Design of SQUIREL

by

Clifford Francis Chew

B.Sc., Simon Fraser University, 1985

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Clifford Francis Chew 1988
SIMON FRASER UNIVERSITY
May 1988

All rights reserved. This thesis may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

Approval

Name: Clifford Francis Chew

Degree: Master of Science

Title of Thesis: Design of SQUIREL

Dr. Binay Bhattacharya
Chairman

Dr. Wo-Shun Luk
Senior Supervisor

Dr. James P. Delgrande
Supervisor

Dr. JiaWei Han
External Examiner

April 5, 1988
Date Approved

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

DESIGN OF SQUIREL

Author: ______________________

(signature)

CLIFFORD FRANCIS CHEW

(name)

5th May 88

(date)

Abstract

Current relational database languages do not provide general users with many reasoning facilities. The data stored in the database has to be fully interpreted by the user before it is used. This applies to both insertion and retrieval of data.

In this thesis we have designed SQUIREL (Structured Query UserInterfaced Relational Extended Language) as an enhanced database language which provides the user with features like inheritance, classification and default value assignments. A hierarchical structure is introduced to incorporate these features. The primary bases of SQUIREL are the database language SQL and the extended relational model RM/T.

SQUIREL has been designed to be compatible with current SQL constructs and is also a language specification for RM/T. We have also made a comparison between SQUIREL and two prominent AI systems.

Quotation

"Many are the plans in a man's heart, but it is the Lord's purpose that prevails.[1]

[1]From the Book of Proverbs, chapter 19, verse 21.

Acknowledgements

I am extremely grateful to my supervisor Dr. Wo-Shun Luk for his invaluable guidance, support and encouragement for the duration of this research. Without his patience and keen insight, I would not have been able to complete this thesis.

I would like to thank Dr. James Delgrande for being on my committee, his teaching and his helpful and insightful comments which served to improve this work considerably. I would also like to thank Dr. Jiawei Han for taking the time to be my external examiner and Dr. Binay Bhattacharya for serving on my thesis examination committee.

My discussions with fellow students Cathy Levinson, Pat Pattabhiramin, Sanjeev Mahajan, Steven Yap and Wang Xiao were very helpful in clarifying parts of the thesis. Finally, I would like to thank my beloved friend, Joyce, for her constant support and encouragement over the past two and a half years. I dedicate this work to her.

Table of Contents

Approval ii
Abstract iii
Quotation iv
Acknowledgements v
Table of Contents vi
List of Figures ix
1. INTRODUCTION 1
- 1.1. Motivation 1
- 1.2. The Relational Model 3
- 1.3. Database Research 6
- 1.4. INTRODUCTION TO RM/T 8
  - 1.4.1. E-Relation 9
  - 1.4.2. P-relation 9
  - 1.4.3. Characteristic Graph relation 10
  - 1.4.4. Associative Graph Relation 11
  - 1.4.5. Generalization 13
  - 1.4.6. Rules 13
  - 1.4.7. Operators 14
  - 1.4.8. Other features of RM/T 14
- 1.5. Objective 15
- 1.6. Thesis Organization 15

2. FEATURES OF SQUIREL 16
- 2.1. Traditional Database's Hierarchical Concept 16
- 2.2. SQUIREL's Hierarchical Concept 17
  - 2.2.1. Objective of Hierarchies 18
- 2.3. Inheritance 19
- 2.4. Classification 19
  - 2.4.1. PlayBack 20
- 2.5. Default Assignments in SQUIREL 21
- 2.6. Management of Views 22
- 2.7. Procedure Invocation 22
- 2.8. SQUIREL's hierarchy 23
  - 2.8.1. Description of Hierarchy 23
  - 2.8.2. Categories and Partitions 23
  - 2.8.3. Properties of the Hierarchy 24

3. SQUIREL CONSTRUCTS AND THEIR USE 25
- 3.1. Base entity types/v-entity types 27
- 3.2. Hierarchy Constructs 28
  - 3.2.1. CREATION 28

3.2.2. INSERTION 29
3.2.3. UPDATES 29
3.2.4. DELETION 30
3.2.5. DROP 30
3.2.6. RETRIEVAL 30
3.2.7. INSERTION 31
3.2.8. DELETION 31
3.2.9. UPDATES 32
3.2.10. RETRIEVAL 32
3.3. Procedure Invocation 33
3.3.1. PlayBack procedure 35
3.4. Use of Default Assignments 35
3.5. Management of views/v-entity types in SQUIREL 37
3.5.1. Views not applicable to SQUIREL's Hierarchy 39
3.6. Use of Constructs in the Hierarchy 41
4. IMPLEMENTATION PLAN FOR SQUIREL 46
4.1. Hierarchy Relation 46
4.1.1. System Hierarchy Table--SYSHIER 48
4.1.2. Category/Partition in Hierarchies 49
4.1.3. Procedure Table 50
4.2. Classification 51
4.2.1. Type Level Classification 51
4.2.2. Instance Level Classification 54
4.3. Inheritance 56
4.4. Default Entity Types 57
4.5. Procedure Construction 58
5. COMPARISON WITH KEE AND CENTAUR 59
5.1. The KEE System 59
5.2. Description of KEE Frames 59
5.3. Capabilities 60
5.3.1. Inheritance 61
5.3.2. Cardinality/Value Class Reasoning 63
5.3.3. Object Oriented System 64
5.3.4. KEE Rules 64
5.4. Use and Concepts of Rulesystem2 65
5.4.1. Classification in KEE 66
5.4.2. Control Strategy 66
5.5. SQUIREL's Answer 67
5.5.1. Classification 68
5.5.2. SQUIREL Procedure Management 70
5.6. COMPARISON WITH CENTAUR 70
5.7. Classification in SQUIREL 75
6. CONCLUSION 79
6.1. Future Research 80
Appendix A. USE OF SQUIREL CONSTRUCTS 82
A.1. Variables 82
A.2. CREATION 82
A.3. INSERTION 83
A.4. UPDATES 84

A.5. DELETION 84
A.6. DROP 85
A.7. RETRIEVAL 85
A.8. INSERTION 86
A.9. DELETION 86
A.10. UPDATES 87
A.11. RETRIEVAL 87
A.12. SETARG 88
A.13. Algorithm for coalescing E-relations 88
Appendix B. ALGORITHMS FOR SQUIREL CONSTRUCTS 90
B.1. TYPE Level Classifier 90
B.2. INSTANCE Level Classifier 91
B.3. CREATE algorithm 92
B.4. DROP algorithm 92
B.5. INSERT algorithm <TYPE> 93
B.6. INSERT algorithm <INSTANCE> 93
B.7. DELETE algorithm <TYPE> 94
B.8. DELETE algorithm <INSTANCE> 94
B.9. RETRIEVAL algorithm <TYPE and INSTANCE> 95
B.10. UPDATE algorithm <TYPE> 95
B.11. UPDATE algorithm <INSTANCE> 96
B.12. TRIGGER algorithm 96
B.13. Pseudo Code for the PLAYBACK rule 97
Appendix C. Pseudo code for the OAD rule 99
References 101

List of Figures

Figure 1-1:	STUDENT Entity Relation	9
Figure 1-2:	STUDENT_INFO	10
Figure 1-3:	Property graph relation	10
Figure 1-4:	STUDENT_SCHOOL Characteristic relation	10
Figure 1-5:	Characteristic Graph Relation	11
Figure 1-6:	Associative and Kernel Entity types	11
Figure 1-7:	Associative Graph Relation	12
Figure 1-8:	An Employee Database	13
Figure 2-1:	A Department hierarchy	17
Figure 3-1:	Personnel UGI hierarchy	25
Figure 3-2:	Payment.Hierarchy	38
Figure 3-3:	Illegal Student Hierarchy	40
Figure 3-4:	Employee.Hierarchy	42
Figure 3-5:	Jobtypes Returned	42
Figure 3-6:	Jobtypes Returned	43
Figure 3-7:	Occupations Returned	43
Figure 3-8:	Occupations Returned	44
Figure 3-9:	Promotion.Hierarchy	44
Figure 4-1:	Personnel hierarchy relation	47
Figure 4-2:	Indexed file for hierarchy relations	47
Figure 4-3:	SYSHIER relation	49
Figure 4-4:	Implementation of PARTITION attribute	49
Figure 4-5:	Temp.Hierarchy	52
Figure 4-6:	Temp.Hierarchy	52
Figure 4-7:	Temp.Hierarchy	53
Figure 4-8:	Temp.Hierarchy	53
Figure 4-9:	Temp.Hierarchy	53
Figure 4-10:	EX hierarchy	55
Figure 4-11:	An Example of a Hierarchy	56
Figure 4-12:	P-relation of STUDENT entity type	57
Figure 5-1:	STUDENT Frame	61
Figure 5-2:	GRAD.STUDENT Frame	62
Figure 5-3:	SFU.GRAD.STUDENT Frame	62
Figure 5-4:	STUDENT1 Frame	63
Figure 5-5:	STUDENT TYPE RULES	65
Figure 5-6:	GRADUATE.HIERARCHY	69
Figure 5-7:	Prototype Network in CENTAUR	73
Figure 5-8:	Air.Disease.Hierarchy	75
Figure 5-9:	Classification Path for Air Disease Hierarchy	78
Figure 6-1:	Computing Science Graduate Students	81

Chapter 1
INTRODUCTION

1.1. Motivation

The relational model for databases [CODD 70] was presented as a tool that allowed users to ignore physical storage representation details, hence providing them with a good conceptual aid. Since 1970, this model has evolved to become one of the most widely used database models for commercial database applications. Many database management systems, including INGRES [STON 76], QBE [ZLOFF 77], DB2 [IBM DB2] and SQL/DS [IBM SQL/DS] have adopted it as their data model.

The relational model provides a level of abstraction above the physical implementation details, so a user is able to create, access, insert and delete data from the database without having to know the actual implementation or storage details of the relations involved. Not only does this reduce the number of details that a user must be concerned with, but it also ensures that the user interface remains consistent over time. For example, suppose the user wants to represent the following information in their database:

```
A Person  has a name, sex and age.

A Student has a name, stud# and dept.
```

The above information can be encoded in a relational form as follows:

```
PERSON(name,sex,age)
STUDENT(name,stud#,dept)
```

This encoding makes no reference to the physical organization of the data, so that a physical reorganization, for example, changing from B-trees to a hash index does **not** change the users' perception of the relations. He/She would still visualize the relations as shown above and would continue to manipulate them using the same operators. Thus the relational model can help focus the user's attention on conceptual relations that exist between the data.

SQL (Structured Query Language) is a query language for relational databases. It is used both to express the relational model as well as to query it. SQL uses an English-like syntax which allows the user to express his/her request for data in a clear and logical manner.

Some examples of SQL are:

```
CREATE TABLE PERSON(name CHAR(9),SEX CHAR(6),AGE NUMERIC);

INSERT INTO PERSON VALUES('John', 'Male', '23');

SELECT name, sex , age
 FROM  PERSON;
```

The first SQL statement defines a PERSON relation to have three attributes, name, sex and age. The second SQL statement inserts a value into the PERSON relation and the third queries the database about the values in the PERSON relation. As can be seen, SQL is both easy to understand and use. In addition, it has been proposed as the ANSI industry standard for relational database query languages. All of this helps to explain why it is fast becoming the most widely used query language for relational DBMS (database management systems).

The relational model has been embraced by the database community but it still has some shortcomings as is shown in section 1.2. Therefore, we propose to add several features to the relational model that will enhance its reasoning capabilities. These features are described as follows: the **inheritance** of attributes between related conceptual relations, e.g., a student is a person and should also have all the attributes of a person. Secondly, the **classification** of conceptual relations to represent the connection(s) that exist between the conceptual relations, e.g., a graduate student is a student who is in turn a person. We also add a default value assignment relation that enables default values to be assigned to attribute values of conceptual relations and a form of object orientation[2] where a procedure defined in a high level programming language can be attached and invoked by a conceptual relation.

One extended relational model, RM/T (Relational Model Tasmania) [CODD 79] has already

[2]Both information about an object and the procedures appropriate to an object are grouped together into a data structure [GERV 87].

proposed a form of inheritance called generalization which is similar to what we want to provide but RM/T has no classification. RM/T also does not have a query language and to allow access to the features we have added, we present SQUIREL (Structured Query User Interfaced Relational Extended Language). SQUIREL is an extension of SQL and is proposed as a language specification for RM/T.

In the rest of this chapter we present a description of the relational model, some research on adding semantic information to databases, a more detailed description of the RM/T model and finally, an overview of the layout of this thesis.

1.2. The Relational Model

Relations are **sets** of tuples each having the same attributes and can be represented as R(A,B,C) where A, B and C all represent attributes. The attributes themselves are taken from a collection of domains D1, D2,..., Dn (n > 0) that contain sets of values of similar type, e.g., all possible student numbers for a given institution or all possible courses offered in a given semester.

The *cartesian product* denoted X {Di: i=1,2,..., n} is the set of all n-tuples <t1, t2,..., tn> such that ti is an element of Di for all i <= n. The n-tuples in X represent all combinations of attributes taken from the n domains, where t1 is taken from D1, t2 is taken from D2 and so on. A relation R is of degree n (has n attributes) and is defined on these n domains if it is a subset of this cartesian product.

A relational database is a time varying collection of tabular relations of assorted degree, all of which can be accessed and updated. *Base relations* are relations that cannot be completely derived from other relations whereas *derived relations* are relations which can be completely derived from base relations. An important class of derived relations is the *view*. Views are 'windows' to a database, in the sense that the view represents an important subset of data derived from one or more base relations. They are virtual relations that do not physically exist but are derived from the base relations when a view is referenced in a query. An example of a view definition in SQL is:

```
CREATE VIEW REAL_STUDENT(name,sex,age,stud#,dept)
 AS     SELECT name,sex,age,stud#,dept
 FROM   PERSON P, STUDENT S
 WHERE  P.name = S.name;
```

This view has the attributes: name, sex, age, stud# and dept taken from the base relations person and student. Once defined, a view can be queried as if it is a relation that physically exists in the database. All the operations that can be applied to relations can also be applied to views. Views have proven to be an important aspect of relational database systems and are advantageous in that they:

1. allow the same data to be seen by different users in different ways (at the same time).
2. provide automatic security on data that exists in the relations but is not included in the view.
3. provide some logical data independence[3].

In what follows, we will use the terms view and v-entity type interchangeably.

Each relation has a set of *candidate keys* associated with it, which is a collection of attributes that uniquely identify a tuple in R. One candidate key is selected as the *primary key* for each base relation in the database.

All insertions, updates, and deletions of base relations are constrained by two rules.

1. Entity Integrity: The primary key value of a base relation cannot have a null component.
2. Referential Integrity: Suppose we have a compound (multi-attribute) primary key, K, of a relation R. If A is an attribute of K then for all values v of A in K, v must occur as a value of a simple primary key in some base relation.

Relations are sets, allowing the use of relational algebraic set operators like UNION, INTERSECTION and SET DIFFERENCE to be applied to relations in databases. In fact there are no structural links such as pointers between relations and all associations between relations are represented solely by values. Thus, the relational model consists of:

---

[3]This means that we are able to change some aspects of the base relations without having to redefine the view.

1. A collection of time-varying tabular relations.
2. Insert-update-delete rules.
3. A set of relational algebraic set operators.

The reader is directed to [ULL 82] and [DATE 86] for a more thorough explanation of the relational model.

One shortcoming of relational databases is that they fail to capture some aspects of meaning which are attached to the data in a relation. For example, given the PERSON and STUDENT relations above, two straight-forward questions which might be asked are:

1. If I have a STUDENT, is that STUDENT a PERSON?
2. If I have a PERSON who has a student number, is that PERSON a STUDENT?

Current relational databases are unable to answer the above questions even though the answers are quite apparent to the user. The user has to fully interpret any given data and associate meaningful concepts to it. We find that even though the relation names are PERSON and STUDENT, the database is unable to associate the data to these concepts independently and the user has to assign data to specific relations using his/her own ideas of who or what constitutes a STUDENT or a PERSON. This activity of associating meaning to data has been called *semantic data modeling* [CADIOU 76] and a great number of database researchers have been looking into this 'problem'.

This does not imply that relational databases provides no semantic information. Domains, the names of keys and functional dependencies (non key attributes in a relation **depend** on the key attribute/s) can all help to incorporate this type of information into relational databases. Still database researchers have long seen the need to enhance database capabilities and much research has been done to extend the relational model to contain additional semantic knowledge, while still preserving the independence of implementation.

1.3. Database Research

Researchers in DBMS (database management systems) are concerned with handling large databases **efficiently**. Although they desire to equip DBMSs with reasoning power, the development of these facilities cannot be made at the expense of performance. Current databases deal with voluminous data and are mostly disk-based. The data is needed continuously and must be accurate. Therefore robustness and efficiency are of primary concern.

Some of the main research contributions to semantic data modeling are summarized below.

