and -L3.

The soundness and completeness of this primary algorithm immediately follow from that of the original algorithm. This is because this primary form is equivalent to the original one without redundancy elimination capability. The proof of soundness and completeness of the original algorithm is found in [Bib82b].

9.5 AND/OR Connection Graph

The job of redundancy elimination is done with the help of an "AND/OR Connection Graph". Nodes in the graph are called AND-nodes. Each disjunct in the input matrix owns an AND-node. An AND-node represents a literal or a disjunction of literals and/or clauses. A path passes through a disjunction should pass through each of the disjuncts, hence the name "AND" for the node. A slot in the AND-node is allocated for each literal or sub-clause within the disjunct corresponds to the AND-node. In case a slot is assigned a conjunction of literals or clauses, a new AND-node is created for each of the conjuncts, and a clause-edge is created to link between the slot and each of the AND-node of the conjuncts, forming a hierarchical of AND-nodes. A path passes through a conjunction could pass through some of the conjuncts (a path is complementary simply if it has a pair of complementary literals), hence the name "OR". A connection edge is maintained, linking the slots corresponding to the pair of complementary literals, for each connection found, hence the name "connection". The procedure connect in table 4 performs this task.

By proper traversal of the AND/OR connection graph, one can identify any subpath that has already been proved to be complementary, thus able to reject any redundant attempt. An example of the graph is shown in figure 32. The matrix corresponds to the formula L1 & (((-L1 v L2) & -L2) v L3) & -L3. In

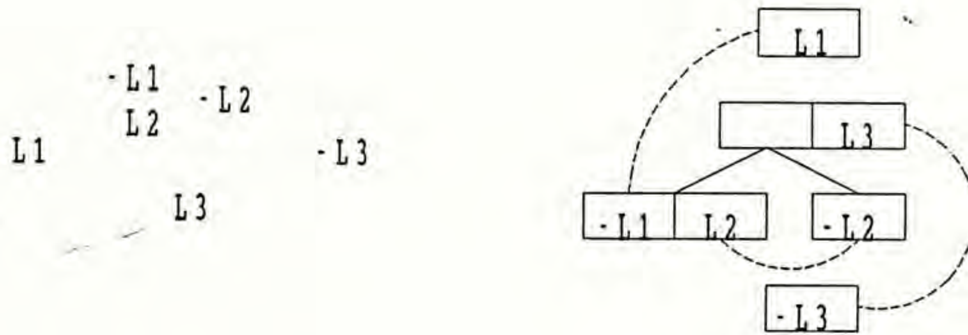

Figure 32 Illustration of AND\OR Connection Graph

graphical representation, we use a rectangle to represent an AND-node, solid line for sub-clause edge, and dotted line for connection edge. The graph is built before the connection method algorithm is invoked. The graph construction algorithm is shown in table 5.

```
AND_OR_Graph();
for each clause C in input matrix do
    create a pointer, Ptr, pointing to an empty AND-node;
    call Build_Graph(C, Ptr);

Build_Graph(C, Ptr);
if C is a literal or a disjunction of literals then
    allocate slot(s) in the AND-node pointed by Ptr;
    assign literal(s) to the slot(s);
else
    if C is a conjunction of clauses then
     allocate a slot, S, in the AND-node pointed by Ptr;
        for each clause C1 of C do
          create a pointer, Ptr1, pointing to an empty AND-node;
          make a clause-edge from S to Ptr1;
          Build_Graph(C1, Ptr1);
    else
        for each disjunct, M, of C do
           Build_Graph(M, Ptr);
```

Table 5 Algorithm to Build AND/OR Connection Graph

9.6 Graph Traversal Procedure

With the AND/OR connection graph, we can determine whether a path beginning with a given list of literals have been proved to be complementary or not. The procedure is called "Chk_Compl", and is shown in table 6. It is invoked at the end of step 3, after having chosen a literal L whose complementary literal is to be located. The procedure returns whether the path being extending from ACT U {L} is complementarity, base on solely previously located connections. If it is so, there is no need to consider this literal, and we can go on with other subgoal. Otherwise, we continue as usual.

```
Function Chk_Compl(LL, C);
for each slot in the AND-node of C do
    if the slot corresponds to a literal, L then {
        if LL U {L} is complementary then
            exit this iteration with "complementary";
        if L only has a connection with a literal L' s.t. L' and some literal,
          L" in LL share the same clause with L' not an immediate parent
          or child of L" in the AND/OR connection graph then
            exit this iteration with "non-complementary";
        if L has connection with some clauses then {
            for each clause C' of these clause do
                if C' is fully connected then
                    if Chk_Compl(LL U {L},C') return complementary
                        then exit this iteration with "complementary";
            if no complementary is reported from the immediate for loop
                then
                exit this iteration with "non-complementary";
            }
    for each sub-clause C' pointed by the slot do
        if C' is fully connected then
            if Chk_Compl(LL, C') is complementary then
                exit this iteration with "complementary";
    if no complementary is reported from the immediate for loop then
        exit this iteration with "non-complementary";
    }   /* end of outer for loop */
if each of iteration in the main loop reports complementary then
    return "complementary";
else
    return "non-complementary";
```

Table 6 Procedure for Checking Complementarity

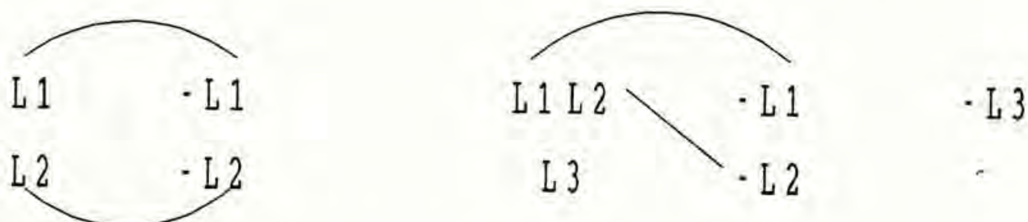

Figure 33 Example on Re-Visit Clause in Graph Traversal

In traversing the graph, care should be taken in making connection to a clause which has already been visited. For example, in figure 33, the formula (L1 v L2) & (-L1 v -L2) is not complementary. After making the connection between L1 and -L1, we should not allow connecting L2 and -L2, otherwise, we will get into an infinite loop. However, it is not the case that we can never visit a clause of the input matrix twice. Consider another example for the formula ((L1 & L2) v L3) & (-L1 v -L2) & -L3 which is complementarity. After connecting L1 and -L1, we visit the first clause again by connecting -L2 and L2. The confusion arise because we have apparently treated two clauses as one. In fact, in the second example, we should consider ((L1 & L2) v L3) as two clauses, namely (L1 v L3) and (L2 v L3). Thus, if we refer a clause as a disjunction of literals, then the rule of avoiding repeat visiting a clause hold. In terms of the AND/OR connection graph, a clause can be re-visited if it consists those literals that are collected by traversing AND-nodes of a clause in input matrix through clause edges in depth-first manner without backtracking, equivalent to that described in the third if statement of the outer for loop in table 6.

To speed up the procedure of checking complementarity, we define a term called "fully connected" and restrict the check to involve only fully connected clauses. A clause is fully connected if every path passing through the disjuncts of the clause has at least one connection. The restriction is justified as only these clauses are fruitful to the checking procedure. This can be easily implemented by

maintaining a flag for each clause in the original matrix, and set the flag once the clause is fully connected.

9.7 Eliminating Redundancy using AND/OR Connection Graph

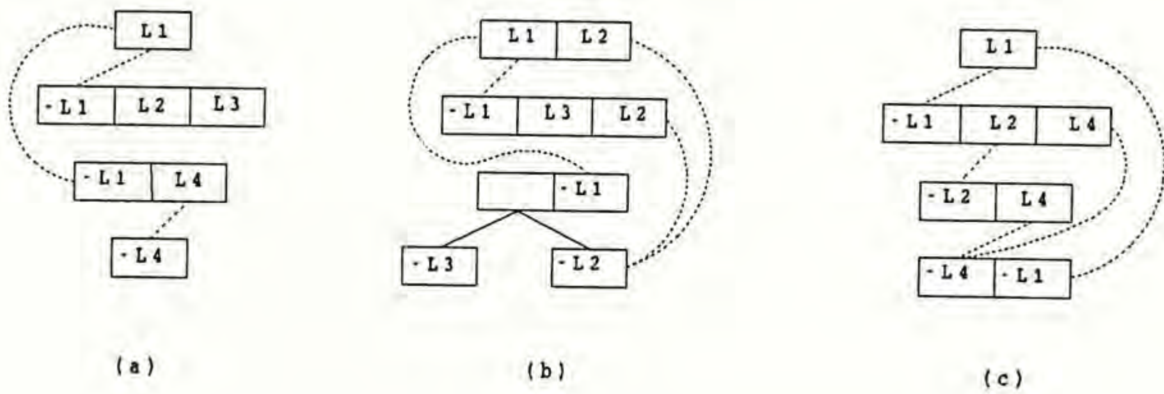

Figure 34 Examples on Eliminate Redundancies Using AND/OR Connection Graph

Using AND/OR connection graph, we can not only eliminate redundancies that can be handled by the original method, namely the first two types of redundancies as mentioned, though less efficient, but also tackle case that the original one cannot, namely factorization. The AND/OR connection graph corresponds to the formulas in figure 27, 28, and 29 are shown in figure 34a, 34b, and 34c respectively. The technique of using AND/OR connection graph to eliminate redundancies for these three cases is the same: avoid repeating those steps that have been taken before. In figure 34a, after the subgoal of checking paths extending from {L1, L3} is pop and the third clause is chosen, Chk_Compl

can determine that the paths is complementary and thus stop further executing the connection searching algorithm. In figure 34b, after connecting L3 and -L3, we conclude complementarity is reached since L1 and -L1 has already been connected. Similar operation for the case as shown in figure 34c.

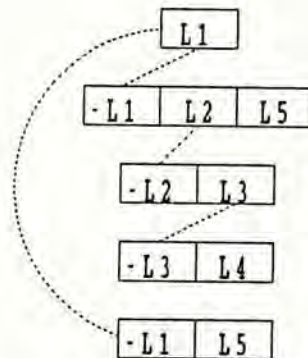

Figure 35 Example of AND/OR Connect Graph for Factorization

Referring back to the counterexample for factorization in figure 30, the corresponding AND/OR graph is shown in figure 35. Using our approach, we can successfully return non-complementarity, since no attempt is made to delete any subgoal before knowing that it leads to complementarity. If encountering the L5 in the fifth clause first, we simply return non-complementarity as there is no -L5 in the formula and all clauses has considered. If encountering the L5 in the second clause first, we will not eliminate it since there is no complementary clause found for the remaining formula.

9.8 Further Improvement on Graph Traversal

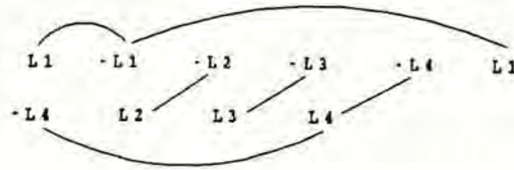

Figure 36 Example on Improvement Using Graph Traversal

One of the drawbacks of the original redundancy elimination mechanism is that it cannot completely made use of result obtained in previous search. It only reduces subgoal that is sure to be complementary, but ignores any other possibility that can reduce redundancy. For example, consider the case as shown in figure 36. When subgoal with -L4 of the first clause is pop, the original connection method algorithm will choose the fourth, third, and second clause in turn, not noticing that, in fact, this sub-paths have already been considered previously. Though the sequence of the proof is different, they are essentially the same. In our approach, we handle this situation by reference to the graph and make use of those sub-path proved to be complementary. When we encounter -L4, we know that it has complement literal in the fourth clause, we then focus on the remain literal, -L3. This leads to the consideration of the third clause, and then the -L2. The process repeat until we cannot further avoid repeating steps. Then, we will arrive at the -L1 of the second clause. Then, this literal is push into WAIT and continue as usual, but already have saved three looping through the

main algorithm.

This strategy will not repeatedly lead to useless deadend. At the first place, repeating steps in different round of run of the algorithm need not be useless. Syntactically speaking, no single path is more promising than the other in looking for complementarity. In fact, the original algorithm does not put any restriction on the order of clauses to be examined. Also, steps that are useless for showing one sub-path is complementary does not imply the same result to the others. Thus, there is occasion where repeating previous step is necessary. Secondly, repeating steps will not lead to deadend. Since we repeat steps for different sub-paths under consideration (i.e. different content in ACT), we will not get into an infinite loop. In sum, we just skip some proved paths, and indeed does not affect the effectiveness of the original algorithm.

9.9 Comparison with Original Connection Method Algorithm

Using the AND/OR connection graph, we can handle different kinds of redundancies using the same algorithm without affecting the effectiveness of the complementarity checking algorithm. While, in the old approach, a correctness proof should be derived in case a new mechanism is introduced to the system, which is time-consuming, and may has problem of incompatibility between the new and old mechanisms. More important, our method can handle some redundancies that the old one cannot. Yet, there is still room for further

improvement of the graph traversal procedure. For example, reduce the search space through useful book-keeping.