1. DATA ABSTRACTION [SMITH 77] is a process where certain details are deliberately omitted, helping users to focus their attention on the relevant details. Smith proposes two kinds of abstraction *generalization* and *aggregation*. Generalization is an abstraction which turns a class of objects into a generic object, e.g., TRUCKERS, ENGINEERS, SECRETARIES can be generalized under the heading EMPLOYEES. Aggregation is an abstraction which turns a relationship between objects into an aggregate object, e.g., aggregating the attributes (Name, Room, Date) to form the relation RESERVATION. Both of these abstractions are based on Codd's relational schema [CODD 70]. Smith also introduces a new data type *generic*, which is used to represent an abstraction of a class of objects (in most applications this would be a relation). A generic type does not represent individuals but instead represents a class of objects like TRUCKER. Using the generic types, Smith introduces the concept of a generic hierarchy. The hierarchy when combined with aggregates allows a rich variety of models to be defined.

   Using these concepts, databases are able to provide information more concisely. The concept of generalization, that is, where a trucker, doctor and engineer may be generalized to be an employee and employees are looked upon as an aggregation of name, sex and age (the common attributes of trucker, doctor and engineer) allow databases to be more precisely defined.

2. A proposal for incorporating *ADT* (Abstract Data Types) and their *rules* as part of the

relational database model is discussed in [STON 85]. The author considers the use of ADTs for individual columns/attributes of a relation. The relational model is augmented with a system relation called ADT which stores all of the abstract data types. Every entity in the ADT defines its own functions, procedures and type. Possible extensions to this concept include allowing ADTs to inherit the properties of other ADTs and also allowing ADTs to have multiple arguments. For example, an ADT might be TIME which could have values like '1300 - 1400' and might have procedures to move it ahead by a specific amount of time. TIME could also allow multiple arguments like 'dinner, 1500 - 1600' which are needed to determine the validity of an update. For example, the TIME ADT could use the arguments 'dinner, 1500 - 1600' to check on an update which specifies that dinner is at 1800. This update would then be rejected.

A rule system, RAISIN (Rules from AI specified for INGRES [STON 85]) is proposed to combine the separate existing rule systems for integrity control, protection and views into a single special purpose programming language. Its basic structure is a sequence of ON-THEN clauses where the action clause is triggered whenever the condition clause is true. An example is shown below:

```
ON <function> TO <relation>
UPDATE <relation2>;
```

Here the condition clause is ON <function> TO <relation> and the action clause is the command UPDATE <relation2>.

3. Logical query languages for databases [ULL 85]: Queries on databases are expressed in a subset of first order logic, Horn clauses. The clauses and predicates are transformed to a rule/goal graph representation. Queries on the database, which may be recursive, are evaluated by a *capture rule* technique. Ullman defines backward chaining and forward chaining[4] through these rules. In addition, a sideways capture rule that allows results to be passed from one goal to another is defined.

[4]Backward chaining refers to a top down, goal directed approach where a goal is broken up into smaller subgoals to be solved, which in turn are broken up into smaller subgoals and so on. Forward chaining refers to a bottom up approach where solutions to simple problems are combined to obtain solutions to larger, more complex problems.

The capture rules enable a first order logical language similar to PROLOG [CLOCKSIN & MELLISH 82] to be used in a relational database framework. An example would be, if we have a database relation EDS(E, D, S) representing employees, their departments and salaries. We might wish to have a "secure" version of the EDS relation where salaries of 100 000 or more are represented as 0. This is expressed as:

```
SecureEDS(e,d,s) :- EDS(e,d,s), s <  100 000
SecureEDS(e,d,0) :- EDS(e,d,s), s >= 100 000
```

Other database researchers, DQL [HAN 86], PROSQL [CHANG 86], have sought to combine PROLOG with relational databases, so as to introduce the deductive power of logic into the database system. The hopes of this research is to enable relational databases to capture more of the semantics of the data. However as [BROD 86] points out, there are still many problems and open issues in this area.

1.4. INTRODUCTION TO RM/T

RM/T, one of the major extensions of the relational model is of interest to us because it includes a form of inheritance called generalization. RM/T uses the concept of *entities* to capture additional semantic information in the database. An entity like a tuple in a relation, is a distinguishable object of some particular type. An entity type represents a relation like STUDENT or PERSON and a particular entity may have several types, e.g., John Smith may be a PERSON as well as a STUDENT.

Entities and their types are classified as:

1. KERNEL: Kernel entities are entities which do not describe other entities. They have an independent existence and are 'what the database is all about'. Typically students, persons, employees or parts will be kernel entities.
2. CHARACTERISTIC: A characteristic entity describes other entities. They arise because of multivalued dependencies and are existence-dependent[5] on the entities

[5]A characteristic entity cannot exist without a kernal entity.

which they describe. For example, a SCHOOL_HIST entity representing the list of schools a student has attended is existent dependent on STUDENT entities.

3. ASSOCIATIVE: Associative entities interrelate two or more entities and represents a many to many relationship among the entities. An ABLE-TO entity representing the courses each instructor is able to teach is an association between courses and instructors.

The E-relation and P-relation are the building blocks of SQUIREL and the constructs and algorithms presented later in this thesis will refer to them extensively.

1.4.1. E-Relation

A unary relation called an *E-relation* is created for **each** entity type and its sole attribute is the *E-attribute* which consists of the character '?' preceded by the relation name, e.g., student.?. The E-attribute is a unique system assigned **surrogate** for each tuple (the system assigns the E-attribute). This surrogate is a system assigned primary key and all references inside the system to each entity is made via this value. An example of an E-relation is given in Fig 1-1.

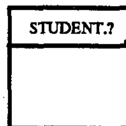

Figure 1-1: STUDENT Entity Relation

1.4.2. P-relation

The immediate (single-valued) properties of an entity type are represented as attributes of one or more P-relations. These relations contain the properties associated with the entity. The first attribute of each P-relation is the system surrogate and its other attributes are the properties attached to the entity in question. An example of a P-relation is given in Fig 1-2.

In order to indicate which P-relations represent attributes associated with which E-relation, the

STUDENT.?	STUD#	DEPT	STATUS	GPA	STARTDATE

Figure 1-2: STUDENT_INFO

relation names of the P and E relations are used in the P-G (property graph) relation. This is a binary relation made up of tuples of the form (SUB:p(m), SUP:e(n))[6] where p(m) is the name of a P-relation and e(n) is the name of an E-relation. An example of a P-G relation is given in Fig 1-3.

SUB	SUP
STUDENT_INFO	STUDENT

Figure 1-3: Property graph relation

1.4.3. Characteristic Graph relation

The characteristic entity types that provide a description of a given kernel entity type form a strict hierarchy called the 'characteristic tree'. In this tree, entity type p is the parent of entity type q if q is an immediate characteristic of p (not a characteristic of a characteristic, for an example see below). Characteristic relations are binary relations which store the multiple values for each entity.

SCHOOL_HIST.?	STUDENT.?

STUD_SCHOOL

Figure 1-4: STUDENT_SCHOOL Characteristic relation

In Fig 1-4, STUD_SCHOOL is the name of the characteristic relation and SCHOOL_HIST is a characteristic entity type. The P-relation for SCHOOL_HIST will have attributes school_name, startdate and enddate and will enable us to obtain information about the schools a student previously attended. A student is assumed to have attended more than one school in his lifetime.

---

[6]SUB stands for SUBORDINATE and SUP stands for SUPERORDINATE.

We could also have a PRINCIPAL_HISTORY entity type which represents the principals which have presided over each school. This characteristic entity type can be looked upon as a characteristic of a characteristic of a student because it is a characteristic of schools which in turn is a characteristic of students. SCHOOL_PRINCIPAL would represent the characteristic relation.

The C-G (Characteristic Graph) relations represents the collection of characteristic trees and are binary relations of (SUB:e(m),SUP:e(n)) similar to the P-G relations. Here e(m) is a characteristic relation and is immediately subordinate to e(n) where e(n) is the name of an E-relation. In the above example STUD_SCHOOL would be subordinate to STUDENT and SCHOOL_PRINCIPAL would be subordinate to SCHOOL. An example of a C-G relation is given in Fig 1-5.

SUB	SUP
STUD_SCHOOL	STUDENT

Figure 1-5: Characteristic Graph Relation

1.4.4. Associative Graph Relation

The A-G (Associative Graph) relation is a binary relation (SUB:e(m), SUP:e(n)) following the two previous graph relations and e(m) belongs to the A-G relation if it participates immediately in the definition of associative entity type e(n). This relation is used to resolve ambiguity that might arise if similar attributes of two different associative entity types were used in another entity type.

If we wanted to set up the course offerings for the semester we would need an INSTR, COURSE pair indicating the instructor who is to teach a particular course.

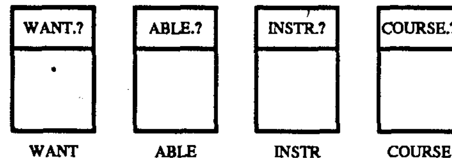

Figure 1-6: Associative and Kernel Entity types

In Fig 1-6, WANT and ABLE are associative entity types. INSTR and COURSE are kernel entity types.

Suppose we are given two relations, defined as:

```
WANT_TO (INSTR, COURSE, FACULTY)

ABLE_TO (INSTR, COURSE, FACULTY)
```

WANT_TO represents INSTRUCTORS and the COURSES they are able to and would like to teach. This is a subset of ABLE_TO. ABLE_TO represents INSTRUCTORS and the COURSES they can teach. An ambiguity arises because we don't know where to obtain the INSTR, COURSE pair for our COURSE_OFF (course offering) relation.

The COURSE_OFF relation is represented as follows:

```
COURSE_OFF (INSTR, COURSE, TIMEofCLASS)
```

If we define COURSE_OFF(INSTR, COURSE) to come from WANT_TO(INSTR, COURSE) we will be able to resolve the ambiguity at the type level but not at the instance level. This is because two instructors may want to teach the same course.

By the use of associative entities the ambiguity is resolved both at the type and instance level. We have the relations WANT_TO, ABLE_TO and COURSE_OFF represented as:

```
WANT_TO     (WANT.?,    INSTR,   COURSE, FACULTY)

ABLE_TO     (ABLE.?,    INSTR,   COURSE, FACULTY)

COURSE_OFF (COURSE.?, WANT.?, INSTR,   COURSE, TIMEofCLASS)
```

The entity type COURSE_OFF uses the attribute WANT.?. This resolves ambiguity at the type level because we know that the attributes INSTR, COURSE, FACULTY are to be taken from the entity type WANT_TO. It also resolves ambiguity at the instance level because each WANT.? refers to specific entities in WANT_TO. An example of an A-G relation is given in Fig 1-7.

SUB	SUP
WANT_TO	COURSE_OFF

Figure 1-7: Associative Graph Relation

This indicates that the tuples to be used in COURSE_OFF are to be taken from the WANT_TO relation.

1.4.5. Generalization

Generalization is an important dimension for forming larger meaningful units. There are two aspects to generalization and they are both forms of specialization. The first is set membership where 'John Smith' might be an instance of the STUDENT and PERSON entity types, this is taken care of by the E-relations and the second is set inclusion where the STUDENT entity type is a subset of the PERSON entity type. Generalization by inclusion is supported in RM/T by the UGI (Unconditional Generalization Inclusion) graph relation. The graph shows entity types which are subtypes of other entity types. In the UGI relation the **property inheritance rule** is applicable: Given any subtype e, all of the properties of its parent type(s) are applicable to e. The graph is represented by a binary relation (SUB:e(m), SUP:e(n)) where e(m) is an immediate subtype of e(n). An example of an UGI graph is shown in Fig 1-8.

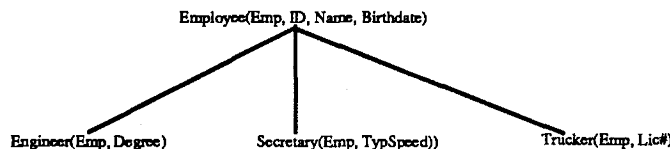

Figure 1-8: An Employee Database

Here engineer, secretary and trucker are subtypes of employee and will inherit all attributes associated with the employee entity type.

1.4.6. Rules

RM/T includes a set of integrity rules so that besides the entity and referential integrity rules of the relational model, we also have:

1. Entity integrity in RM/T: E-relations accept insertions and deletions but not updates.
2. Property integrity: If a tuple t appears in a P-relations P then the surrogate of t must appear in the E-relation corresponding to P.
3. Characteristic integrity: A characteristic entity cannot exist unless the entity it describes exist in the database.
4. Association integrity: An associative entity cannot exist unless each entity participating in the association exists in the database.

5. Designation integrity: A designative entity cannot exist in the database unless the entity it designates is also in the database. Designations represents a many to one relationship between entities. For example, an employee will designate some corresponding department entity.
6. Subtype integrity: If a surrogate appears in the E-relation E, then it will also appear in all E-relations of which E is a subtype.

1.4.7. Operators

RM/T also includes a set of operators to manipulate the entities. An example of an RM/T operator is PROPERTY whose effect is to gather all the properties of a specified entity type regardless of how many properties there are, how they are grouped as P-relations and of the naming conventions used for these P-relations.

1.4.8. Other features of RM/T

RM/T also supports event precedence, aggregation and an AGI (Alternative Generalization Inclusion) graph relation. It also has a system CATALOG which provides information about relation names, attributes, and domain names. More details of these can be found in [CODD 79]. We see that with RM/T, Codd has attempted to deal with many of the issues in database research concerning semantic data modeling. With RM/T, he has presented a set of objects, a set of rules and a set of operators. However as Date has said

> "...a substantial amount of definitional work must be done before they (the operators) are in any final form..."

Hence, these operators as currently defined are only preliminary. The design of SQUIREL is also an effort to provide a language specification for RM/T's generalization hierarchy and one aspect of SQUIREL is to define operators for the UGI hierarchy in some detail.

1.5. Objective

We set out to design SQUIREL using a relational model as a starting point, it was then decided that RM/T be taken as the basis for SQUIREL because it addresses many of the issues dealing with semantics in databases.

The approach taken in the design of SQUIREL is from a database perspective. We are interested in **practicality, efficiency** and **robustness**. An important point to note is that we are looking at enhancing SQL, not at revolutionizing it. It is felt that only the minimum number of new constructs should be added. Another important consideration is that what is currently available in SQL should be kept and made as compatible as possible with any enhancements.

The primary purpose of our effort is to provide new higher level constructs for database programmers, to facilitate easier access to data. Since most database users are already familiar with SQL, we have decided to design our extension to have an SQL-like interface to decrease the learning curve for SQUIREL. From the research point of view, enhancing an existing language will allow the design to focus on new additional concepts without having to worry about issues that have already been dealt with.

1.6. Thesis Organization

In this chapter, we briefly introduced the relational model, some current database research , RM/T and the objective of this research. The rest of this thesis is organized as follows. Chapter 2 presents the features of SQUIREL. Chapter 3 describes how to use SQUIREL constructs. Chapter 4 gives details about how SQUIREL could be implemented. Chapter 5 compares SQUIREL with two prominent AI systems. Chapter 6 provides a conclusion. Appendix A provides a detailed description of how the constructs in SQUIREL are used. Appendix B provides details about the algorithms of these constructs and appendix C gives the pseudo code for a procedure used by SQUIREL.

Chapter 2
FEATURES OF SQUIREL

SQUIREL an extension to SQL, incorporates a *hierarchical structure* which leads to several new features. These features facilitate the development of more reasoning and inferencing in SQUIREL. Since efficiency in access is a primary concern, we shall also consider how these features can be implemented efficiently.

Normal relational operators as well as RM/T primitive operators have been included in SQUIREL. SQUIREL uses RM/T relations to store all its information and base relations are stored as P-relations (property relations) while meta-data information is captured by SQUIREL's hierarchical structure.

An overview of the hierarchical concept as it applies to both traditional databases and SQUIREL is presented before SQURIEL's features are discussed.

2.1. Traditional Database's Hierarchical Concept

Traditionally, the hierarchical model defines physical access paths for linking records together (those records that tend to be accessed together in an application are linked). It consists of an ordered set consisting of multiple occurrences of a single type of tree [DATE 86].

The tree type is made up of a single root record type and an ordered set of zero or more dependent subtree types. The subtree types are also made up of a single root record type and an ordered set of zero or more dependent subtree types. An example of this structure is given in Fig 2-1. Here each department has a department number, the number of employees and a set of employees working in that department.

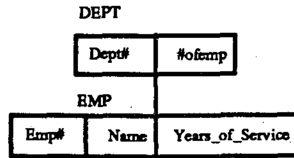

Figure 2-1: A Department hierarchy

Fig 2-1 is an example of a 'single type of tree' since there is only one department which has many employees. The hierarchy would have multiple occurrences of a single type of tree, if there were many departments stored in the database having many employees. There is no semantic information like the names of subclasses stored in the hierarchy. It only provides physical access to an ordered set of records and allows the user to be in a 'get next record' type of situation. In any given application a collection of records are linked together and the user is able to easily access the next record in that application.