On the other hand, the old method is quite efficient in the sense that it can reduce redundancies with lesser effort. But the cost is to take an ad hoc, or hard-wired, approach which seems has less hope in reducing any more redundancy without affecting the effectiveness of the connection method algorithm. In a general proof, there is no reason of occurrence of only a particular types of redundancies, thus efficient in handling only certain type of redundancies is not sufficient in all applications. We think that a general approach, like ours, is more promising to reach a complete solution to the problem.

9.10 Application of Connection Method to Schema Verification

Schema verification aims at determining if there are two or more constraints making contradicting restriction on the occurrences of concepts such that at least occurrence of one concept is always empty. The process of schema verification can be divided into three steps.

9.10.1 Express Constraint in Well Formed Formula

Constraint is expressed in well formed formula (wff) which is built from atomic formula using connectives (and, or, not, imply), quantifiers (for all, there

exist) and parentheses. Atomic formula is a constant, variable, predicate or function. A n-place predicate

$$R(arg_1, arg_2, ..., arg_n)$$

has a specific interpretation. The predicate symbol, "R", refers to a base table in the underlying relational DBMS, and "arg_i" is an attribute of that table. This predicate represents occurrences of the concept corresponds to "R". This representation is justified as constraint is imposed on occurrences of concepts which can be expressed in terms of base tables. User defined constraints should be input in wffs.

Structures of constraints are of various forms. Constraint relates to an association type has a particular structure. For example, generalization association requires constraints between the concepts being generalized fall into one of the four types: set subset, set equal, set exclusive and set intersection. At the same time, user can impose constraints according to the application domain. Therefore, we need a common notation through which to express constraints and then examine the interaction among them. First order logic is chosen as the tool because of its elegance and ability in expressing all practical constraints. The way of formulating a constraints has been widely discussed. In SAM*, constraints can be specified for the following associations: generalization (set related constraints), interaction (the mapping constraints), aggregation (inter-occurrences constraints), and membership. The following is some example on expressing constraints in first order logic.

A is-a B :

A(x) -> B(x)

A = B union C :

(A(x) -> B(x) v C(x)) &

(B(x) -> A(x)) &

(C(x) -> A(x))

A = B intersect C :

(B(x) & C(x) -> A(x)) &

(A(x) -> B(x) & C(x))

the first argument is a primary key of A :

A(x, z) & A(y, z1) -> x = y

Table 7 Examples on Expressing Constraints

Thus, instead of expressing every constraint in well form formula, user can use symbols (like is-a, union, intersect, and etc.) to represent the underlying restrictions.

Refer back to the example in figure 20, assume the concept "tourism", "hotel", "heavy_industry" and "1st_class_hotel" own base tables. Then, the constraint "occurrence of hotel sector is set exclusive to that of the industrial sector" can be expressed as,

forall S# (hotel_sector(S#) -> not (industrial_sector))

However, as hotel sector and industrial sector do not own a base table, it is converted into the following form,

forall S# (hotel(S#) or 1st_class_hotel(S#) ->

not (heavy_industry(S#) or tourism(S#) or hotel(S#)))

9.10.2 Convert Formula into Negation Normal Form

All the wffs are converted into negation normal forms, i.e. negation is at atomic formula. Thus, the above constraint is expressed as,

forall S# ((-hotel(S#) and -1st_class_hotel(S#)) or (-tourism(S#) and -1st_class_hotel(S#) and -heavy_industry(S#))

or in matrix form,

-hotel(S#)	-1st_class_hotel(S#)	
-tourism(S#)	-1st_class_hotel(S#)	-heavy_industry(S#)

with only two rows of atomic formulas.

9.10.3 Verification

Lastly, the connection method algorithm is invoked. Inconsistent is reported under one of two situations:

a) every path has a complementary pair of atomic formulas;

b) when the set of input formulas is not complementary, and there is an atomic formula such that every path without a complementary pair of

literals has a negation of that atomic formula.

The first situation implies there is a constraint does not logically follow from the other constraints. The second situation implies that there is no occurrence for the concept corresponds to the atomic formula. The algorithm needs modification to cater for the second situation. Whenever non-complementarity is reported in step 4, we store up the ACT and pass control to step 9 which then goes on as usual. At the end of the process, if the set of formulas is found to be non-complementary, we compare each copy of ACT collected to see if there is a negation atomic formula occurs in every copy. If so, we should report it as an error of the schema. There is no need to introduce skolem function as mentioned in section 8.3.3. When inconsistency exist, we can trace the proof to see which constraints constitute the contradiction, or the single negation atomic formula that implies a concept has no occurrence, thus help to correct the schema.

PART IV

CHAPTER 10

FURTHER DEVELOPMENT

In this chapter, we discuss the directions along which this system can be improved.

10.1 Intelligent Front-End

The data model can be treated as a form of knowledge representation, similar to the role taken by semantic network. Our system allows room for incorporating with an intelligent front-end that can take an active role in answering question. The front-end can not only display information explicitly asked by the user, but also able to guide the user to explore other information interested to him by using the acyclic directed graph. For example, suppose a user asks about the interest rate of a fund, but the database does not has this piece of information yet. Instead of simply saying no to the user, the system can try to suggest source of relevant information by displaying the sibling of the concept "interest rate", say through the generalization association grouping these concepts.

10.2 On Connection Method

The algorithm for the current graph traversal on the AND/OR connection graph is still needed to be improved for efficient purpose. The AND\OR connection graph can be constructed implicitly by adding pointers to the structure storing the input formulas. Additional book-keeping may help in identifying paths that are complementary. Another related topics that has not been discussed is equality reasoning. It deals with inference using equality assertion, e.g. f(a) and a=b should imply f(b). This implication can not be drawn solely by the mentioned proof procedure, rather additional axiom is needed, like the approach of paramodulation [Stic86].

10.3 Many-Sorted Calculus

The proving procedure can produce a shorter proof by incorporating with a many-sorted calculus. The idea runs as follows. A sort is a class of objects. The universe of discourse (the set of interested objects) consists a set of sort symbols which are partially ordered by subsort order relation, forming a sort hierarchy. For example, animal, man and bird are sorts. Man is a subsort of animal, and so does bird. Variables and function symbols are also associated with sorts. Arguments of functions and predicates have domainsorts defining the sorts to which the arguments can belong. A formula is well-sorted if each variable and function has a sort, and argument of functions and predicates belongs to or is a

subsort of the corresponding domainsort. For example, the function area(polygon) returns area of triangle, square, and etc. But it does not accept line as argument. In the proving procedure, each step must only involve well-sorted formulas, using a restricted inference rules. With this extension, the search space for a proof will be narrowed, since it rejects meaningless attempt, say, substitute two terms with unrelated sorts.

There is reported studies on many-sorted calculus with resolution [Walt87]. The idea can well be applied to connection method. For example, given the following axioms

Figure 37 An Example on Sort Hierarchy

a) japanese companies and korean companies are companies;
b) there are some of each of them;
c) there are some companies deal with US Government;
d) all companies like to deal with companies that deal with US Government;
e) japanese companies does not like to deal with korean companies;
f) there is a company does not deal with US Government.

We can express these sentences into well-sort formulas.

a) { D(company1, company2), D(company2, us) }

b) { -D(japanese, korean) }

c) { -D(company3, us) }

where D(A, B) is interpreted as party A likes to deal with party B. We can connect D(company1, company2) and -D(japanese, korean) by substituting japanese into company1 and korean into company2, but not company1 into japanese and company2 into korean. Also, we cannot connect D(company2, us) and -D(japanese, korean), since there is no sort relation between companies and US Government. Without many-sort calculus, four connections must be considered. However, with the calculus, only two connection is sufficient to report complementarity of these formulas.

It happens that the acyclic directed graph constructed in the implementation of a DBMS using SAM* provides the necessary structure, the sort hierarchy, for using many-sort calculus. Thus, it is a promising direction to develop a many-sorted connection method for verifying schema using SAM* as the data model.

CHAPTER 11

CONCLUSION

In this thesis, we have presented the concept of semantic data model, in particular the SAM*. A prototype of a DBMS using SAM* as a data model has been developed. Implementation issues include: design of a DDL for the schema, map the schema into a logical organization for an underlying DBMS, and develop two query manipulation languages to access the DBMS. The results of implementation reveal the power of a DBMS using semantic data model in two aspects:

a) user need not to know about the details of the logical organization of the database system in formulating a query to the database; and
b) user can inquire the schema such that he can know what information is stored in the database.

The benefits as pointed out in (a) can be obtained provided that the following assumptions are valid:

1. the user and the system can share a common perception on the real world objects and concepts;
2. the data model is completely developed;

3. the user does not simply perform simple retrieval operation;
4. effectiveness is the primary concern.

Besides the advantages of increased expressiveness, there is also gain from the data independent characteristic of this approach. User is expected to be able to access the database easily even though the logical structure is modified or another database for another application domain (not only stock investment) is plugged in, so long as the above assumptions are still valid.

The use of semantic data model gives rise to a problem called schema verification. Schema verification is a process that checks the existence of any contradicting constraint being specified. Constraint is expressed in first order logic. Proving consistency (or satisfiability) of a set of first order logic assertions has long be the task of automated theorem proving methods. Thus, these methods are chosen as the candidate for the solution. Four provers are investigated. Their pros and cons are discussed. Connection method is suggested to take on the job of schema verification because of its,

a) more systematic in finding a proof than other methods, it dose not easy to get lost;
b) seems to be not worse than resolution;
c) less overhead than resolution;
d) less storage usage required; and
e) available of a totally mechanical proof procedure.

However, the ad hoc approach taken by the connection method in handling redundancies elimination makes the method has less hope in further speeding up its efficiency. Thus, we develop another general and reliable approach which works with an AND/OR connection graph. The results show that our approach can handle cases that the original can, though less efficient, as well as cases that the original method cannot.

At last, the author would like to point out some advantages of using automated theorem proving [Wos 88],

a) the proof of an inconsistent "theorem" can tell which hypotheses is wrong, thus, in schema verification, we can know which constraint cause the inconsistent;

b) user added strategies allow user involves in the proof; and

c) the result can be trusted, as the provers are sound, they will not draw wrong conclusion.

Though no automatic theorem prover is complete, it is not totally useless, as remarked by Wos [82],

> The point which is becoming increasingly clear is that automated theorem-proving programs can be treated as colleagues. With such a program, one can, for example, make conjectures, test conjectures, and find holes in proposed methods of proof. Each of these activities is reminiscent of that which a colleague performs.

Thus, it is still promising to explore the use of automated theorem prover to handle the schema verification.

APPENDIX A Comparison of Semantic Data Models

	Unstructured Object Representation	Standard Abstraction	Network versus Hierarchy	Relationship
E-R	Limited	Aggr.	Strong Network	User Select
TAXIS	Limited	Gene. Aggr. Clas.	Strong Hierarchy	Pre-defined
SDM	Limited	Gene. Aggr. Clas. Asso.	General Hierarchy present	User-defined
DAPLEX	Limited	Asso. Aggr. Clas.	No direct support for neither	User-defined
SAM*	Enhanced (special-purpose set built in)	Gene. Aggr. Clas. Asso.	Network	Pre-defined

Remarks: "Aggr." stands for aggregation, "Gene." for generalization, "clas." for classification, and "Asso." for association.

APPENDIX B Construction of Occurrences

We define occurrence of a concept v, denoted as occu(v), in a recursive manner as follows:

```
Function occu(v)
If v is at sufficient specific level then
    occu = {u | u belongs to dom(v)}
Else
    let v1, v2, ..., vn be the child vertices of v in the schema
    let a = asso(v)
    occu = stru(a, v1, v2, ..., vn)
Endif
```

Depending on the association concern, the function stru takes different forms as follows:

```
Function stru(a, v1, v2, ..., vn)
Case a of
    M: stru = {t | t = name(vi) for 1 <= i <= n}
    A: stru = {occu(v1) x occu(v2) x ... x occu(vn)}
    G: stru = {unique-occu(v1) U unique-occu(v2) U ... U unique-occu(vn)}
    I: stru = {t1 x t2 x ... x tn |
        For i:= 1 to n
        Case asso(vi) of
        A: ti = unique-occu(vi)
        I: ti = occu(vi)
        G: let u1, u2, ..., un be the child vertices of vi, and asso(ui) = A
              or I for 1 <= i <= n
              ti = {unique-occu(u1) U unique-occu(u2) U ... U
                  unique-occu(un)}
        End Case
    C: stru = {{t1, t2, ..., tn} | ti belongs to power set of occu(vi)}
    X: stru = {occu(v1) x occu(v2) x ... x occu(vn)}
    S: stru = {occu(v1) x (occu(v2) x ... x occu(vn))}
End Case
```

where "x": a cross-product operation;

"U": a set union operation;

"unique-occu(v)": the occurrences of the set of a number of (up to the whole set) child vertices of v such that this set of child vertices can uniquely identify an occurrence of v;

"asso(v)": the association type attached to the vertex v;

"dom(v)": domain for the vertex which may be integer, characters and etc.

APPENDIX C Syntax of DDL for the System.