2.2. SQUIREL's Hierarchical Concept

The hierarchical structure in SQUIREL allows information to be shared by different record types[7] allowing attributes to be inherited from a common parent record type. To accomplish this, each child record type inherits a unique set of parent attribute values. This is unlike the traditional hierarchical model where many child record type occurrences[8] share one parent record type occurrence. There is a one-to-one correspondence of parent to child record type occurrences in SQUIREL as opposed to the one-to-many correspondence of parent to child record type occurrences in the hierarchical structure of traditional database theory.

The hierarchy in SQUIREL uses views as its components and allows semantic information to be stored in the views. Inheritance, classification and simple default assignments are also supported in the hierarchy.

---

[7]Here different record types refer to different view definitions, i.e., different view schemas.

[8]Record type occurrence refers to an entity or tuple in a relation.

Reasoning about the meaning of data or the concepts associated with a specific collection of data is not dealt with in most relational database systems. For example, given (John Smith, Male , 25), traditional hierarchies cannot answer the question "Is John Smith a Person?", since they contain only pointers, no semantic information. However, SQUIREL can handle this type of query by grouping a collection of views into a hierarchy where these views represent concepts like person or student. The hierarchy then represents a higher level structure which enables us to reason about these concepts. As will be evident in the examples to be presented, these hierarchies are very specialized. Each one is used to answer specific types of queries, e.g., about fees a student has to pay or about courses a student is able to preregister for. Therefore these hierarchies are developed based on the DBA's (database administrator) perception of the use of the database.

2.2.1. Objective of Hierarchies

There are several objectives to incorporating a hierarchy into SQUIREL. Firstly, the hierarchy enables us to support the **property inheritance rule** [CODD 79]: Given any subtype e, all of its parent type(s) are applicable to e. This simply means that when one entity type subsumes another, the attributes of the former will be automatically inherited by the latter.

Secondly, the hierarchy is a structure which avoids **null** values of certain types. Null values can arise because of:

1. entity types have attributes which do not apply to all entities.
2. when creating entities the user may miss some attributes.

Both of these situations should be avoided as they make management of the database more difficult. The hierarchy leads to the elimination of null values in the first case because at each level in the hierarchy only attributes which apply to the entity type will be represented. Thus we will not get a situation where some attributes apply to some entities but not to other entities for the same entity type. With the hierarchy, normalization of entity types is encouraged and connections between entity types are implicitly represented in the hierarchy. The hierarchy also provides a structure for the user to formally manage views because a collection of views can be accessed as one structure, the hierarchy.

The features of SQUIREL's hierarchy are introduced in general terms and the details of the hierarchy will be discussed in section 2.8. The use and an implementation plan of SQUIREL's constructs which use these features is discussed in chapters 3 and 4.

2.3. Inheritance

The idea of inheritance has already been established in databases [SMITH 77] and artificial intelligence [BRAC 79] and there are many knowledge representation systems and languages which deal with this idea. Unfortunately the terminology of inheritance can be confusing and terms such as "subtype", "subclass", "specialization" and "is-a" have been used to imply an ordering among concepts like PERSON and EMPLOYEE.

In SQUIREL we implement a hierarchy to incorporate inheritance. The hierarchy is a generalization hierarchy where the higher level components are generalizations of the lower level components, e.g., both student and employees can be generalized to be persons. In the hierarchy all attributes of the parent are inherited by its children. If the person concept had the attributes sex and age then these attributes would be inherited by the student and employee concepts.

2.4. Classification

Classification in rule based systems has been thought of as a subtype of analysis problems and includes for instance situation assessment, troubleshooting and fault isolation [BROWNSTON 86]. It is well known that there are many systems which solve classificatory problems like diagnosing medical diseases, MYCIN [BUCHANAN & SHORTLIFFE 84] or locating cable problems in telephone networks, ACE [VESONDER 83]. From these systems we see that the ability to classify is indeed a worthy feature to have.

SQUIREL is able to perform some form of inferencing like determining where a newly created v-entity type or entity is to be positioned in a given hierarchy. For v-entity types, this arises when a v-entity type is defined and inserted into a hierarchy. In SQUIREL two classification algorithms (classifiers) facilitate this automatic inferencing.

In order for the user to use the classification algorithms he/she would first define appropriate v-entity types and insert them into hierarchies. The classifier will then automatically determine the structure of the hierarchy by classifying the v-entity types entered into parent and child v-entity types.

In SQUIREL classification is used in two ways:

1. Classifying v-entity types to determine its position in a hierarchy (a general class-subclass classifier).
2. Classifying entities to determine which v-entity types are being represented (a more limited class-instance classifier).

The classifier is invoked on v-entity types entered into each hierarchy. It finds other subsuming or subsumed v-entity types and automatically places the newly entered v-entity types at their proper position in a specified hierarchy.

The advantages of having a classifier are :

1. The user is relieved of having to know the explicit structure of the hierarchy though he/she still has to know which hierarchy he/she is addressing.
2. Provides a means of enforcing consistency of attributes (user will not enter data into the wrong entity types).
3. Can provide a basis for generalized search. When attribute values are needed, the classifier can determine the entity types where these attributes are found.

2.4.1. PlayBack

Associated with the classifier is a playback facility which can be used to follow the path of the classifying algorithm. This facility is useful to the user in cases when he/she requires the logical inferences made by the classifier. This will allow the user to visualize the reasoning done by SQUIREL's classifier and can help in his/her understanding of the problem at hand.

2.5. Default Assignments in SQUIREL

Default reasoning has been well studied in AI [REITER 78a, REITER 80]. In SQUIREL we adopt Reiter's meaning of a default in that "If certain information cannot be deduced from the given knowledge base, then conclude..." [REITER 78b]. Defaults correspond to the assignment of values to attributes of individual entities in the absence of information to the contrary of this assignment. Thus SQUIREL's defaults are similar to those of some knowledge representation scheme like KRL [BOBROW 77] and KEE [FIKE 85] which also explicitly provide for the assignment of default values to variables.

Default values add more reasoning power to closed world databases[9]. It allows SQUIREL to automatically assign values to attributes if no values are given by the user. Instead of not being able to conclude anything in the absence of information, SQUIREL will use meaningful default values set by the user. For example, defaulting a STUDENT's department attribute to GENERAL when he/she first enters university and has not decided on a department is a situation where default assignments would be practical. Defaulting values is an optional feature and the user is not compelled to specify default values for all entity type attributes. In fact there are many cases where defaults would be inappropriate. For example, having a default for the sex of a STUDENT.

An important property of default values is that they cannot be specified for key attributes. This follows from the concept of relational databases where the assignment of default values to key attributes would violate the functional dependency requirement of keys in relations.

The advantage of defaults comes to light when entities are entered into the database. With the addition of default values, the user is able to insert an entity into a hierarchy without specifying all of its relevant attribute values.

[9]These are databases where the absence of information implies that the information is false.

2.6. Management of Views

Current relational database systems only manage views to the extent that the attributes and the entity types from which the views are derived are stored in system tables. The only use of views at present is to form a single customized table from which to retrieve data.

In SQUIREL, views are the basic components of the hierarchy and will lead to v-entity types being linked to each other. We will be using multiple v-entity types to form hierarchies which will lead to the further abstraction of the available data. The user will look at several v-entity types as one conceptual structure, the hierarchy.

2.7. Procedure Invocation

Our design philosophy is to extend SQL to incorporate new features which are simple and concise and also still be compatible with what can be currently done. We realize that SQUIREL could be significantly enhanced if rules were provided but still feel that SQUIREL should **not** contain a rule base. Adding a full fledged rule base to SQUIREL will not only be costly but involve a substantial effort on users to learn new rule constructs. Users typically work with only a small subset of available features and it is felt that the great amount of learning required to effectively use rules in a rule base would deter most users from ever using it. As will be evident from later chapters SQUIREL's hierarchical structure already provides some basic functions normally defined using rules in a system like KEE.

In addition, most SQL systems already allow an interface to a high level programming language like PL/1, PASCAL or C. If procedures are needed in SQUIREL, this interface will be used and the procedures will be written in a high level programming language. In AI systems/languages with a rule base, a basic necessity is to have a mechanism which will cause the rules to fire. KEE uses a triggering mechanism called *active values* to invoke rules. OPS5 [FORGY 81] needs a start or initialize procedure to invoke rules and PROLOG needs to instantiate a predicate to trigger rules.

We see that in any rule based system some form of control to invoke rules is essential. SQUIREL provides just such a construct called TRIGGER. This new construct can be attached to any operation on entity types, i.e., SELECT, UPDATE, DELETE, INSERT and other operations.

2.8. SQUIREL's hierarchy

We will now go on to a more detailed description of SQUIREL's hierarchy.

2.8.1. Description of Hierarchy

SQUIREL's hierarchy is an extension of the powerful view mechanism currently available in relational database systems in that it is a hierarchy of *views*, i.e., all components in the hierarchy are implemented as views. We develop a model for hierarchies based on RM/T's UGI hierarchies [CODD 79].

A hierarchy in SQUIREL is a collection of v-entity types linked together in a structure which represents the users' perception of how the v-entity types *subsume* or are *subsumed* by each other. The hierarchy provides us with the ability to perform updates, insertions, and deletions of entities without having to know the v-entity types in which these entities are stored.

Using a hierarchical structure, views can be easily managed. Connections between views are made explicit and we now have a mechanism by which we can use views other than for retrieving entities.

2.8.2. Categories and Partitions

Two important attributes which arise because of the hierarchical structure are *category* and *partition*. Both of these provide information about the **structure** of the hierarchy, that is, subsumed or subsuming v-entity types. The category is an attribute attached to each hierarchy and its values are all the v-entity types in the hierarchy. It is used when a hierarchy is referenced in a query. The partition is an attribute attached to each v-entity type (A) in a hierarchy and its values are the v-entity types that A subsumes. It is used when a specific v-entity type is referenced in a query. We are thus representing meta-knowledge[10] about v-entity types and entities. Instead of only getting information about attribute values, information about subsumed v-entity types or the v-entity type where an entity is found can be accessed.

---

[10]Meta-level knowledge generally consists of creating one or more layers of knowledge above the rule base [HARANDI 86] but in our case we are building a layer of knowledge above the database.

The category represents the heading to be used for all v-entity types in a hierarchy. Similarly partition represents a heading to be used for the subsumed v-entity types of a particular v-entity type in a specific hierarchy.

2.8.3. Properties of the Hierarchy

Some of the properties which are applicable to hierarchies are that child v-entity types in a hierarchy **must** have more attributes than its parent v-entity types. In this way, attributes which apply to all v-entity types are attached to the top level v-entity type.

We do not allow another hierarchy relation name to be entered as a v-entity type of a hierarchy. Therefore, we cannot define a hierarchy of hierarchies. Another property is that the same v-entity type can participate in multiple hierarchies. This means that a student v-entity type could be in a university personnel hierarchy as well as an employment hierarchy. Since v-entity types may participate in more than one hierarchy, for each hierarchy a v-entity type participates in, a **different** partition name is required. Note that the same partition name can be used among different v-entity types in the same hierarchy. This means that if the STUDENT v-entity type had the partition name of STATUS in the university personnel hierarchy, the NONSTUDENT v-entity type in the same university personnel hierarchy could also have the partition name of STATUS.

Another property is that hierarchy names must be unique in the database. This allows the same category name to be used for different hierarchies. This implies that two hierarchies can have the same category name and the university personnel and employment hierarchy might have the category name of TYPE.

A final property of hierarchies is that only unique subsumption links are stored in a hierarchy relation that is if the tuple (A, B) and (B, C) are in the hierarchy relation then tuple (A, C) will not be. It can be implicitly derived by transitivity. Here A, B and C are v-entity type names.

Chapter 3
SQUIREL CONSTRUCTS AND THEIR USE

We will base most of our examples about the use of constructs on the PERSONNEL hierarchy shown in Fig 3-1.

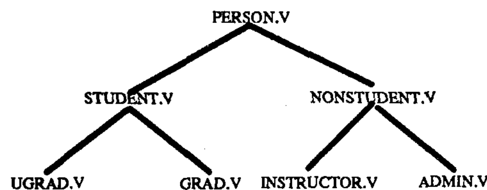

Figure 3-1: Personnel UGI hierarchy

The base entity types[11] and their attributes (stored in P-relations) are:

```
PERSON      (PERSON.?,SIN#,NAME,SEX,AGE)

STUDENT     (PERSON.?,STUD#,DEPT,GPA,STARTDATE)

NONSTUDENT (PERSON.?,OFFICE,QUALIFICATION)

GRAD        (PERSON.?,LAST_DEGREE)

UGRAD       (PERSON.?,MAJOR)

INSTRUCTOR (PERSON.?,CURR_WORK)

ADMIN       (PERSON.?,JOBTITLE)
```

All the v-entity types in the PERSONNEL hierarchy shown in Fig 3-1 are defined using the above base entity types. An example of a definition of an entity type and a v-entity type follows:

---

[11]Base entity types are base relations used to form v-entity types.

```
CREATE TABLE PERSON( SIN#  NUMBER(9),    NAME   CHAR(20),
                     SEX   CHAR  (6),    AGE  NUMBER(3));

CREATE TABLE STUDENT(STUD# NUMBER(9),    DEPT   CHAR(20),
                     GPA   NUMBER(3,2),  STARTDATE DATE);

CREATE TABLE GRAD   (LAST_DEGREE CHAR(10));

CREATE VIEW  GRAD.V
   AS  SELECT        SIN#  ,NAME,SEX,AGE,STUD#,DEPT,
                     GPA   ,STARTDATE,LAST_DEGREE
   FROM              PERSON,STUDENT,GRAD;
```

The first command creates a base entity type person which has the attributes sin#, name, sex and age. The second and third commands create the base entity types student and grad. Recall that RM/T provides a surrogate for all base entity types that is why grad can be defined without a user given key. The last command defines a v-entity type GRAD.V to have attributes sin#, name, sex, age, dept, g.p.a, startdate and last_degree. GRAD.V is derived from the base entity types PERSON, STUDENT and GRAD.

The database will consists of many base entity types storing unique attributes. The user with the aid of the hierarchical structure will create 'windows' of the database specialized to suit his/her purposes. Without the hierarchical structure the user would lose the ability to store meta-knowledge, that is information about the hierarchy structure. In other words, the user is unable to access information about the subsumed or subsuming v-entity types in a hierarchy. However, using the PERSONNEL hierarchy we can obtain information about the structure of the hierarchy using the following query:

```
SELECT PERSONNEL.CATEGORY
FROM   PERSONNEL.HIERARCHY;
```

This will return PERSON.V, NONSTUDENT.V, STUDENT.V, UGRAD.V, GRAD.V, INSTRUCTOR.V and ADMIN.V which are the names of all the v-entity types in the personnel hierarchy.

To obtain information about the INSTRUCTOR 'Mike Cray', we could make the following query:

```
SELECT * FROM PERSONNEL.HIERARCHY
 WHERE   Name = 'Mike Cray';
```

Assuming Mike Cray is a graduate student, this query would return (765900453, Mike Cray, Male, 34, 854903211, Mathematics, 4.00, 090584, Bsc) which indicates the sin#, name, sex, age, stud#, department, g.p.a, start date and last degree of Mike Cray. Note that we do not have to know where Mike Cray lies in the hierarchy. The hierarchy will automatically search for the appropriate v-entity type which contains Mike Cray and return the required information.

3.1. Base entity types/v-entity types

We have defined the generalization hierarchy slightly differently from RM/T and use v-entity types instead of entity types as components in the hierarchy because this facilitates a simple, concise way in which hierarchies can be implemented. To distinguish v-entity types from entity types, the name of a v-entity type will have a .V suffix.

When an entity type is created, a unique E-relation (storing the system surrogate) is assigned to that entity type. The WHERE clause in SQUIREL is a condition clause and can be either used to specify an integrity constraint like AGE < 25 or used to join two or more entity types by directing the system to match attributes of these entity types, e.g., WHERE PERSON.NAME=EMPLOYEE.NAME. When a v-entity type is created either without the WHERE clause, as shown in the example on page 26, or if the WHERE clause references the key attributes of two or more entity types, then SQUIREL will internally equate the E-relations of these base entity types. This occurs when the v-entity types are entered into a hierarchy. In the above example, all E-relations of the different entity types are set to be the PERSON E-relation. That is, the system surrogate for each of the entity types defined above will be the E-attribute, PERSON.?. Appendix A.13 provides an algorithm to coalesce E-relations.

The v-entity type used in SQUIREL encompasses all the concepts of the view in relational database systems and also includes:

- The storage of the names of subclasses.
- Use of defaults values.
- Use of two different classification algorithms. A general class-subclass as well as a class-instance classifier.

3.2. Hierarchy Constructs

There are two levels in the hierarchical structure of SQUIREL: the TYPE level and the INSTANCE level. The TYPE level deals with the concept of class and subclass[12]. The INSTANCE level deals with the concept of class and instances.