Schema ::= { Statement }*

Statement ::= { AggregationStatement
| GeneralizationStatement
| InteractionStatement
| CompositionStatement
| CrossproductStatement
| SummarizationStatement }

AggregationStatement ::=
aggregation ParentName [**root**] [**base**]
{ FieldName [**key**] }+
{ ChildName [**key**] }+
under ConstraintStatement

GeneralizationStatement ::=
generalization ParentName [**root**]
ChildName { ChildName }+
with { **si-si** | **se-se** | **sx-sx** | **st-ss** | **ss-ts** }#
under ConstraintStatement

InteractionStatement ::=
interactive ParentName [**root**] [**base**]
FieldName { ChildName }+
ChildName { ChildName }+
with { **1-1** | **1-N** | **N-1** | **N-M** }#
under ConstraintStatement

CompositionStatement ::=
composition ParentName [**root**]
ChildName { ChildName }+
under ConstraintStatement

CrossproductStatement ::=
crossproduct ParentName [**root**]
ChildName { ChildName }+
under ConstraintStatement

SummarizationStatement ::=
  summarization ParentName [**root**]
  IdentifyName { ChildName }+
  under ConstraintStatement

ParentName ::= NameType

FieldName ::= NameType

ChildName ::= NameType

IdentifyName ::= NameType

NameType ::= letter { letter | digit | _ }*

Notation:

1. word in bold face is terminal symbol; name beginning with a capital letter is non-terminal symbol.

2. **root** following a parent name indicates that the parent itself has no parent concept;

3. **base** following a parent name indicates that the occurrence of parent form a relational base table in implementation;

4. **key** following a child name indicates that the child is treated as a component of primary key in every member of occurrence of the parent;

5. **multiple** following a child name indicates that the child can be attached with more than one value in a member of occurrence of parent;

6. ParentName and ChildName are names of parent concept and child concept respectively; IdentifyName is a name of the concept whose information is summarized; FieldName is the attribute name of table being used in the underlying DBMS.

7. [...]: item in the bracket is optional;

8. {...}: choose one of the items, separated by "|", in the bracket;

9. $\{...\}^{+}$: repeat the items, separated by "|", in the bracket one or more than one time;

10. $\{...\}^{*}$: repeat the item in the bracket for any number of times (include zero)

11. $\{...\}^{\#}$: choose items, separated by "|", from the bracket for ${}_nC_2$ times where n is the number of child concepts; they represent the relationship between the following pair of child concepts: 1st&2nd, 1st&3rd, ..., 1st&last, 2nd&3rd, 2nd&4th, ..., 2nd&last, ..., last-but-one&last.

APPENDIX D Syntax of Semantic Manipulation Language

The notations used in this appendix is the same as those mentioned in Appendix C.

query ::= { PointerListType
    | find_node_query
    | node_info_query
    | find_path_query }

find_node_query ::=
    VariableName := Find_Node(restriction
        { Logical_Operator restriction }*)

Find_Node ::= **find_node** | **fn**

Logical_Operator ::= **and** | **or**

restriction ::= { **name** = NameType
    | **asso** = AssociationType
    | **parent** = ParentList
    | **child** = ChildList
    | **key** = KeyList }

node_info_query ::= **?** Projection (PointerListType)

Projection ::= { **name** | **asso** | **parent** | **child** }

find_path_query ::= **find_path**(PointerType PointerType)

VariableName ::= { letter | digit | _ }+

NameType ::= letter { letter | digit | _ }*

AssociationType ::= { **membership** | **aggregation** | **generalization**
    | **interaction** | **composition** | **cross-product** | **summary** }

ParentList ::= PointerListType

ChildList ::= PointerListType

KeyList ::= PointerListType

PointerListType is a variable which is a pointer pointing to a list of vertices (can be one) in the graph. PointerType is a variable name which is a pointer pointing to one vertex in the graph.

APPENDIX E Testing Schema for Fund Investment DBMS

```
begin
generalization :: fund_database , ,
  fund_equities , fund_currency , fund_warrant ;
  with sx-sx , sx-sx , sx-sx ;
  under constraint ;

generalization :: fund_equities , ,
  asia , non_asia , equities_intl , equities_others ;
  with sx-sx , sx-sx , sx-sx , sx-sx , sx-sx , sx-sx ;
  under constraint ;

generalization :: asia , ,
  japan , hk , korea , se_asia ;
  with sx-sx , sx-sx , sx-sx , sx-sx , sx-sx , sx-sx ;
  under constraint ;

generalization :: non_asia , ,
  us , uk , europe , australia ;
  with sx-sx , sx-sx , sx-sx , sx-sx , sx-sx , sx-sx ;
  under constraint ;

generalization :: equities_intl , ,
  intl_equities , intl_managed ;
  with sx-sx ;
  under constraint ;

generalization :: fund_currency , ,
  us , japan , uk , europe ;
  with sx-sx , sx-sx , sx-sx , sx-sx , sx-sx , sx-sx ;
  under constraint ;

generalization :: fund_warrant , ,
  japan , hk , far_east , warrant_intl , others_warrant ;
  with sx-sx , sx-sx , sx-sx , sx-sx , sx-sx , sx-sx ,
       sx-sx , , sx-sx , sx-sx , sx-sx ;
  under constraint ;

aggregation :: japan , , base ,
  [ fund_code key , fund_name , date_formed , fund_assets ,
    trustees , securities_no , initial_charge ,
    annual_charge , unit_accounts , reports , dealing ,
```

```
  minimum_investment ] ,
 fund , , : fund_name , , : date_formed , , :
 fund_assets , , : trustees , , : securities_no , , :
 initial_charge , , : annual_charge , , :
 unit_accounts , , : reports , , : dealing , , :
 minimum_investment , , : fund_port_dist , , :
 fund_performance , , ;
 under constraint ;

aggregation :: hk , , base ,
 /* same as japan */

aggregation :: us , , base ,
 /* same as japan */

aggregation :: uk , , base ,
 /* same as japan */

aggregation :: australia , , base ,
 /* same as japan */

aggregation :: korea , , base ,
 /* same as japan */

aggregation :: se_asia , , base ,
 /* same as japan */

aggregation :: europe , , base ,
 /* same as japan */

aggregation :: intl_equities , , base ,
 /* same as japan */

aggregation :: intl_managed , , base ,
 /* same as japan */

aggregation :: far_east , , base ,
 /* same as japan */

aggregation :: warrant_intl , , base ,
 /* same as japan */

aggregation :: others_warrant , , base ,
 /* same as japan */
```

```
aggregation :: fund_basic_info , , base ,
  /* same as japan */

aggregation :: fund_port_dist , , base ,
  [ fund_code key , port_component , port_percentage ] ,
  fund_code , key , : port_component , , :
  port_percentage , , ;
  under constraint ;

aggregation :: fund_performance , , base ,
  [ fund_code key , date , return ] ,
  fund_code , key , : date , , : return , , ;
  under constraint ;
end .
```