The following is a brief description of the constructs that have been added to SQUIREL which are used specifically to manipulate the TYPE level of the hierarchical structure. At this level the v-entity types are the important concepts and creating or removing a hierarchy, inserting, updating, deleting and retrieving v-entity types from a hierarchy are the major constructs presented. Appendix A provides a more detailed explanation of each of these constructs while the algorithms for these constructs and for the two types of classification are given in appendix B. In the following examples, user defined variables are denoted by lower case while the constructs are denoted by upper case.

3.2.1. CREATION

The following is an example of creating a hierarchy using the SQUIREL CREATE construct:

```
CREATE HIERARCHY  personnel
       CATEGORY = university_personnel;
```

In this example, a hierarchy relation named PERSONNEL is created. Every hierarchical relation will have two system assigned attributes: SUB (subordinate) and SUP (superordinate) which are used to record all the v-entity types that participate in the hierarchy. The CATEGORY attribute is an alias for the SUB attribute so that in the above example, university_personnel can be used whenever the keyword SUB is required. The categorization refers to v-entity types and indicates the general theme, idea or type of structural breakdown of the hierarchy.

[12]Classes are represented by v-entity types while instances are represented by entities.

3.2.2. INSERTION

To insert into the hierarchy we would use the SQUIREL INSERT construct:

```
INSERT INTO personnel.hierarchy
V-ENTITY =  person.v,      PAR = status,
V-ENTITY =  student.v,     PAR = level,
V-ENTITY =  nonstudent.v,  PAR = type,
V-ENTITY =  grad.v,
V-ENTITY =  ugrad.v,
V-ENTITY =  instructor.v,
V-ENTITY =  admin.v;
```

The v-entity types specified are entered into the personnel hierarchy and the type level classifier will determine which v-entity types subsume which other v-entity types and thus determine the structure of the hierarchy. In our example, the PERSONNEL hierarchy will be formed as shown in Fig 3-1.

The PAR (partition) attribute is a system assigned attribute which stores the subsumed v-entity types of a given v-entity type. The v-entity types which have PAR specified will use the user given names to reference their subsumed v-entity types while the v-entity types without a user given partition name will use a default name of <hierarchy>.PARTITION. In the above example, four v-entity types (GRAD.V, UGRAD.V, INSTRUCTOR.V, ADMIN.V) will use the default partition name (PERSONNEL.PARTITION). Using the same partition name for v-entity types will not lead to problems when retrievals are required because in each of the v-entity types the partition name is unique.

3.2.3. UPDATES

To update a v-entity type in a hierarchy we use the SQUIREL UPDATE construct:

```
UPDATE personnel.hierarchy
SET    PAR       = "occupation",
WHERE  V-ENTITY =  nonstudent.v;
```

The partition attribute for NONSTUDENT.V is changed to occupation and all references to NONSTUDENT.V's subsumed v-entity types will now be through the attribute occupation. This is the only update operation that is allowed on the hierarchy because altering the structure of the hierarchy can only be done by the type level classifier.

3.2.4. DELETION

To delete a v-entity type from a hierarchy we use the SQUIREL DELETE construct:

```
DELETE FROM personnel.hierarchy
WHERE  V-ENTITY = grad.v;
```

The GRAD.V v-entity type will be removed from the personnel hierarchy and the type level classifier will be invoked to restructure the hierarchy due to the deletion of a v-entity type. In our example, the link from STUDENT.V to GRAD.V will be removed because GRAD.V is a leaf v-entity type whose parent v-entity type is GRAD.V.

3.2.5. DROP

To remove a hierarchy from the databases we use the SQUIREL DROP construct:

```
DROP HIERARCHY personnel.hierarchy;
```

The personnel hierarchy relation is removed and will no longer exist in the database though the v-entity types that formed the hierarchy are still in existence.

3.2.6. RETRIEVAL

To retrieve information about the structure of v-entity types in a hierarchy we use the SQUIREL SELECT construct:

```
SELECT university_personnel FROM personnel.hierarchy;
```

This query will return all the v-entity types that form the hierarchy. In our example, PERSON.V, STUDENT.V, NONSTUDENT.V, INSTRUCTOR.V, ADMIN.V, GRAD.V and UGRAD.V will be returned.

We will also be able to query individual v-entity types about the v-entity types that it subsumes. Given that the PERSON.V v-entity type has been given the partition name STATUS on page 29, the following SQUIREL command is used to query PERSON.V:

```
SELECT status  FROM person.v
```

This will return STUDENT.V and NONSTUDENT.V which are the two v-entity types that PERSON.V immediately subsumes.

The database administrator is the one who most likely sets up the structure of the hierarchy while

the database user is the one who uses the hierarchy by entering entities at the instance level. We will now briefly describe SQUIREL constructs which are used at the INSTANCE level. At this level the important concepts are the individual entity in a hierarchy and insertion, deletion, updates and retrieval of entities from a hierarchy are the major constructs presented.

3.2.7. INSERTION

To insert entities into the hierarchy we have to specify the key attribute[13] of the entity and then use the SQUIREL INSERT construct:

```
INSERT INTO personnel.hierarchy
VALUES( Name='John Smith', Office = 'LB 1214');
```

The instance level classifier is invoked to determine the v-entity type which contains the entity in question. In the above example, this would be NONSTUDENT.V. The entity represented by John Smith is then inserted into the base entity types of NONSTUDENT.V which are PERSON and NONSTUDENT. Since the NONSTUDENT.V v-entity type has the attributes name, sex, age, office and qualifications, the attribute values not specified for John Smith will be assigned as unknown or have default values if they are specified.

A new person has been created and this person will participate in all the hierarchies that the v-entity type PERSON.V participates in, for instance if PERSON.V is also found in the employment hierarchy then John Smith will also be found in the employment hierarchy.

3.2.8. DELETION

To delete an entity from a hierarchy we would use the SQUIREL DELETE construct:

```
DELETE FROM personnel.hierarchy
WHERE  Name = 'John Smith';
```

The instance level classifier is used to determine the v-entity types where John Smith is found. In our example, if John Smith is in the ADMIN.V v-entity type and is represented by the following entity:

```
(John Smith, Male, 25, LB1211, PostSecondary, Accountant)
```

---

[13]Key attributes are attributes which have been indexed using the INDEXED command of SQUIREL and in our example we assume the attribute Name has been indexed.

which tells us that John Smith is male, 25 years old, has his office at LB1211, has a post secondary qualification and is an accountant. John Smith will then be removed from the base entity types of ADMIN.V, namely the PERSON, NONSTUDENT and ADMIN base entity types.

3.2.9. UPDATES

To update an entity in a hierarchy, the attributes to be updated as well as the key attribute which references this entity must be given. The SQUIREL UPDATE construct accomplishes this task:

```
UPDATE personnel.hierarchy
SET    jobtitle = "librarian",
WHERE  Name     = 'John Smith';
```

The instance level classifier determines the v-entity type that contains John Smith and has the attributes to be updated. In our example, this would be the ADMIN.V v-entity type because ADMIN.V is the only v-entity type that has the attribute job title. The base entity type of ADMIN.V that contains the attribute to be updated is then changed and the job title of John Smith will be changed to librarian from accountant. By using more SETs, e.g., SET Dept = mathematics, in the update command, any number of relevant attributes can be changed.

3.2.10. RETRIEVAL

To retrieve information about the entities in a hierarchy the SQUIREL SELECT constructs are used:

```
SELECT  university_personnel FROM personnel.hierarchy
 WHERE  Name = 'John Smith';

SELECT    *       FROM personnel.hierarchy
 WHERE  Name = 'John Smith';

SELECT  status    FROM person.v
 WHERE  Name = 'John Smith';
```

For the first query, all the v-entity types in the personnel hierarchy where John Smith can be found are returned and these would be PERSON.V, NONSTUDENT.V and ADMIN.V. For the second query, the * indicates that all attributes are required and the attribute values associated with John Smith are retrieved which might be the following:

```
(John Smith, Male, 25, LB1211, PostSecondary, Accountant)
```

This represents the name, sex, age, office, qualification and job title of John Smith. In the third

query, only the v-entity type directly below PERSON.V where John Smith can be found is returned and this would be NONSTUDENT.V.

One point of interest is that for every construct used at the instance level, the user is relieved of the need to know in which v-entity types an entity can be found. This is because the instance level classifier is invoked for all instance level queries and will determine the needed v-entity types required in each given query. The user is only required to query the hierarchy using key attributes.

3.3. Procedure Invocation

When procedures are written for entity types, the TRIGGER construct can be attached to any of the above constructs. <procedures> will then be invoked when operations (SELECT, DROP etc.) are performed on the entity types. There are three ways in which the TRIGGER construct can be used and they are described in turn.

The first way TRIGGER is used will be in conjunction with an SQL operation as in:

```
<SQL operation> TRIGGER <procedure>
```

An example is shown below:

```
UPDATE    personnel.hierarchy
SET       PAR          = "occupation"
WHERE     V-ENTITY     =  nonstudent.v;
TRIGGER   playback;
```

<SQL operation> : any allowable SQUIREL operation

<procedure> : must be the name of a program defined in a high level programming language interfaced with SQUIREL, which contains rules to be invoked.

In this example, when the update is performed on NONSTUDENT.V, the playback procedure (explained in the next section) is invoked.

Alternatively the procedure can be attached to an entity type directly in the CREATE command. This will cause the program to be invoked whenever the entity type is referenced. The advantage of this method for invoking procedures is that it is more practical if procedures are to be invoked frequently or if procedures are to be invoked on all operations to a particular entity type.

The form of TRIGGER here would be:

```
CREATE TABLE  <entity type>
                   .
                   .
TRIGGER       <procedure>;
```

An example is shown below:

```
CREATE TABLE person (name char(12), sex char(12), age numeric)
TRIGGER playback;
```

Here any time an access or reference (indirectly or directly) is made to the PERSON base entity type, the PLAYBACK procedure is invoked. Therefore, if we had referenced PERSON directly as in:

```
    INSERT INTO person
    VALUES('John Smith', 'Male', '25');
```

or if we had referenced PERSON indirectly as in:

```
    INSERT INTO personnel.hierarchy
    VALUES( Name='John Smith', Office='LB1211');
```

the PLAYBACK procedure will be invoked.

Thirdly the TRIGGER mechanism could be used in the same way as a *call* or *invoke* in a high level programming language. In this case, all the user is required to do is to TRIGGER a specified procedure as shown below:

```
TRIGGER playback;
```

Here the PLAYBACK procedure is invoked directly similar to a call in a high level programming language.

Whenever TRIGGER is used, the <procedure> called may be given arguments and these arguments are inserted into a special system entity type called <procedure>.ARG. There will be one of these entity types for each procedure defined in SQUIREL. These entity types are automatically created when a procedure is attached to a base entity type. The user can define the type of arguments a procedure is to have by using the SETARG construct as shown below:

```
SETARG callme.ARG
 arg1 CHAR(20), arg2 CHAR(20), arg3 NUMBER;
```

In the above example, callme is a procedure and its first and second arguments are of type character 20 while its third argument is a numeric value. Further details of SETARG are provided in appendix A.12.

An example of specifying argument values for a <procedure> and invoking it follows:

```
INSERT INTO callme.ARG
VALUES('John Smith', 'Male', '25');

TRIGGER callme;
```

In the above, callme will be able to use the three arguments John Smith, male and 25 stored in callme.ARG.

SQUIREL provides a triggering mechanism not currently available in SQL which enables procedures written in a high level language to be invoked from within the SQL environment.

3.3.1. PlayBack procedure

A special system procedure PLAYBACK is encoded into SQUIREL. It uses the hierarchy relation to construct the path that the classification algorithm uses. Given a hierarchy and a v-entity type, a specific path will always be produced. This is because in any given hierarchy each v-entity type will only have one parent. For an example of a hierarchy relation see Fig 4-1.

As seen in the previous examples, the PLAYBACK procedure can be invoked in several ways and whenever it is invoked the path (sequence of the classification) of the **currently** referenced v-entity type in the **currently** referenced hierarchy is given. PLAYBACK requires two arguments, the name of the hierarchy being checked and the v-entity type whose path is required.

3.4. Use of Default Assignments

In SQUIREL, all entity **inserts** into a hierarchy must have corresponding attributes named in the command and must include the key attribute. With the addition of defaults those attributes which are not explicitly given will have default values.

Defaults are optional and the user does not have to provide defaults for all attributes. The user is also not allowed to specify defaults for any of the key attributes (indexed attributes).

The user will enter defaults as follows:

```
INSERT INTO <entity type>.default
   <attribute> = <value>;
```

An example of this is shown below:

```
INSERT INTO student.default
   DEPT =   "general";
```

Here the default value for the department attribute of the student entity type is set to be general. If the user inserts entities into the personnel hierarchy where the entities are students and if he/she does not specify the student's department attribute then the student's department attribute will be set to be general. The student entity type has several more attributes, Sex, Age, G.P.A and Startdate, which will not have default values because default values were not specified for the student.default entity type. The user can also override previous defaults by repeating the above command and inserting new values into the default entity type as in:

```
INSERT INTO student.default
   DEPT =   "mathematics";
```

Given the PERSONNEL.HIERARCHY in Fig 3-1 each v-entity type will inherit the default values assigned to the attributes of its base entity types. Assume we have the STUDENT default relation as above and enter a new entity into the PERSONNEL hierarchy.

```
INSERT INTO PERSONNEL.HIERARCHY
VALUES(name='John Smith',Sin#='865749',last_degree='Bsc');
```

Recall that GRAD.V has the attributes Name, Sin#, Stud#, Dept, Last_degree, Sex, G.P.A and Startdate. Using the instance level classifier, we find that the newly entered entity will be in GRAD.V. The v-entity GRAD.V will have "GENERAL" as its default value for department (inherited from the STUDENT.default entity). Thus this new entity will have "GENERAL" as his department. We see that default assignments are used whenever inserts into entity types are performed.

3.5. Management of views/v-entity types in SQUIREL

From previous examples we see that some management of views has been implicitly taken care of in that we know the entity types the view uses. This means that each view is automatically updated if the entity types or views it uses are updated. The user does not have to specify any other control mechanism given this feature of views.

We want to represent a hierarchy that shows the amount of fees a graduate student must pay. First and second year students pay $429 while students who have been in the program for more than 2 years pay $179.

The following example illustrates how views can be grouped into a HIERARCHY:

```
CREATE VIEW FEE.V(stud#)
AS SELECT   stud# FROM GRAD.V;

CREATE VIEW 1STYEARGRAD.V(name,stud#,fees)
AS SELECT   name, stud#, assign(429)
   FROM     GRAD.V
   WHERE    days_in_program  <= 365;
```

Here the view 1STYEARGRAD.V specifies a first year graduate student to be one who has been in the program for less than a year. It then assigns the fee that the student is to pay into the v-entity type.

The v-entity types 2NDYEARGRAD.V, OLDGRAD.V and the PAYMENT hierarchy are defined as follows:

```
CREATE VIEW 2NDYEARGRAD.V (name,stud#,fees)
AS SELECT   name, stud#, assign(429)
   FROM     GRAD.V
   WHERE    days_in_program  <= 730
   AND      days_in_program  >  365;

CREATE VIEW OLDGRAD.V (name,stud#,fees)
AS SELECT   name, stud#, assign(179)
   FROM     GRAD.V
   WHERE    days_in_program  >  730;

CREATE HIERARCHY PAYMENT;

INSERT INTO PAYMENT.HIERARCHY
V-ENTITY = 'FEE.V',
V-ENTITY = '1STYEARGRAD.V',
V-ENTITY = '2NDYEARGRAD.V',
V-ENTITY = 'OLDGRAD.V';
```

We would thus have the hierarchy as shown in Fig 3-2.

Figure 3-2: Payment.Hierarchy

The interesting feature to note when *inserting* into hierarchies is that the user need not specify the connections between v-entity types. The view name representing a v-entity type is entered into a hierarchy and SQUIREL's classifier will automatically determine the v-entity types that subsume it as well as those that it subsumes.

What we have defined above is the PAYMENT HIERARCHY which will determine how much fees a student has to pay. Every graduate student will be represented in this hierarchy and whenever the GRAD base entity type is updated, the PAYMENT hierarchy will also be updated. Although every graduate student will have a tree associated with him/her in the form of a hierarchy, this is not wasteful because the hierarchy is made up of views which are not physically stored. Default assignments could also be used without the PAYMENT hierarchy where we set the default value of fees a graduate student is to pay at $429. In this case the user has to explicitly override this default with another value when a graduate student does not need to pay this amount of fees. This is unlike

the PAYMENT hierarchy where the user does not need to know anything about the fees a graduate student has to pay but instead allows the hierarchy to determine the fees.

A typical query of the above HIERARCHY would be:

```
SELECT name, fee
FROM   PAYMENT.HIERARCHY
WHERE  stud# = '85300 4532';
```

This query might return the entity (Greg James, 429).