APPENDIX F Testing Schema for Stock Investment DBMS

```
begin
composition :: stock_info , root ,
  stock_basic_info , contact , directors , auditors ,
  bankers , business_activities , subsidiaries ,
  ordinary_shares , preference_shares , share_price ,
  turnover , dividends , capital_commitments ,
  contingent_liabilities , earnings ,
  ordinary_shareholders , preference_shareholders ;
  under constraint ;

aggregation :: stock_basic_info , , base ,
  [ stock_code key , stock_name , secretary , registrars ,
  main_business_area ] ,
  stock_code , key , : stock_name , , : secretary , , :
  registrars , , : main_business_area , , ;
  under constraint ;

aggregation :: contact , , base ,
  [ stock_code key , registered_office , tel , telex ,
  cable ] ,
  stock_code , key , : registered_office , , : tel , , :
  telex , , : cable , , ;
  under constraint ;

interaction :: directors , , base ,
  [ stock_code key , director , post ] ,
  stock_code , director , post ;
  with 1-N , 1-N , N-M ;
  under constraint ;

interaction :: bankers , , base ,
  [ stock_code key , banker ] ,
  stock_code , banker ;
  with 1-N ;
  under constraint ;

interaction :: auditors , , base ,
  [ stock_code key , auditor ] ,
  stock_code , auditor ;
  with 1-N ;
  under constraint ;
```

```
interaction :: subsidiaries , , base ,
  [ stock_code key , subsidiary , percentage ] ,
  stock_code , subsidiary , percentage ;
  with 1-N , 1-N , N-M ;
  under constraint ;

aggregation :: business_activities , , base ,
  [ stock_code key , activities ] ,
  stock_code , key , : activities , , ;
  under constraint ;

aggregation :: ordinary_shares , , base ,
  [ stock_code key , authorized , issued , outstanding ,
  par_value ] ,
  stock_code , key , : authori , , : issued , , :
  outstanding , , : par_value , , ;
  under constraint ;

aggregation :: preference_shares , , base ,
  [ stock_code key , authori , issued , outstanding ,
  par_value ] ,
  stock_code , key , : authori , , : issued , , :
  outstanding , , : par_value , , ;
  under constraint ;

aggregation :: share_price , , base ,
  [ stock_code key , year , high , low ] ,
  stock_code , key , : year , , : high , , : low , , ;
  under constraint ;

aggregation :: turnover , , base ,
  [ stock_code key , year , shares , value ] ,
  stock_code , key , : year , , : shares , , : value , , ;
  under constraint ;

aggregation :: dividends , , base ,
  [ stock_code key , year , dividend , %par_value ,
  per_share , paid_int_fin , earnings_per_share ] ,
  stock_code , key , : year , , : dividend , , :
  %par_value , , : per_share , , : paid_int_fin , , :
  earnings_per_share , , ;
  under constraint ;

aggregation :: capital_commitments , , base ,
  [ stock_code key , year , capital_commitment ] ,
```

```
stock_code , key , : year , , : capital_commitment , , ;
under constraint ;

aggregation :: contingent_liabilities , , base ,
 [ stock_code key , year , contingent_liability ] ,
 stock_code , key , : year , , : contingent_liability , , ;
 under constraint ;

aggregation :: earnings , , base ,
 [ stock_code key , year , capital_employed ,
 pretax_bef_depr , pretax_aft_depr , net_profit ,
 earnings_per_share , dividends_per_share ] ,
 stock_code , key , : year , , : capital_employed , , :
 pretax_bef_depr , , : pretax_aft_depr , , :
 net_profit , , : earnings_per_share , , :
 dividends_per_share , , ;
 under constraint ;

aggregation :: ordinary_shareholders , , base ,
 [ stock_code key , ordinary_shareholder , share# ,
 tot_percentage ] ,
 stock_code , key , : ordinary_shareholder , , :
 share# , , : tot_percentage , , ;
 under constraint ;

aggregation :: preference_shareholders , , base ,
 [ stock_code key , preference_shareholder , share# ,
 tot_percentage ] ,
 stock_code , key , : preference_shareholder , , :
 share# , , : tot_percentage , , ;
 under constraint ;
end .
```

APPENDIX G CONNECTION METHOD

We adopt the convention that all variables get value NIL in the beginning, except M which gets as value the matrix, the complementarity of which is to be checked. An element of WAIT has the form B = (L I V), where L is the label of B, I the index of B, and V the value of B.

```
0:   push(WAIT, ("lm", 0, undef);

1:   ACT := NIL;
     I := 0;
     SC := NIL;
     PRSG := NIL;
     M" := NIL;

2:   choose a clause C from M;
     M := M \ C;
     forall literals K s.t. -K in C and (K,j) in ACT for some j do {
          C := C \ -K;
          choose j s.t. (K, j) in ACT;
          push(WAIT, ("sc", I+1, {j}));
          }
     if C = NIL then goto 8;

3:   choose a matrix M' from C;
     C := C \ M';
     if there exists (l,M0) in PRSG s.t. l <= I and M' U M0 is complementary
     then
          if C <> NIL then goto 3 else goto 8;
     if C <> NIL then push(WAIT, ("sc", I, (C, ACT, NCPATH, M)));
     if SC <> NIL then {
          SC := SC \ {j: j > I};
          push(WAIT, ("sc", I+1, SC));
          SC := NIL;
          }
     if PRSG <> NIL then {
          PRSG := PRSG \ {(l, M0) : l > I};
          push(WAIT, ("prsg", I+1, PRSG));
```

```
            PRSG := NIL;
            }
        if M' is not a literal then {
            push(WAIT, ("lm", I, undef));
            M := M' U M;
            goto 2;
            }
        I := I + 1;
        L := M';
        ACT := ACT U {L};
        NCPATH := NCPATH U {(L, I)};

4:      if M = NIL then return ("non-complementary");

5:      if there is no clause C in M s.t. -L in C then
            if there is no clause C in M s.t. -K in C for some (K,j) in ACT then {
                while the label of top of WAIT is not "lm" do
                    pop(WAIT);
                goto 1;
                }
            else {
                choose C from M s.t. -K in C for some (K,j) in ACT;
                push(WAIT, ("dm", I, undef));
                }
        else
            choose C from M s.t. -L in C;
        M := M \ C;

6:      if -L in C then
            choose a matrix M' in C s.t. -L in M'
        else
            choose a matrix M' in C s.t. -K in M' for some (K,j) in ACT;
        C := C \ M';
        forall literals K s.t. (-K in C or -K = M') and (K,j) in ACT do {
            C := C \ -K;
            choose j s.t. (K, j) in ACT;
            push(WAIT, ("sc", I, {j}));
            }

7:      if M' is a literal then {
            if M" < > NIL then {
                push(WAIT, ("prsg", I, {(0,M")}));
                M" := NIL;
                }
            if C < > NIL then goto 3 else goto 8;
```

```
         }
     if C <> NIL then {
          push(WAIT, ("sg", I, (C, ACT, NCPATH, M)));
          push(WAIT, ("lm", I, undef));
          }
     if -L in M' then
          choose a clause C from M' s.t. -L in C;
     else
          choose a clause C from M' s.t. -K in C for some (K,j) in ACT;
     M" := M' \ C;
     M := M" U M;
     goto 6;

8:   if WAIT = NIL then return ("complementary");

9:   if the label of top of WAIT is "sg" then
          if I = index of top of WAIT then {
               (C, ACT, NCPATH, M) := value of top of WAIT;
               pop(WAIT);
               if SC <> NIL then {
                    SC := SC \ {j: j>I};
                    push(WAIT, ("sc", I, SC));
                    SC := NIL;
                    }
               if PRSG <> NIL then {
                    PRSG := PRSG \ {(l, M): l>I};
                    push(WAIT, ("prsg", I, PRSG));
                    PRSG := NIL;
                    }
               goto 3;
               }
     else {
          (C, ACT, NCPATH, M) := value of top of WAIT;
          I := index of top of WAIT;
          goto 3;
          }

10:  if the label of top of WAIT is "sc" then {
          SC := SC U value of top of WAIT;
          I := index of top of WAIT;
          pop(WAIT);
          goto 8;
          }

11:  if the label of top of WAIT is "dm" then {
```

```
        SC := SC \ {j: j>index of top of WAIT};
        while index of top of WAIT <> "lm" do
            pop(WAIT);
        goto 8;
        }

12:  if the label of top of WAIT is "lm" then {
        I := index of top of WAIT;
        pop(WAIT);
        goto 8;
        }

13:  if the label of top of WAIT is "prsg" then {
        SC := SC \ {j: j>index of top of WAIT};
        m := max(SC U {0});
        for each (l, M_0) in value of top of WAIT do
            if m>l then
                substitute m for l in (l, M_0);
        PRSG := PRSG U value of top of WAIT;
        I := index of top of WAIT;
        pop(WAIT);
        goto 8;
        }
```

APPENDIX H COMPARISON BETWEEN RESOLUTION AND CONNECTION METHOD

In this appendix, a comparison is made between resolution and connection method so as to illustrate the correspondence between the two methods. For simplicity reason, we assume the input formulas are converted in conjunctive normal form before the proof procedure is carried out, though connection method does not require this conversion. With this assumption, the connection method does not loop between step 2 and 3 as well as step 6 and 7. In the comparison, a step in resolution is an application a rule of inference (i.e. modus ponens), and a step in connection method is successfully connecting a pair of complementary literals. We claim that resolution with any refinement strategies can be simulated by connection method using no less steps for propositional logic without occurrence of common literals.

The proof of this argument consists two parts. Firstly, we prove that the number of connections for a set of input formulas that is satisfiable is equal to number of steps taken in resolution. Secondly, we show that connection method can terminate with that number of connections.

The first part is proved by mathematical induction. Suppose we have obtained the shortest proof for the input formulas using resolution. Without loss

of generality, we assume the proof takes n steps, for some positive integer n. The basis of the induction is that there is one connection within the two parent clauses chosen for the nth step (the last step) of the resolution. As parents in the last resolution step must be a pair of complementary literals, it is obvious that there is only one connection. Let the formula that is acted upon at the (n-k)th step of the resolution is M, where k ≥ 1. Assume there are (n-k) connections in M. Consider the proof at the (n-k+1)th steps. This step must resolve a pair of complementary literals. Since no common literal occur, these pair must be new to M. Thus, the number of connections at the formula which is acted upon at the (n-k+1)th step is (n-k+1). By mathematical induction we conclude that there are exactly n connections in an optimal proof using resolution having n steps, in particular.

Next, we should show that the connection method using these n steps can show the input formulas is complementary. This is immediately follows from the soundness and completeness of connection method. As the input formula is complementary, we can determine it using the n connections. Thus, we have shown that connection method takes no more steps than resolution in case no duplicate literal occur.

In case duplication exist, connection method may require more than steps than that of resolution. Only for simple case can factorization be used to eliminate the redundancy due to common literals. For example, consider the

formula: {-L2}, {-L1, L2}, {L1, L2}, here L2 appears in the second and third clause. Resolution needs two steps, while connection method must consider the three connections in them (due to the duplicated connection [-L2, L2]). This is because during a resolution, the resolvent can combine duplicated literals, while in connection method, there is no corresponding action. Without factorization, connection method cannot terminate after considering the connection: [-L2, L2] and [-L1, L1].

Next, we will show that connection method is no worse than linear resolution at the worst case for propositional logic in case no duplication of literals exist. Linear resolution requires at least one of the parent clauses to each resolution operation must be either an input clause or ancestor clause of the other parent [Stic86]. This refinement is complete.

The argument is as follows: in the set of input formulas, suppose there are totally n pairs of complementary literals. Then, using connection method, at most n connections are needed. We claim that the number of the potential pairs of parent clauses used for resolution is greater than or equal to n. As linear resolution allows one of the parents from the input clauses, all pairs of clauses having complementary literals in the input clauses belong to the mentioned potential pairs, and the number is exactly n. In addition, the ancestor clause of the other parent is allowed as the parent clause, thus the number of potential pairs will greater than or equal to n. At the worst case in resolution, the procedure

have to exhaustively try all the pairs before arriving the empty clause; while the connection method can terminate at most n steps. Thus, the claim follows.

REFERENCES

[Abit87] Abiteboul S., and Hull R. (1987). IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems* 12, 4, 525-565.

[Andr81] Andrews P.B. (1981). Theorem proving via general matings. *Journal of the ACM*, Vol. 28, No. 2, (April), pp. 193-214.

[BaLe88] Batory D. S., Leung T. Y., and Wise T. E. (1988). Implementation Concepts for an Extensible Data Model and Data Language. *ACM Transactions on Database Systems*, 13, 3, 231-262.

[Bibe81] Bibel W. (1981). On Matrices with Connections. *Journal of ACM*, Vol. 28, No. 4, pp. 633-645.

[Bib82a] Bibel W. A (1982). Comparative Study of Several Proof Procedures. *Artificial Intelligence*, 18, pp. 269-293.