It should be seen that the above query relies on the ability to reason on the facts given in a database. We certainly do not expect the casual user to know the conditions imposed for charging of fees (e.g. foreign students have to pay differential fees, first and second year graduate students pay more than graduate students who have been in the program longer). What is asked for is not a specific v-entity type, but rather information stored in a hierarchy. The hierarchy is able to function as a meta-rule, infer from the hierarchical structure where the information is stored and provide the user with an answer.

This reasoning process associated with hierarchies stems from their ability to classify specific entities (instance level) into the v-entity type where they belong and being able to classify v-entity types into parent and child v-entity types (type level). The classifiers thus plays an important role in reasoning.

3.5.1. Views not applicable to SQUIREL's Hierarchy

A very important point to note is that not all user defined views or v-entity types can be used in a hierarchy. Only in situations where **different** kinds of facts are to be recorded about different members of the hierarchy do we separate these members into different v-entity types.

There can be situations where v-entity types on the same level have the same attributes but subsuming (parent) v-entity types can never have more attributes than their subsumed (child) v-entity type. To further illustrate this point SQUIREL does not use hierarchies when the v-entity types do not have unique attributes. This may happen when v-entity types are specializations of some other v-entity types.

Assume we define a honor student v-entity type as follows:

```
CREATE VIEW HON_STUDENT.V(Name,Sex,Dept,Stud#,G.P.A)
AS SELECT   Name,Sex,Dept,Stud#,G.P.A
   FROM     STUDENT.V
   WHERE    G.P.A > 3;
```

To define a v-entity type that stores the female honor students in the computing science department we do the following:

```
CREATE VIEW FEM_CS_HON_STUDENT.V(Name,Stud#,G.P.A)
AS SELECT   Name,Stud#,G.P.A
   FROM     HON_STUDENT.V
   WHERE    Dept = 'Comp Sc'
   AND      Sex  = 'Female';
```

To define a v-entity type that stores all the computing science students we do the following:

```
CREATE VIEW CS_STUDENT.V(Name,Sex,Stud#,G.P.A)
AS SELECT   Name,Sex,Stud#,G.P.A
   FROM     STUDENT.V
   WHERE    Dept = 'Comp Sc';
```

If we wanted to create a hierarchy we would then do:

```
CREATE HIERARCHY STUDENT.HIERARCHY;

INSERT INTO STUDENT.HIERARCHY
V-ENTITY = STUDENT.V,
V-ENTITY = HONOR_STUDENT.V,
V-ENTITY = FEM_CS_HONOR_STUDENT.V,
V-ENTITY = CS_HONOR_STUDENT.V;
```

We might visualize the hierarchy as given in Fig 3-3.

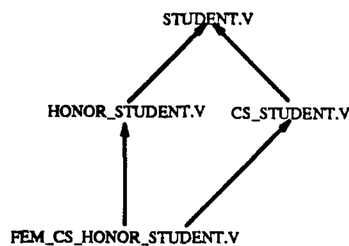

Figure 3-3: Illegal Student Hierarchy

At each level no unique attributes are present and the child v-entity types may in fact have fewer attributes than their parents. This will be in direct contrast to a fundamental property of SQUIREL's hierarchies. Thus the above hierarchy is invalid and cannot be formed by SQUIREL.

3.6. Use of Constructs in the Hierarchy

In this section we will further describe with examples the use of the hierarchy in retrieval operations.

It should be noted that there are two levels of knowledge that can be obtained from the hierarchy. We might want to obtain information about v-entity types or we might want to obtain information about the attribute values attached to the v-entity types.

There are two ways in which SQUIREL uses the hierarchy to obtain information:

1. It can be used like a normal relation (Explicitly).
2. SQUIREL will traverse the hierarchy relation to obtain the attributes of a v-entity type (Implicitly).

When using a hierarchy relation explicitly we remember that the two system attributes of every hierarchy are SUB and SUP, using these attributes we can find all the subclasses of a v-entity type in a particular hierarchy.

An example as to how this could be done is:

```
SELECT SUB FROM personnel.hierarchy
WHERE  SUP  =  "student.v";
```

This is a query asking for the subsumed v-entity types of the STUDENT.V v-entity type and for our example, GRAD.V and UGRAD.V will be returned

The hierarchy is used to distinguish between different classes. We have the <category> for each hierarchy as well as the <partition> for each v-entity type. We can partition people according to their occupation and may create v-entity types which reflect this, e.g., create views of ENGINEER.V and SECRETARY.V, and in this hierarchy a person's occupation is either an engineer or secretary. We find that the information about a person's occupation is 'hidden' in the name of the v-entity type and the user is unable to obtain that information. With the hierarchy the user can obtain this 'hidden' knowledge with the use of the <category> of the hierarchy or the <partition> of a v-entity type.

In Fig 3-4, we have an EMPLOYEE hierarchy relation where the <category> is JOBTYPE. That is, each of the v-entity types in the hierarchy can be seen as a JOBTYPE.

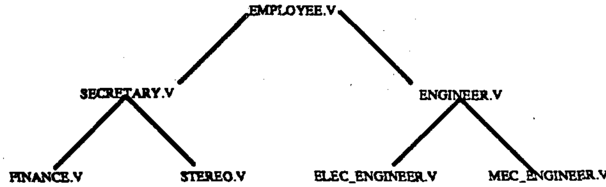

Figure 3-4: Employee.Hierarchy

An example of querying the category is if we did the following:

```
SELECT JOBTYPE FROM EMPLOYEE.HIERARCHY;
```

Here JOBTYPE is the categorization (category name given when the hierarchy is created) of the hierarchy. That is, the general theme/purpose of the hierarchy is to categorize a person according to his/her job title. When we query the hierarchy, all the v-entity type names in the hierarchy are returned.

JOBTYPE
EMPLOYEE
ENGINEER
SECRETARY
FINANCE
STEREO
MEC_ENGINEER
ELEC_ENGINEER

Figure 3-5: Jobtypes Returned

If we need information about a specific person we do the following:

```
SELECT JOBTYPE FROM EMPLOYEE.HIERARCHY
 WHERE Name = 'John Smith';
```

SQUIREL will determine all the v-entity types where the entity specified is found using the instance level classifier and inform the user of the job types of John Smith. For our example,

JOBTYPE
ELEC_ENGINNER
ENGINEER
EMPLOYEE

Figure 3-6: Jobtypes Returned

SQUIREL might return the table shown in Fig 3-6. This indicates that John Smith is an electrical engineer.

To obtain information about v-entity types we could also query the partition attribute of a v-entity type as shown below:

```
SELECT OCCUPATION FROM EMPLOYEE.V;
```

OCCUPATION is the **partition** name (virtual attribute) given to the v-entity type EMPLOYEE.V for the EMPLOYEE hierarchy that is, EMPLOYEE.V is partitioned into either ENGINEER.V or SECRETARY.V under OCCUPATION. SQUIREL will look under the EMPLOYEE.V v-entity type and return the names of the v-entity types that it subsumes. This is shown in Fig 3-7.

OCCUPATION
SECRETARY
ENGINEER

Figure 3-7: Occupations Returned

To determine the OCCUPATION of a specific EMPLOYEE we make the following query:

```
SELECT OCCUPATION FROM EMPLOYEE.V
 WHERE name = 'John Smith';
```

This query will return the v-entity type under EMPLOYEE.V where John Smith lies, which is shown in Fig 3-8. Notice here that when we query the partition of EMPLOYEE.V only the v-entity types directly under EMPLOYEE.V are returned whereas when we query the category of a hierarchy all the subsumed v-entity types at every level are returned.

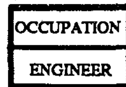

Figure 3-8: Occupations Returned

We see that the information required by the user might not be an actual attribute of any v-entity type but the subsumed or subsuming v-entity types of a particular entity or entity type. We can query the database through the hierarchy or a v-entity type in a hierarchy to obtain information about subsumed v-entity types.

Recall that v-entity types may participate in more than one hierarchy. Unique <partition> and hierarchy names ensure that SQUIREL does not obtain the wrong subsumed v-entity types. When a partition name or a hierarchy name is specified, SQUIREL will know which hierarchy to use in order to obtain the necessary information.

A case in point might be if EMPLOYEE.V is also in another hierarchy, as shown in Fig 3-9.

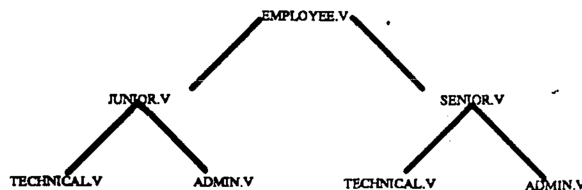

Figure 3-9: Promotion.Hierarchy

The same EMPLOYEE.V v-entity type is used but the hierarchy name is different. The <partition name> for EMPLOYEE in the PROMOTION hierarchy would not be OCCUPATION but might be LEVEL instead.

We also note that if we want to know the <category> or <partition> of specific entities we must reference the entity uniquely, as shown below:

```
SELECT OCCUPATION FROM EMPLOYEE.V
 WHERE name = 'John Smith';
```

The following query will not reference a specific entity:

```
SELECT OCCUPATION FROM EMPLOYEE.V
 WHERE sex = 'Male';
```

If we specify a collection of entities then SQUIREL will return all v-entity type names that have at least one instance of this collection.

Chapter 4
IMPLEMENTATION PLAN FOR SQUIREL

To implement SQUIREL, we need to first of all evaluate current DBMSs and determine which one would be the most appropriate to use in terms of extensibility, robustness and efficiency. Parts of RM/T should be first implemented and the basic requirements would be to have the E-relations, the P-relations and a procedure that assigns system surrogates to new entities. Assuming we have all of these, we now provide a description of the storage structures needed for SQUIREL and some uses of SQUIREL's features.

Efficiency is a primary concern and the first issue to consider is the storage structures which are needed to implement the hierarchy. In SQUIREL there will be at least two structures needed, the hierarchy relation (there will be one for each hierarchy) and a system hierarchy table (SYSHIER).

4.1. Hierarchy Relation

A SQUIREL hierarchy is represented as a relation which has a **unique** hierarchy relation name and is made up of two attributes, SUB (Subordinate) and SUP (Superordinate). Each tuple in the relation consists of two v-entity types (view names). The first attribute indicates the subsumed or child v-entity type and the second attribute represents the subsuming or parent v-entity type. An example is shown in Fig 4-1 which is the PERSONNEL hierarchy relation.

A hierarchy relation is used by the retrieval constructs (SELECT) and the classifiers (type and instance). Since efficiency is a primary concern, both fields of the relation will be **indexed**. Thus, knowing which v-entity types subsume which others can be accessed very efficiently. An indexing scheme we could use would be ISAM (indexed sequential access method) where the records are sorted and there will be a one level index to access the appropriate entities in a hierarchy relation. The index will consists of short records (*v*,*b*) which represent a key value and a block address pair while the main file will have a v-entity type with its parent v-entity types. This is shown in Fig 4-2.

SUB / UNIVERSITY PERSONNEL	SUP
PERSON.V	TOP
STUDENT.V	PERSON.V
NONSTUDENT.V	PERSON.V
UGRAD.V	STUDENT.V
GRAD.V	STUDENT.V
INSTRUCTOR.V	NONSTUDENT.V
ADMIN.V	NONSTUDENT.V

Figure 4-1: Personnel hierarchy relation

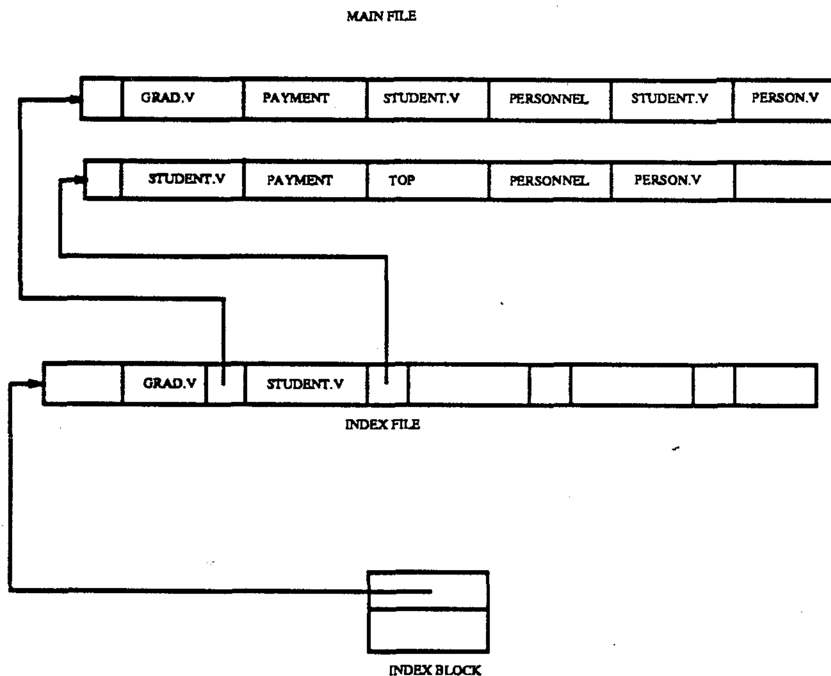

Figure 4-2: Indexed file for hierarchy relations

As seen in the figure we also need to store the hierarchy name in the record. This is because v-entity types may participate in different hierarchies and have parent or child v-entity types for each of these hierarchies. For example, in Fig 4-2, GRAD.V's parent v-entity type in the payment hierarchy is STUDENT.V while its parent v-entity types in the personnel hierarchy are

STUDENT.V and PERSON.V. Although we require more storage to replicate some information[14] this is justified because speed and efficiency in access is a primary goal of SQUIREL and secondly we envision the continued fall in the cost of memory. With each hierarchy relation represented by the structure just defined we will be able to do an efficient search and access of the hierarchy.

4.1.1. System Hierarchy Table--SYSHIER

The system hierarchy table (SYSHIER) is a two attribute table that contains the names of all hierarchy relations (**Name**) and the v-entity types that participate in each hierarchy (**Participant**). There is a one to many mapping from hierarchy relation name to v-entity types as one hierarchy has many v-entity types as its components. The SYSHIER is not directly accessible by the user and is only updated when hierarchy relations are updated by any of SQUIREL's constructs.

SYSHIER is used for *indexing*. Several SQUIREL constructs will need to use the SYSHIER to find the hierarchy relations of v-entity types. Without the SYSHIER, a linear search of all hierarchy relations would have to be performed in order to find the needed hierarchy relations.

An efficient way to implement the SYSHIER is to make it a hash table, as shown in Fig 4-3. Using this scheme, a very fast access to the v-entity types in a hierarchy can be achieved. Recall that v-entity types are not physically stored so the access is to the definition of these v-entity types which are stored in system tables[15] Using these definitions we will be able to obtain the base entity types of the v-entity types and perform any needed updates to the base entity types.

---

[14]STUDENT.V's parent v-entity type for the personnel hierarchy can be found in the GRAD.V record as well as in the STUDENT.V record.

[15]DBMSs have system tables that store v-entity type definitions, i.e., the CREATE VIEW command as shown on page 26. This gives us the structure of the v-entity type and tells us its attributes as well as its base entity types.

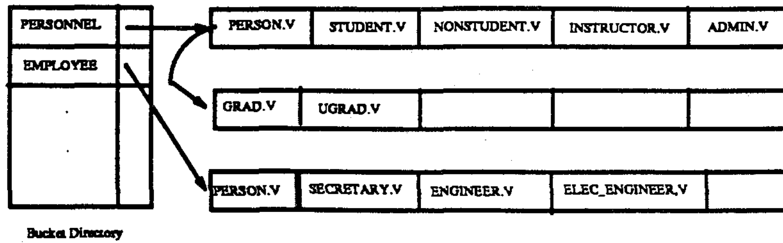

Figure 4-3: SYSHIER relation

4.1.2. Category/Partition in Hierarchies

The *partition* is a virtual attribute attached to v-entity types. V-entity types can participate in different hierarchies and for each hierarchy it is assigned a unique partition. A partition can be looked upon as a pointer to a virtual table which stores the names of v-entity types that this v-entity type subsumes. For example, in the PERSONNEL hierarchy the PERSON.V v-entity type might have STATUS as its partition name, shown in Fig 4-4. This implies that a person's STATUS is either a STUDENT or NONSTUDENT (these are the subsumed v-entity types of PERSON.V) in the PERSONNEL HIERARCHY (shown in Fig 4-1)

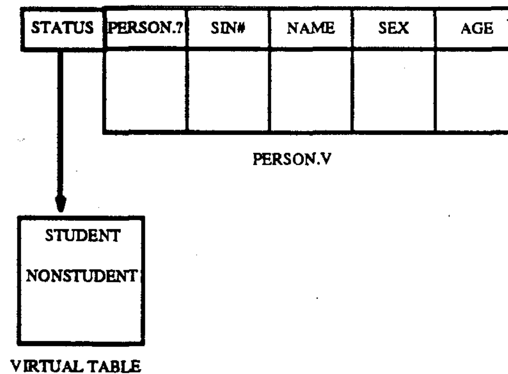

Figure 4-4: Implementation of PARTITION attribute

Since there is one partition for each hierarchy a v-entity type participates in, the definition of the v-entity type will be dynamically changed every time a v-entity type is inserted into or deleted from a hierarchy. Therefore, the system tables that store the v-entity type definitions must record these changes.

The partition is a virtual attribute and for any v-entity type it can be realized by the following query:

```
SELECT SUB FROM <hierarchy name>.HIERARCHY
 WHERE SUP  =   <view name>;
```

This query finds the subsumed v-entity types of a given v-entity type.

The *category* is much easier to implement because it is really an alias for the SUB attribute of a hierarchy. Therefore, we only need to reserve an extra block of storage next to the SUB attribute in each hierarchy relation to store the category attribute name.

The category and partition attributes are **optional** and the user need not give either, when defining a hierarchy or inserting a v-entity type into a hierarchy. If no category for a hierarchy is given, a default name of <name>.CATEGORY will be used for category. If partition is not specified for a v-entity type when it is inserted into a hierarchy, the default name for the partition will be <name>.PARTITION. In both of the above cases <name> is the unique hierarchy name which refers to the hierarchy in question.

4.1.3. Procedure Table

SQUIREL requires a procedure table to store the names of procedures it uses. An interface to control the communication between SQUIREL and the procedures is also needed. The interface will control the use of procedures, including the automatic invocation of procedures when the base entity types to which they are attached are referenced.

To implement procedural attachment to base entity types, the system table which stores the definition of base entity types also includes the names of procedures attached to the base entity types. The procedure table itself can be efficiently implemented as a base entity type while the interface is the algorithm for the trigger construct given in appendix B.

The use of SQUIREL's features and other implementation issues will now be provided.

4.2. Classification

The classifiers (type and instance) are essential to the implementation of the hierarchy because almost all the hierarchy constructs need to use them in order to be effectively processed. Both the type and instance level classification algorithms are given in appendix B and here we provide a brief description of the classifiers in use.

The classifier in SQUIREL works on this principle: a v-entity type (A) subsumes another v-entity type (B) if all of A's attributes subsume B's and if the v-entity types which subsume A also subsume B.

Subsumption determines a partial order on v-entity types because of the following properties.

1. Transitive : if A subsumes B and B subsumes C then A subsumes C.
2. Reflexive : A subsumes A.
3. Anti-symmetric: if A subsumes B, B does not subsume A.

We note that the classifier can be invoked either at the type level (v-entity type, i.e. view names) or at the instance level (entities, tuples) but for both of these the same principle mentioned above applies.

4.2.1. Type Level Classification

We use the classifier to build the hierarchy. Given a set of base entity types, the user might define several v-entity types. He/She may want to organize the v-entity types into a structure that allows him/her to manage them easily. The structure that SQUIREL provides is the hierarchy and it can be built either top-down or bottom-up. This implies that when we insert v-entity types into a hierarchy no ordering, (parent or child first) is needed.

Given the following v-entity types:

```
STUDENT.V     (name,sin#,stud#,sex,age,Dept,G.P.A,
               startdate)

GRAD.V        (name,sin#,stud#,sex,age,Dept,G.P.A,
               startdate,Last_degree)
```

```
UGRAD.V        (name,sin#,stud#,sex,age,Dept,G.P.A,
                startdate,Major)


PERSON.V       (name,sin#,sex)

SECRETARY.V    (name,sin#,sex,typing_speed)

ENGINEER.V     (name,sin#,sex,Degree)

ELECT_ENGIN.V(name,sin#,sex,Degree,Accreditation)
```

We can build a TEMP.HIERARCHY as follows:

```
CREATE HIERARCHY TEMP;

INSERT INTO TEMP.HIERARCHY
V-ENTITY = STUDENT.V;
```

The hierarchy now has only one component and is shown in Fig 4-5.

STUDENT.V

Figure 4-5: Temp.Hierarchy

We could then insert v-entity types into the hierarchy in the following way:

```
INSERT INTO TEMP.HIERARCHY
V-ENTITY =  GRAD.V,
V-ENTITY =  UGRAD.V;
```

The updated hierarchy is shown in Fig 4-6.

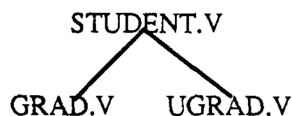

Figure 4-6: Temp.Hierarchy

We have been entering the v-entity types in a top-down fashion.

In order to insert v-entity types using a bottom-up method we do:

```
DROP HIERARCHY TEMP;

CREATE HIERARCHY TEMP;

INSERT INTO TEMP.HIERARCHY
V-ENTITY =  ELEC_ENGIN.V,
V-ENTITY =  SECRETARY.V;
```

The new hierarchy is shown in Fig 4-7.

ELEC_ENGINEER.V SECRETARY.V

Figure 4-7: Temp.Hierarchy

We now enter the higher level v-entity type:

```
INSERT INTO TEMP.HIERARCHY
V-ENTITY =  PERSON.V;
```

The updated hierarchy is shown in Fig 4-8

PERSON.V

ELEC_ENGINEER.V SECRETARY.V

Figure 4-8: Temp.Hierarchy

We now enter the middle v-entity type:

```
INSERT INTO TEMP.HIERARCHY
V-ENTITY =  ENGINEER.V
```

The updated hierarchy is shown in Fig 4-9.

PERSON.V

ENGINEER.V SECRETARY.V

ELEC_ENGINEER.V

Figure 4-9: Temp.Hierarchy

An example of how the type level classifier works for the TEMP hierarchy, in the state as shown in Fig 4-8 and for the insertion below follows.

```
INSERT INTO TEMP.HIERARCHY
V-ENTITY =  ENGINEER.V;
```

When such a SQUIREL command is typed. The classifier will do the following:

1. The newly entered v-entity type is ENGINEER.V and the hierarchy is TEMP.HIERARCHY.
2. Get the attributes of PERSON.V, SECRETARY.V and ELEC_ENGIN.V.
3. Compare the attributes and classify:
   a. Start from the top level v-entity type in TEMP.HIERARCHY. Check the PERSON.V v-entity type. On checking the attributes we find that it subsumes ENGINEER.V.
   b. Go down to the next level and check the ELEC_ENGIN.V v-entity type. Again on checking the attributes we find that ENGINEER.V subsumes ELEC_ENGIN.V.
   c. Check the SECRETARY.V v-entity type. There are contradictory attributes (typing speed) and (degree) so no link is needed.
4. From the above the PERSON.V - ELEC_ENGIN.V link is removed and ENGINEER.V is placed between PERSON.V and ELEC_ENGIN.V, i.e., (PERSON.V ELEC_ENGIN.V) is removed from TEMP.HIERARCHY while (ENGINEER.V, PERSON.V) and (ELEC_ENGIN.V, ENGINEER.V) are added to the TEMP.HIERARCHY relation.

4.2.2. Instance Level Classification

The iclassifier (instance level classifier) follows the same principle as the type level classifier but differs in its function. It is used to find out where entities are to be placed in a given hierarchy. The iclassifier will find the v-entity type which holds the entity that is being inserted. The entity attribute values are then inserted into the base entity types of the v-entity type found.

The iclassifier will find the subsuming v-entity types of the ones selected and discard the ones that are subsumed by some other in the set. This will take care of the situation where the parent (A.V) of a *leaf* v-entity type (E.V) is the parent of another v-entity type (B.V) and B.V also happens to be the parent of another leaf v-entity type (C.V and D.V). This is illustrated in Fig 4-10. Only key and unique attributes are shown attached to the v-entity types. B.V, C.V, D.V and E.V will inherit all attributes of their parents.

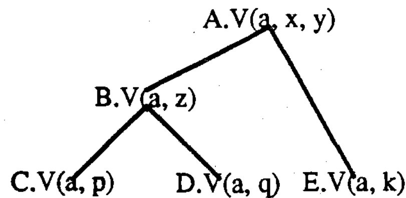

Figure 4-10: EX hierarchy

Assume we perform the following insertion:

```
INSERT INTO EX.HIERARCHY
 VALUES(a = 'john', x = 'tall');
```

Using the instance level classifier:

1. The leaf v-entity types C.V, D.V and E.V are returned.
2. C.V has a, x, y, z, p. D.V has a, x, y, z, q. E.V has a, x, y and k.
3. Then A.V and B.V are returned since they are the parent v-entity types of C.V, D.V and E.V.
4. Find applicable v-entity types:
   a. B.V is discarded because it is subsumed by A.V.
   b. Now only the v-entity type A.V is left, and the entity will be inserted into the base entity types of A.V.

When we are dealing with entities (instances) the iclassifier works bottom up in the hierarchy to obtain the required v-entity type in which to place the entity.

Using our PERSONNEL hierarchy shown in Fig 4-1, an insertion which will use the iclassifier is:

```
INSERT INTO PERSONNEL.HIERARCHY
 VALUES(Name    = 'John Smith', Curr_Work = 'Research')
        Office  = 'LB1233');
```

When the above SQUIREL command is issued, the iclassifier will:

1. Get all "leaf" entity types (INSTRUCTOR.V, GRAD.V, UGRAD.V, ADMIN.V).
2. Obtain the attributes of GRAD.V, INSTRUCTOR.V, UGRAD.V and ADMIN.V.
3. Only INSTRUCTOR.V is selected because it has the 'Curr_Work' attribute.

4. Since it is the only one selected, the entity is inserted at INSTRUCTOR.V.

Note that:

1. the insertion of the entity is **not** into the v-entity type but into the underlying P-relations of the base entity types of the v-entity type. The underlying base entity types can be found using system defined tables.
2. the key attribute must be given in the **insert** command.

4.3. Inheritance

The basic mechanism for inheritance is the *view*, a powerful construct in relational databases. There is a symbiotic relation between views and hierarchies. By using views, the inheritance of attributes which naturally proceeds through the links in a hierarchy is made trivial. Since views are virtual relations it is easy to define views with all needed attributes specified.

An example of a hierarchy is:

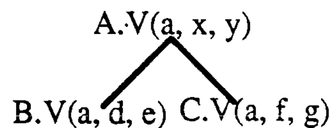

Figure 4-11: An Example of a Hierarchy

Here A.V, B.V and C.V are v-entity types. Only the unique and key attributes attached to the v-entity types are shown in Fig 4-11. The links in the hierarchy allow B.V and C.V to inherit the attributes x and y from A.

The v-entity type B.V is created as follows:

```
CREATE VIEW B.V(a, x, y, d, e)
 AS SELECT  A.a,x,y,d,e
    FROM    A,B;
```

We see that the inheritance of attributes is encompassed in the user's definition of a view and the user can easily define v-entity types, all specifically suited to his/her application.

4.4. Default Entity Types

To specify defaults in SQUIREL, a **default entity type** could be implemented. This will be known as <entity.name>.default[16]. The default entity type will be first created whenever a new entity type is defined and will be updated whenever new P-relations are added to the entity type. It gives the user the ability to express default values for most of the attributes[17] of that entity type. We will only have one set of default values for each P-relation and if the user does not specify attribute values when inserting entities into the database then those default values will be used.

An efficient way to implement default value assignments is to reserve the first storage block of the P-relations of each base entity type to store the set of default values. An example is shown in Fig 4-12.

STUDENT.?	DEPT	SIN#	STUD#	STATUS	GPA	START
	GENERAL					

FIRST STORAGE BLOCK RESERVED

Figure 4-12: P-relation of STUDENT entity type

Default values are not assigned to the v-entity types but to the base entity types in which the v-entity types are defined. V-entity types will implicitly inherit the default values from its base entity types. Therefore the default entity type specifies defaults for the attributes of a v-entity type indirectly through the base entity types.

In the example given, GENERAL is the department attribute given to a student whose DEPT field is not specified. This is a reasonable default because most new students have not decided on a major when they first enter university.

---

[16]Where entity.name is the name given to an entity type.

[17]Key attributes cannot have default values.

4.5. Procedure Construction

SQUIREL's **playback** procedure is described and this will give a general idea about how any procedure can be constructed and used. The playback procedure determines the classification path of the type level classifier. It requires two arguments which are the hierarchy relation name and the v-entity type whose path needs to be found. The playback procedure has its own entity type called PLAYBACK.ARG. PLAYBACK.ARG has two attributes, HNAME (hierarchy name) and ETNAME (v-entity type). These attributes are updated whenever the classifier is used and PLAYBACK.ARG will always store the name of the hierarchy and v-entity type currently referenced. To illustrate this, if the classifier was last used to place the v-entity type (GRAD.V) in the PERSONNEL hierarchy. The attributes in PLAYBACK.ARG would be HNAME = PERSONNEL, ETNAME = GRAD.V.

If the user wants to know the classification path of a previously referenced v-entity type, the PLAYBACK.ARG entity type can be updated as in:

```
UPDATE PLAYBACK.ARG
SET ETNAME = 'INSTRUCTOR.V';
```

PLAYBACK.ARG will then contain the new v-entity type name whose path the user wishes to know. The hierarchy to be used (HNAME) can be similarly updated. Note that as PLAYBACK.ARG will always have only **one** entity (only one tuple in PLAYBACK.ARG), any update to PLAYBACK.ARG will be performed on this entity.

Using embedded SQL statements in a PL/1 environment as given in [DATE 86], the playback rule is shown in appendix B.13.

Chapter 5

COMPARISON WITH KEE AND CENTAUR

We will first describe the KEE system in some detail and then describe SQUIREL constructs which provide similar capabilities.

5.1. The KEE System

KEE (Knowledge Engineering Environment) [FIKE 85], one of the most widely used programming environments for developing sophisticated expert systems is a frame based, object oriented system developed at Intellicorp. It is a frame system combined with facilities for production rule based reasoning.

In KEE there are *subclass* and *member* links. Membership refers to a particular instance of a class whereas subclass refers to a smaller class (a subset). For example, given a class frame STUDENT, a GRAD.STUDENT frame is a subclass of STUDENT while a STUDENT1 frame (name='John Doe') is a member of STUDENT because it represents a unique student.

5.2. Description of KEE Frames

Each frame has slots which are 'attribute descriptors' of the frame. These slots are of two types:

1. OWN SLOTS: describes attributes of the instance or class represented by the frame, e.g., TOPSTUDENT - representing the student with the highest G.P.A in the STUDENT CLASS frame shown in Fig 5-1.
2. MEMBER SLOTS: describes attributes of each member of a class, e.g., NAME - an attribute of each member of the STUDENT frame.

Own slots represent an actual attribute and contains a value specific to the frame while a member

slot represents an attribute but does not contain actual values. Member slots are used to provide a structural framework because these are the slots that are inherited along the *subclass* and *member* links.

Attached to these slots are 'sets of properties' called facets which provide a 'description' of the slots, e.g., *cardinality* (number of values), *ValueClass* (type of values) and so on. Facets contribute to the deductive abilities of frames by restricting the number and type of values slots can take.

KEE's value class facet can also be a boolean combination of class descriptors, i.e., predicate logic is used in combination with available class descriptors.

An example from [FIKE 85] is:

```
MEMBER SLOT:   FATHER
VALUECLASS :  (INTERSECTION MEN
                (UNION DOCTORS LAWYERS)
                (NOT.ONE.OF FRED))
```

Here the FATHER SLOT is designated to be a man who is a doctor or a lawyer but is not Fred. MEN, DOCTORS and LAWYERS are class frames and FRED is an instance frame.

Examples of KEE's frame structure are given in Fig 5-1, Fig 5-2, Fig 5-3 and Fig 5-4.

5.3. Capabilities

The KEE system is a powerful knowledge representation system and its features can be divided into three parts.

1. Inheritance mechanism which includes *cardinality/value class* reasoning.
2. An *object oriented* approach.
3. Its *rule system*.

```
Unit:          STUDENT        in knowledge base SCHOOL
  Superclasses:         PERSON
  Subclasses:           GRAD.STUDENT, UGRAD.STUDENT
  Member:               CLASSES.OF.PHYSICAL.OBJECTS

MemberSlot:             NAME from PHYSICAL.OBJECTS
    ValueClass:         CHARACTER
    Cardinality Min:        1
    Cardinality Max:        1
    Comment:                'Full Name'
    Values:                 Unknown

MemberSlot:             CHECK from STUDENT
    Inheritance:            METHOD
    ValueClass:             METHODS
    Cardinality Min:        1
    Cardinality Max:        1
    Comment:                'A method for diagnosing course problems'
    Values:                 COURSE.PROBLEM.FUNCTION
MemberSlot:             PRESENT.SCHOOL from PHYSICAL.OBJECTS
    Cardinality Min:        1
    Cardinality Max:        1
    Values:                 Unknown

MemberSlot:             GPA from PHYSICAL.OBJECTS
    Cardinality Min:        1
    Cardinality Max:        1
    Values:                 Unknown
MemberSlot:             TOTAL.CREDITS from PHYSICAL.OBJECTS
    Cardinality Min:        1
    Cardinality Max:        1
    Values:                 Unknown
OwnSlot:                TOPSTUDENT from CLASS.OF.PHYSICAL.OBJECTS
    ValueClass:             STUDENT
    Cardinality Min:        1
    Cardinality Max:        1
    Comment:                'The student with the highest GPA'
    Values:                 Unknown
```

Figure 5-1: STUDENT Frame

5.3.1. Inheritance

When a frame A is a *subclass* of another frame B, **member** slots of frame B will be inherited as **member** slots of frame A. Thus for a hierarchy of class-subclass relations, the member slots need only be defined at the top level frame and will then be inherited by all its subclass frames and in turn by the subclass frames of these subclass frames.