[Bib82b] Bibel W. (1982). *Automated Theorem Proving*. Vieweg.

[Bibe83] Bibel W. (1983). Matings in Matrices. *Comm. of ACM*, Vol. 26, No. 11, pp. 844-852.

[Bibe86] Bibel W. (1986). Methods of Automated Reasoning. In: *Fundamentals of Artificial Intelligence*, Lecture Notes in Computer Science, Springer-Verlag, pp. 171-217.

[BlHe85] Bledsoe W.W. and Henschen L.J. (1985). What Is Automated Theorem Proving? In: An Overview of Automated Reasoning and Related Fields. *Journal of Automated Reasoning*, Vol. 1, pp. 23-28.

[Date86] Date C. J. (1986). *An Introduction to Database Systems*. Addison-Wesley, fourth edition.

[Fros86] Frost R. (1986). *Introduction to Knowledge Base System*. McGraw Hill.

[GaMi84] Gallaire, H., Minker J., and Nicolas J.M. (1984). Logic and Database: A Deductive Approach. *ACM Computing Surveys*, Vol. 16, No. 2, June, pp. 153-185.

[GeNi87] Genesereth R.M. and Nilsson N.J. (1987). *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Publishers.

[HaMc81] Hammer M., and McLeod D. (1981). Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems* 6, 3 (1981), 351-386.

[Hon89a] The Hong Kong Unit Trust Association. (1989). *The Hong Kong Unit Trust Yearbook 1989*. Longman.

[Hon89b] The Hong Kong Unit Trust Association. (1989). *Investment Performance Measurement.*

[HoBi82) Hornig K.M. & Bibel W. (1982). Improvements of a Tautology-Testing Algorithm. In: *The 6th Conference on Automated Deduction*. pp. 326-341.

[HuKi87] Hull R. and King R. (1987). Semantic Database Modelling: Survey, Applications, and Research Issues. *ACM Computing Surveys* 19, 3, 201-260.

[JaGu88] Jagannathan D., Guck R. L., Fritchman B. L., Thompson J. P., and Tolbert D. M. (1988). SIM: A Database System Based on the Semantic Data Model. In *Proceedings of the ACM SIGMOD Conference (Chicago, 1988)*. ACM, New York, pp. 46-55.

[Jeff89] Jeffrey R. (1989). *Formal Logic: Its Scope and Limits*. McGraw Hill, Second Edition.

[Kent79] Kent W. (1979). Limitations of Record-Based Information Models. *ACM Transactions on Database Systems*, Vol. 4, No. 1, March, pp. 107-131.

[Lans88] Lans R. F. *Introduction to SQL*. Addison-Wesley.

[Ling87] Ling T.W. (1987). *Integrity Constraint Checking in Deductive Databases using the Prolog not-Predicate*. Internal Report, The National University of Singapore.

[Love78] Loveland D.W. (1978). *Automated Theorem Proving: A Logical Basis*. North-Holland.

[LyVi87] Lyngbaek P., and Vianu V. (1887). Mapping a Semantic Database Model to the Relational Model. In *Proceedings of the ACM SIGMOD Conference (San Francisco, 1987)*. ACM, New York, pp. 132-142.

[MiFe86] Miller D. and Felty A. (1986). An Integration of Resolution and Natural Deduction Theorem Proving. In: The Fifth National

Conference on Artificial Intelligence, pp. 198-202.

[MyBe80] Mylopoulos J., Bernstein P. A., and Wong H. K. T. (1980). A Language Facility for Designing Database-Intensive Applications. *ACM Transactions on Database System* 5, 2 (1980), 185-207.

[Nico82] Nicolas J.M. (1982). Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, Vol. 18, No. 7, pp. 227-253.

[NiCh87] Nixon B., Chung L., Lauzon D., Borgida A., Mylopoulos J., and Stanley M. (1987). Implementation of a Compiler for a Semantic Data Model: Experiences with Taxis. In *Proceedings of the ACM SIGMOD Conference (San Francisco, 1987)*. ACM, New York, pp. 118-131.

[NiCh89] Nixon B. A., Chung K. L., Lauzon D., Borgida A., Mylopoulos J., and Stanley M. (1989). Design of a Compiler for a Semantic Data Model. In: *Foundations of Knowledge Base Management*. Springer-Verlag.

[OpSu88] Oppacher F. and Suen E. (1988). HARP: A Tableau-Based Theorem Prover. *Journal of Automated Reasoning*, 4, pp. 69-100.

[PeMa88] Peckham J., and Maryanski F. (1988). Semantic Data Models. *ACM Computing Surveys*, Vol. 20, No. 3, pp. 153-189.

[Rich83] Rich E. (1983). *Artificial Intelligence*. McGraw-Hill.

[Robi66] Robinson, J.A. (1966). A Machine-Oriented Logic Based on the Resolution Principle. *Journal of ACM*, Vol. 12, No. 1, January, pp. 23-41.

[Robi79] Robinson, J.A. (1979). *Logic: Form and Function*. Edinburgh University Press.

[Ship81] Shipman D. W. (1981). The Functional Data Model and the Data Language DAPLEX. *ACM Transaction on Database System*, Vol. 6, No. 1 , pp. 140-173.

[SmSm77] Smith J. M., and Smith D. C. P. (1977). Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, Vol. 2, No. 3, pp. 105-133.

[Stic86] Stickel, M.E. (1986). An Introduction to Automated Deduction. In: *Fundamentals of Artificial Intelligence*, Lecture Notes in Computer Science, Springer-Verlag, pp. 75-132.

[Su83] Su S. Y. W. (1983). SAM*: A Semantic Association Model for Corporate and Scientific-Statistical Databases. *Information Sciences* 29, 151-199.

[Su86] Su S. Y. W. (1986). Modelling Integrated Manufacturing Data with SAM*. *IEEE Computer* Jan. pp. 34-49.

[Tsur84] Tsurt S. (1984). An Implementation of GEM - supporting a semantic data model on a relational back-end. In *Proceedings of the ACM SIGMOD Conference (Boston, 1984)*. ACM, New York, pp. 286-295.

[Walt87] Walther C. (1987). *A Many-Sorted Calculus Based on Resolution and Paramodulation*. Morgan Kaufmann Publishers.

[Wos82] Wos L. (1982). Solving Open Questions with an Automated Theorem-Proving Program. In: *6th Conference of Automated Deduction*, Lecture Notes in Computer Science, Springer-Verlag, pp. 1-21.

[Wos88] Wos L. (1988). *Automated Reasoning: 33 Basic Research Problems*. Prentice Hall.

CUHK Libraries
000316046