Unit: GRAD.STUDENT in knowledge base SCHOOL

Superclasses: STUDENT

Subclasses: ....

Member: ....

MemberSlot: PASTDEGREE from PHYSICAL.OBJECTS

ValueClass: ....

Cardinality Min: 1

Cardinality Max: 10

Values: Unknown

MemberSlot: PUBLICATIONS from PHYSICAL.OBJECTS

ValueClass:

Cardinality Min: 0

AvUnits: AV.STUDENT

.

.

Figure 5-2: GRAD.STUDENT Frame

Unit: SFU.GRAD.STUDENT in knowledge base SCHOOL

Superclasses: STUDENT, GRAD.STUDENT

Subclasses: ....

Member: ....

MemberSlot: NAME FROM STUDENT

ValueClass: ....

Cardinality Min: 1

Cardinality Max: 1

Values: Unknown

MemberSlot: PUBLICATIONS from GRAD.STUDENT

ValueClass:

Cardinallity Min: 1

.

.

Figure 5-3: SFU.GRAD.STUDENT Frame

When a frame A is a *member* of another frame B, **member** slots of B will be inherited as **own** slots of A. Frame A will be a particular instance of frame B and thus the member slots of B will now be own slots of A and have specific values attached to them. Own slots are not inherited because they represent specific attributes with corresponding values for a particular frame.

```
Unit: STUDENT1 in knowledge base SCHOOL
   Member: GRAD.STUDENT

OwnSlot:      FATHER
   ValueClass:     MEN (UNION DOCTORS LAWYERS)
                       (NOT.ONE.OF.FRED)
                   AGENTS
   Cardinality Max: 1
   Values:         PAUL

                   .
                   .
                   .
OwnSlot:      CHECK from STUDENT
   ValueClass:     METHODS
   Cardinality Min: 1
   Cardinality Max: 1
   Comment:        'A method for diagnosing course problems'
   Values:         COURSE.PROBLEM.FUNCTION

OwnSlot:      PUBLICATIONS from GRAD.STUDENT
   Cardinality Min: 1
   Comment:        'Papers published'
   Values:         Unknown
```

Figure 5-4: STUDENT1 Frame

Notice that own slots of classes, e.g., TOPSTUDENT in the class frame STUDENT, are a form of meta-knowledge. These own slots provide information about the whole class and have to perform some form of deduction. In the case of the TOPSTUDENT own slot for the class of students, every student's G.P.A has to be checked to obtain the student with the highest G.P.A.

5.3.2. Cardinality/Value Class Reasoning

When we enter values for own slots, if we enter values that do not satisfy its **value class** or if we exceed/fall short of the **cardinality** requirements, an error is reported and the object is not be created.

An example could be if we made the following mistakes:

1. Enter an INTEGER value instead of a CHARACTER.
2. If CARDINALITY MIN: 2 and we only enter one value.

This is another one of KEE's inferencing abilities. It allows the system to check on individual values entered to ensure correctness of data. The cardinality also enables a criteria check on the number of values a slot can take. For example, *sex* must have only one value so we set

CARDINALITY.MIN:1 and CARDINALITY.MAX:1 for the slot *sex*. The value class facet on slots is a restriction mechanism to ensure that the slot will contain reasonable or user controlled values.

5.3.3. Object Oriented System

Object oriented systems are systems which have the procedures or functions applicable to an object attached to the object. In KEE, the frame is an object and the procedures which are applicable to that object are also stored in the frame structure as slot values. These procedures or functions are called **methods.** For example, the CHECK slot in the STUDENT frame (Fig 5-1) has the method COURSE.PROBLEM.FUNCTION. The methods are invoked by the use of messages. Messages can be sent directly by the user or by other frames.

5.3.4. KEE Rules

Another interesting aspect of KEE is its use of frames for the management and representation of production rules. Rules in KEE are written in an "IF-THEN" format and provide the system with the ability to reason from available facts in the system.

Rules are of the following form:

```
(IF    <condition>
       <condition>
         .
 THEN  <conclusion>
       <conclusion>
         .
 DO    <action>
       <action>
         .          )
```

In KEE a production rule is also represented as a frame and an example of a rule class unit is shown in Fig 6-5. Note that the STUDENT.TYPE.RULES has three rules. The OUT2.RULE one of the rules in STUDENT.TYPE.RULE states that to be an outstanding student you must be in the class GRAD.STUDENT, be in the class NORMAL.PROGRESS and you must have published at least one paper.

```
Unit:     STUDENT.TYPE.RULES
           in knowledge base RULESSYSTEM2

      Superclasses: RULES
      Member:            RULE.CLASSES

(Rules (NORMAL.RULE

            (IF ((?X IS IN CLASS GRAD.STUDENT)

                AND

                ((GEQ (THE NO.COURSES OF ?X) #))

                 THEN

                 (?X IS IN NORMAL.PROGRESS))))

        (OUT1.RULE
            (IF ((?X IS IN CLASS GRAD.STUDENT)

                 AND

                 ((GREATERP (THE NO.COURSES OF ?X) 3))
                 THEN

                 (?X IS IN OUTSTANDING.PROGRESS)))

         (OUT2.RULE
            (IF ((?X IS IN CLASS GRAD.STUDENT)
                 AND
                 (?X IS IN CLASS NORMAL.PROGRESS)
                  AND
                 ((GREATERP (THE PUBLICATIONS OF ?X) 0)))
                  THEN
                   (?X IS IN OUTSTANDING.STUDENT)
                  DO
                 (SEND.CONGRATS
                      (GET.VALUE ?X NAME)
                       (GET.VALUE ?X ADDRESS))))
```

Figure 5-5: STUDENT TYPE RULES

5.4. Use and Concepts of Rulesystem2

KEE's Rulesystem2 knowledge base is a powerful tool and some of its main features are elaborated upon here.

5.4.1. Classification in KEE

KEE supports classification[18]to determine which classes a given instance belongs to. In KEE, the member slots of a frame are the necessary while the **production rules** are the **sufficient** conditions. KEE uses a successive refinement strategy [WOODS 83] to classify instances, where each (sub) class description include rules that indicate whether members of some superclass can be members of the (sub) class.

For example, consider the OUT2.RULE in the STUDENT.TYPE.RULES rule class unit. The first condition is a check for membership in GRAD.STUDENT (superclass) while succeeding conditions (which are, taking three courses and having more than one publication) specify the local conditions for membership in the OUTSTANDING.STUDENT frame (subclass).

5.4.2. Control Strategy

KEE provides several control strategies

1. Backward chaining.
2. Forward chaining.
3. Hypothetical reasoning: values of slots can be changed 'on the fly' to simulate different outcomes.

The default strategy used is backward chaining, but the user can modify this approach by using the graphic displays or active value facets. When KEE is running it highlights the active value units and TRUE or FALSE state of premises and conclusions. The user can change the value of attributes while the system is running allowing the user to modify the control strategy interactively (hypothetical reasoning).

It also supports graphic-based interfaces, facilities for graphic-based simulations, multiple forms of inheritance (member, own slots) and demons. KEE provides the user with a powerful mechanism to build and use production rules. With its graphic facilities it makes it very easy to

[18]This is not the same as the more difficult but general classification problem of determining where a class belongs in a given class-subclass taxonomy.

define, connect and change the frame structure. This leads to the user being able to develop an initial knowledge base very quickly.

5.5. SQUIREL's Answer

In SQUIREL we do away with the idea of own slots of classes because SQUIREL already provides powerful constructs to obtain this needed information.

In Fig 5-1 the own slots for the class frame STUDENT is TOPSTUDENT. We assume that STUDENT is an entity type with G.P.A as one of its attributes we would then be able to find the TOPSTUDENT by this SQUIREL query:

```
SELECT NAME, MAX(G.P.A) from STUDENT;
```

We notice that when it comes to a frame representing an individual instance, e.g., STUDENT1, all the slots are own slots. STUDENT1 corresponds to an entity in an entity type. Here the attributes of the entity type will have specific values for individual tuples (entities).

SQUIREL does incorporate most of the reasoning facilities of KEE's frame structure which include:

1. Inheritance of **member** slots.

   Views (v-entity types) are used in SQUIREL to explicitly show the inheritance of attributes. The hierarchy structures the v-entity types into a usable form.

2. Constraint Checking Procedures (ValueClass and Cardinality)

SQL already provides some constraint checking facilities corresponding to the value class facet of KEE. Attributes in SQUIREL are stated to be of type number, character or date. They will also be able to hold NULL values.

In terms of boolean attributes we could have MEN, DOCTORS and LAWYERS as entity types. To find the FATHER's name as in the example on page 60, we could have a v-entity type to check the name of fathers defined as:

```
CREATE VIEW FATHER.CHECK.V
 AS     SELECT        Name
        FROM          MEN, DOCTORS,     LAWYERS
        WHERE         MEN.Name       =  DOCTORS.Name
        AND           DOCTORS.Name =    LAWYERS.Name
        AND           MEN.Name      !=  'FRED';
```

The v-entity type FATHER.CHECK.V would now contain all the acceptable values of FATHER name. Though SQUIREL does not provide an explicit connection of the Name attribute to FATHER.CHECK.V, the user can check values entered to Name attribute of FATHER against the FATHER.CHECK v-entity type.

As for cardinality constraints, the RM/T structure is set up to allow entities to have only one value for each attribute. If multiple values are required for an attribute a characteristic entity type is formed. Though the control of this is left to the user, the structure explicitly shows the user if only one value is needed.

It is felt that this is sufficient for the user to constrain the number of values of attributes to individual entities.

5.5.1. Classification

SQUIREL supports both types of classification in its hierarchy. Member slots in CLASS frames correspond to the relational schema of the entity types or v-entity types.

The GRAD.STUDENT frame could be defined as:

```
CREATE VIEW GRAD.STUDENT.V
 AS SELECT  Name,       PastDegree,Publications,
            CourinProg,LastDegree
    FROM    STUDENT,    GRAD;
```

The frame STUDENT1 represents an entity in SQUIREL which would be a tuple in the GRAD.STUDENT.V v-entity type.

An example of a SQUIREL hierarchy is given in Fig 6-6. Only key and unique attributes are explicitly attached to the v-entity types.

In KEE when the STUDENT1 frame is formed, the classification of which classes this item

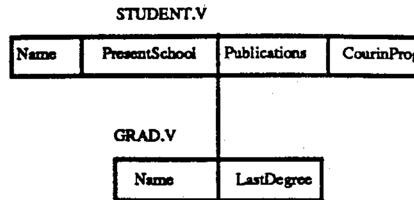

Figure 5-6: GRADUATE.HIERARCHY

belongs to is performed. STUDENT.TYPE.RULES is used to show that STUDENT1 is in the class OUTSTANDING.STUDENT.

Assume we insert a new student into the GRADUATE hierarchy as follows:

```
INSERT INTO GRADUATE.HIERARCHY
VALUES(Name = 'Tom',Publications = '2',CourinProg = '3');
```

In SQUIREL the classifier will place this entity under the v-entity type GRAD.STUDENT.V and enter the data into the entity types STUDENT and GRAD.

Suppose we had defined OUTSTANDING.STUDENT as:

```
CREATE VIEW OUTSTANDING.STUDENT.V
 AS SELECT Name,PastDegree,Publications,PresentSchool
    FROM   STUDENT, GRAD
    WHERE  CourinProg    = '3'
    AND    Publications >= '1';
```

If the inserted entity satisfies the conditions on this v-entity type (which it does), OUTSTANDING.STUDENT.V will also be updated. We see that the OUTSTANDING.STUDENT.V v-entity type in SQUIREL is performing a similar task as the OUT2.RULE in KEE. The classification algorithm inserts the new entity into the entity types GRAD and STUDENT while the v-entity type (OUTSTANDING.STUDENT.V) updates itself because the new entity satisfies the conditions of being in OUTSTANDING.STUDENT.V.

The inheritance of the relational schema (member slots in KEE/ attributes in SQUIREL) is implicitly handled when the user defines the view. We see that the attributes as well as the entity types from which the attributes are taken are specified in the definition.

5.5.2. SQUIREL Procedure Management

SQUIREL does not have demons. A 'pseudo' demon (implicit in its structure) is invoked when a new entity is entered. The classification of that entity is performed because of an insertion. We do not have rules, although views can be looked upon as a form of rules. The integrity constraints applied to views when they are defined, e.g, WHERE Publications >= 3, has been shown to perform the same function as the OUT2.RULE in KEE (shown in Fig 5-4).

SQUIREL provides a construct (TRIGGER) which can be used to trigger procedures. The procedures themselves are defined in a high level programming language like PL/1, Pascal or Ada and it is certainly conceivable that any rule defined in KEE can be similarly defined in a high level language. Most implementations of SQL already provide an interface with several high level programming languages. These high level languages can easily provide the facilities for defining procedures, and with the addition of the trigger in SQUIREL, we are able to emulate most of the capabilities of KEE's rule knowledge base.

We are keeping SQUIREL clean and uncluttered, only providing the user with a triggering mechanism. We feel that the advantages that various features of SQUIREL provides to the users outweigh the perceived overhead of having to learn the language. Most rules are implemented at a high level by the database administrator. Normal users seldom if ever define their own rules. This being the case, it is assumed that the database administrator will have no problem in learning a high level language. The use of a trigger is simple and straightforward and SQUIREL suffers little from complication or complexity.

5.6. COMPARISON WITH CENTAUR

The second AI system to be compared with SQUIREL is CENTAUR [AIKINS 84] which is a consultation system that produces an interpretation of data and a diagnosis based on a set of test results. It uses a knowledge representation scheme that is a combination of frames and rules represented in a data structure called a prototype.

A prototype contains two kinds of information.

1. COMPONENTS: domain specific information which represents characteristics of the prototype.
2. SLOTS: domain independent information which specify information used in running the system.

Components may themselves have slots associated with them, including a special RULE slot that links the component to rules which determine values of the component.

An example of a PROTOTYPE is:

```
PROTOTYPE:        Obstructive Airways Disease (OAD)

  COMPONENTS
                              TOTAL LUNG CAPACITY
                                Plausible Values: > 100
                                Importance: 4

                              REVERSIBILITY
                                Rules: 1,3,5
                                Importance: 0

  CONTROL INFORMATION         Deduce the degree of OAD
                              Deduce the subtype of OAD

  ACTION INFORMATION          Print the findings
```

There are four special prototype slots which are control slots used to control the consultation.

These are:

1. CONTROL INFORMATION: indicates how to proceed in order to instantiate the prototype (asks the user for data to fill components).
2. IF-CONFIRMED and IF-DISPROVED: these slots determine what to do after the clauses in the CONTROL slot have been filled and the prototype is either confirmed or disproved.
3. ACTION INFORMATION: the premise in this slot is "if the system has completed its selection of confirmed prototypes and this prototype is confirmed". When the premise is satisfied, the action is performed which is to generate summary statements or print data interpretations.

CENTAUR is specifically used in conjunction with PUFF [KUNZ 78] which is a pulmonary disease diagnostic system, thus its associated rules deal with diseases. Rules are grouped into four sets according to their functions. They consists of one or more **premise** clauses that refer to values for components and one or more **action** clauses that make conclusions about values for components.

The sets of rules are:

1. patient rules: dealing with the patient.
2. summary rules: summarizes results of tests.
3. triggering rules: check values of components and suggest general disease categories.
4. refinement rules: refine preliminary diagnosis and produces a final diagnosis.

An example of a rule is:

```
RULE013
 PREMISE:   ($OR(LESSP* (VAL1 CNTXT MMF) 20)
                (GREATERP* (VAL1 CNTXT FVC) 80))

 ACTION:    (CONCLUDE CNTXT DEG<-MMF SEVERE TALLY 900)
```

This rule states that:

IF the MMF/MMP predicted ratio is less than 20 or
the FVC/FVC predicted ratio is greater than 80

THEN there is strong suggestive evidence (.9) that the degree
of obstructive airways disease as indicated by MMF is severe.

MMF - maximum midexpiratory flow
FVC - forced vital capacity

An example of a prototype network is given in Fig 5-7.

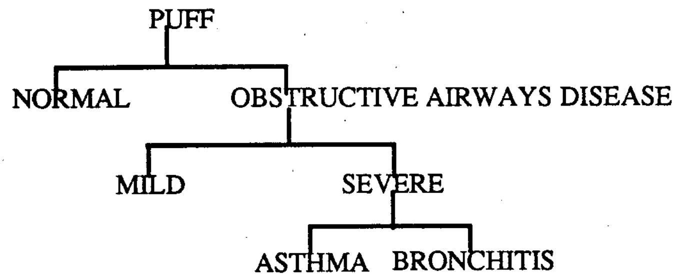

Figure 5-7: Prototype Network in CENTAUR

CENTAUR uses a hypothesis directed approach to problem solving. Each hypothesis is represented by a prototype and the goal of the system is to confirm that one or more prototypes in the network match the actual data.

The system begins by accepting data about a patient. This will trigger one or more prototypes which are ordered by how closely they match the data. The closest matching prototype is selected as the **current** prototype and the matching begins again. At any time there is only one current prototype. When a prototype is activated, essential data is asked of the user (what to ask is obtained from the CONTROL SLOT). The IF_CONFIRMED and IF-DISPROVED slots then determine what to do next.

An example of how CENTAUR works follows:

```
CURRENT PROTOTYPE :  PUFF
1) Patient Number
  097
2) Referral Diagnosis
  ASTHMA
3) Total lung capacity
  139
4) FVC
  82
5) MMF
  12

Triggered Prototypes: ASTHMA, CM 900
                      NORMAL, CM 500: Reason FVC 82
                      OAD,    CM 900: Reason MMF 12

MoreSpecific Prototype Chosen: NORMAL OAD
```

These prototypes are filled with the data values already known. Inconsistent values lead to the CM (certainty factor) of the prototypes being lowered. CENTAUR will continue prompting the user as follows:

```
Hypothesis list: (OAD 990) (NORMAL -699)

CURRENT PROTOTYPE: OAD
1)RV/TLC Observed-predicted
 25

Confirmed list: OAD PUFF
MoreSpecific Prototype chosen: SEVERE

CURRENT PROTOTYPE: SEVERE
12) FEV1
    42

Confirmed list SEVERE OAD PUFF
MoreSpecific Prototype Chosen: ASTHMA

CURRENT PROTOTYPE: ASTHMA
13) CRP
    20

Confirmed list :ASTHMA SEVERE OAD PUFF
```

- the user's answers are in italics
- CM - stands for the certainty factor accorded to each prototype
- MMF - maximal midexpiratory flow

- FVC - forced vital capacity
- FEV1 - forced expiratory volume at one minute
- TLC - total lung capacity
- RV - residual volume
- CRP - change in residual predilation

CENTAUR's hypothesis directed approach is basically a top down, backward chaining strategy of **classification.** The CONTROL-INFORMATION slot in a prototype prompts the user for information needed to confirm or disprove a prototype (hypothesis). The IF-CONFIRMED or IF-DISPROVED slots guide the system to proceed further with the consultation.

Rules associated with the components of a prototype are specific to that component. This leads to an object oriented design where specific components of a prototype will have specific rules associated with it.

5.7. Classification in SQUIREL

SQUIREL can support a hypothesis directed classification scheme similar to CENTAUR's in its hierarchy mechanism. We could have a hierarchy of these v-entity types as follows:

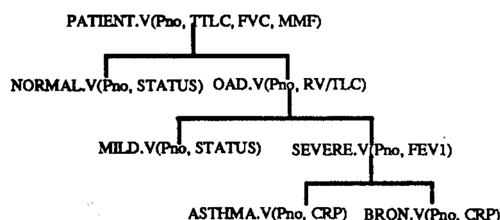

Figure 5-8: Air.Disease.Hierarchy

The v-entity type, ASTHMA.V is defined as follows:

```
CREATE VIEW ASTHMA.V
 AS SELECT PNO,TLC,FVC,REV,TLCCHANGE
    FROM   PATIENT,OAD,SEVERE,ASTHMA
    WHERE  TLC     < 140 AND FVC  > 5
    AND    FVC     < 20  AND MMF  < 15
    AND    RV/TLC  > 20  AND FEV1 < 50
    AND    CRP     > 10  AND CRP  < 25;
```

SQUIREL is able to classify a patient and diagnose the disease he/she might have. This is done when the user gives **all** information required to perform an accurate diagnosis. If all information is given, the determination of which v-entity types the new entity should be inserted into will be performed by SQUIREL's classifier.

An example of inserting a patient entity into the AIR.DISEASE.HIERARCHY follows:

```
INSERT INTO AIR.DISEASE.HIERARCHY
VALUES(PNO    = '792',TLC  = '139', FVC = '82', MMF ='12',
       RV/TLC = '25', FEV1 = '42',  CRP = '20');
```

Given the above insertion, SQUIREL's classifier will determine that the entity referenced by the patient number being 792 will be in the v-entity type ASTHMA.V.

One shortcoming of SQUIREL is that it is only able to classify an entity when all its relevant attributes are known. Unfortunately this does not usually occur in a consultation system which often requires interaction. CENTAUR uses a successive refinement strategy and also gives a step by step account of what it is doing. Reasons for choosing a prototype are also available, this being very important for consultation systems dealing with a person's health. CENTAUR is oriented towards being an expert consultant.

We feel that SQUIREL can also be developed to emulate a system like CENTAUR. To achieve this, rules similar to the ones controlling CENTAUR need to be defined.

1. We would need to know what basic information is always provided, e.g., PNO, TLC.
2. Rules to prompt the user for more information will be defined.
3. SQUIREL's classifier will interactively update the entity's status until the entity is fully classified.

As rules in SQUIREL are implemented in a high level language and triggered by changes in the database, we could develop rules to do a similar task as CENTAUR's prototype network.

An outline of what is to be done follows:

- We use the SQUIREL hierarchy defined above and rules defined in PL/1 to emulate the reasoning done in CENTAUR.
- Rules/programs are attached to each v-entity type found in the hierarchy. These

programs will give the present classification status of the entity in question and the reasons as to why the classification is chosen.

• The rules will then prompt the user for more information until the entity is fully classified.

Assume we do the following:

```
CREATE     OAD.V AS
 SELECT    PNO, TLC, FVC, MMF, RV/TLC
 FROM      PATIENT, OAD
 WHERE     TLC < 140 AND FVC < 90 AND MMF < 15
 TRIGGER   OAD.RULE;

INSERT INTO AIR.DISEASE.HIERARCHY
VALUES(PNO='792',TLC='139',FVC='82',MMF='12');
```

This would cause the OAD.V v-entity type to be updated and the OAD.RULE to be invoked. OAD.RULE is a program written in a high level programming language PL/1 and embedded SQL statements [DATE 86]. The pseudo code for this is given in the appendix C. Assuming we use the OAD.RULE and have the above insertion, the user will be interactively asked for more input as follows:

```
The reason this patient might have an OAD
      (OBSTRUCTIVE AIRWAYS DISEASE) is because
       THE TLC IS , 139
       THE MMF IS , 12
       THE FVC IS , 82
       Which makes the certainty that he has an
       OAD,  80%

The possible kinds of OAD for patient 792 are:
BRONCHITIS
ASTHMA

Could you please tell me 792's RV/TLC:_________
```

Entering the RV/TLC will trigger another rule until a concluding diagnosis such as BRONCHITIS is given. The user can use the PLAYBACK rule to find the path used by the classification algorithm to come up with this diagnosis.

This is done by issuing the following command:

```
TRIGGER PLAYBACK;
```

which will return the path of the most recent classification performed. This is shown in Fig 5-9 and tells us that we classified the inserted entity as a patient then as having an obstructive airways disease then as having a severe case of an obstructive airways disease.

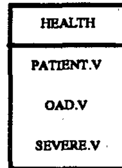

Figure 5-9: Classification Path for Air Disease Hierarchy

Most classification systems have the ability to give a reasoning path for their inferences. This is especially prevalent in expert systems for diagnosis, like MYCIN. We see that SQUIREL also provides this capability through its use of rules like OAD.RULE.

In our system, the AIR.DISEASE.HIERARCHY has the v-entity types PATIENT.V, NORMAL.V, OAD.V, MILD.V, SEVERE.V, ASTHMA.V and BRON.V. Each of these v-entity types will have a rule attached to it. This rule by convention will have <v-entity type>.RULE as its name and will be defined in a manner similar to the OAD.RULE.

The basic function of each rule is to:

- Determine certainty factors.
- Print further possible classifications.
- Prompt the user for more information.
- Update the v-entity type and hierarchy which will lead to other rules being invoked.

It is to be noted that **leaf** v-entity types will not need to be further classified and thus will not prompt the user for more information. Instead the 'leaf' v-entity types will have rules which give a concluding diagnosis. Thus we see that SQUIREL's classifier with the help of rules defined in a high level programming language can perform tasks similar to those that CENTAUR does.

Chapter 6
CONCLUSION

In this research, we have defined SQUIREL, a specification and query language for our modified RM/T relational database model. In the design of SQUIREL, we chose to extend the well-known query language SQL, so that it supports:

1. INHERITANCE: The hierarchy allows subclasses to inherit all attributes of its superclasses.
2. CLASSIFICATION: Both the general class-subclass and the more specialized class-instance classifications are provided.
3. DEFAULTS: Optional default assignments to base entity types allow the system to 'reason' about these entity types based on the users' underlying assumptions.
4. INTERNAL MANAGEMENT OF VIEWS: The hierarchy which consists of v-entity types provides a basis for views to be combined into one structure and managed by the system.

With SQUIREL, the user is able to define objects (v-entity types) in hierarchies allowing semantic information to be captured within the model. These hierarchies can be viewed as data structures and as such can be queried and modified through constructs provided by our language. SQUIREL has been designed to help the user as much as possible. For example, it is possible to add new entities to the hierarchy without being aware of all the existing entity types. SQUIREL also allows the user to define procedures so as to exercise more control over both the hierarchies and the database.

SQUIREL has been compared with KEE and has a general class-subclass classifier as well as KEE's class-instance classifier. Inheritance in both SQUIREL and KEE proceeds through the use

of hierarchies and they also both share a similar default concept. KEE has a powerful rule base and although SQUIREL does not have a rule base, it is able to define and use procedures written in a high level programming language.

The procedure mechanism of SQUIREL is also able to emulate CENTAUR's frame/rule prototype method of classification. In CENTAUR, the prototype directly invokes rules attached to its slots and in SQUIREL the **trigger** construct similarly invokes procedures attached to the v-entity types of SQUIREL's hierarchies. SQUIREL is not able to do a CENTAUR-like probability classification so we once again reiterate that it has not been our intention to incorporate a rule base in SQUIREL.

We have kept SQUIREL simple and have been able to retain the efficiency and robustness of database systems. With its added constructs, SQUIREL has been shown to compare favorably with two prominent AI systems.

6.1. Future Research

SQUIREL's design could be further extended to incorporate the concept of having a hierarchy of hierarchies, where several smaller hierarchies are combined to form a larger hierarchy. For example, we might want to combine a university personnel hierarchy and a university buildings hierarchy into a university hierarchy and in turn combine several university hierarchies into an institute of higher education hierarchy. This aggregation principle might either be incorporated into SQUIREL's current hierarchy structure or a new SQUIREL hierarchy could be developed, the reason being that aggregation of objects might be more suited to specialization relationships. Specialization relationships are used to define possible roles for higher level objects [ABITE 87]. For example, we can have the v-entity types: computing science students and honor students. These are specializations of the v-entity student and are possible roles taken by a student. We notice that the specializations are not disjoint, that is, a computing science student can also be an honor student which is not allowed in SQUIREL's current generalization hierarchy.

If the hierarchy is implemented based on the view facility in SQL, it might also be pertinent to consider incorporating semantics into the view updates. This involves understanding the meaning

and role of updates on views. For example, suppose we have a 'computing science graduate student' base entity type and two v-entity types as shown in Fig 6-1.

Name	Status	Volleyball
John	Thesis	Yes
Mary	Thesis	No
Jane	Course	No
Joe	Thesis	Yes
Tim	Course	Yes

COMPUTING SCIENCE GRADUATE STUDENTS

Name	Volleyball
John	Yes
Joe	Yes
Tim	Yes

VOLLEYBALL

Name	Status
John	Thesis
Mary	Thesis
Joe	Thesis

GRADUATING

Figure 6-1: Computing Science Graduate Students

The graduating v-entity type is used by the graduate program chairman and is a view of all computing science graduate students who are completing their thesis as a final requirement for their degree. The volleyball v-entity type is used by the volleyball captain and is a view of all computing science graduate students who are playing in the inter-mural competition. Now consider the situation where the volleyball captain deletes John from the volleyball v-entity type. Normally this does not mean that John is deleted from the computing science graduate student base entity type. On the other hand, if the graduate chairman deletes John from the graduating v-entity type it would probably mean that John has completed his degree and should also be deleted from the computing science graduate student base entity type. We see that deletions to different v-entity types defined from the same base entity type should result in different operations to be performed on the base entity type. There are many situations where view updates are ambiguous [KEL 86a] and Keller's [KEL 86b] algorithm for view translation could be incorporated to deal with such issues.

Finally more research should be done on the feasibility of incorporating into SQUIREL more elaborate features of AI systems like version management, alternate worlds reasoning and truth maintenance facilities.

Appendix A
USE OF SQUIREL CONSTRUCTS

A.1. Variables

The following is a brief description of the variables used in the constructs to be presented. They are given in alphabetical order.

<attribute>	: attribute of a v-entity type or entity type
<category name>/<category>	: set of characters that identifies the category of a hierarchy
<entity type>	: name of a base entity type
<key attribute>	: key attribute of a v-entity type or entity type
<name>	: name of a hierarchy that has been created
<partition name>/<partition>	: name of a partition of a v-entity type
<unique hierarchy name>	: set of characters that identifies a hierarchy
<value>	: value attached to a attribute
<view name>	: name of a v-entity type that has been created
(......)	: optional variables are placed in brackets

Constructs used at the TYPE level will be described below.

A.2. CREATION

The form of creation would be:

```
CREATE HIERARCHY  <unique hierarchy name>
      (CATEGORY = <category name>);
```

Effect:

1. A hierarchy relation having <unique hierarchy name> is created which will automatically have two attributes SUB and SUP.
2. If CATEGORY is given it will be an alias for SUB. Otherwise <name>.CATEGORY will be the CATEGORY name.

Note:

- Reference to this hierarchy is through the <unique hierarchy name> followed by .HIERARCHY.
- Hierarchy names must be unique (the name is checked against those already existing in SYSHIER).
- The categorization refers to v-entity types and indicates the general theme, idea or type of structural breakdown of the hierarchy.

A.3. INSERTION

To insert into the hierarchy we do:

```
INSERT INTO <name>.HIERARCHY
V-ENTITY =  <viewname>, (PAR(TITION)) = <partition name>)
V-ENTITY =  <viewname>, (PAR          = <partition name>)
.
.
V-ENTITY =  <viewname>;
```

Effect:

1. V-entity types are inserted into the hierarchy relation specified.
2. The SYSHIER is updated with the <name> as the **Name** attribute value and <viewname> as the **Participant** attribute value.
3. Once a v-entity type is inserted into a hierarchy, a virtual attribute field is attached to this v-entity type and will store the **partition** name. This virtual attribute will reference a virtual table consisting of the subsumed v-entity types of the v-entity type in the specified hierarchy.

   If PAR is given, the virtual attribute is named <partition name> otherwise it is named <name>.PARTITION. The hierarchy into which the v-entity type is entered is known by <name>.

Note:

- All the values entered must represent v-entity types.
- SQUIREL's classifier is invoked on all new entries (type level). The classifier will be able to determine all other subsumed as well as subsuming v-entity types of the new v-entity type.

A.4. UPDATES

Updates can be performed on a hierarchy as follows:

```
UPDATE <name>.HIERARCHY
SET    PAR      = <partition name>
WHERE  V-ENTITY = <view name>;
```

Effect:

1. The partition (virtual attribute) name attached to the v-entity type used in the hierarchy specified is changed.

Note:

- The user is only able to update the partition name and not the structure of the hierarchy.

A.5. DELETION

Deletions on hierarchies follow this format:

```
DELETE FROM <name>.HIERARCHY
WHERE  V-ENTITY = <view name>;
```

Effect:

1. The v-entity type referenced is deleted from the hierarchy mentioned.
2. The partition name (virtual attribute) attached to this v-entity type for the hierarchy specified is removed.
3. The subsumed v-entity types will be relinked with the subsuming v-entity types of the deleted v-entity type (the classifier is invoked on the subsumed and subsuming v-entity types of the v-entity type deleted).

4. The v-entity type is also deleted as a **participant** in the named hierarchy from the SYSHIER table.

Note:

- If the top level v-entity type is deleted the hierarchy will be split into two.
- The v-entity type itself is **not** deleted, it is only removed from the hierarchy. This means that the v-entity type still exists in the database but is taken out of the specified hierarchy.

A.6. DROP

To DROP a hierarchy in SQUIREL we do:

```
DROP HIERARCHY <name>.HIERARCHY;
```

Effect:

1. The hierarchy relation referenced is dropped and the SYSHIER is updated. All records having the hierarchy relation referenced in the NAME attribute is deleted.
2. For each of the v-entity types that participated in the hierarchy the <partition> used for that hierarchy is removed.

Note:

- The v-entity types that participated in the hierarchy are not dropped.

A.7. RETRIEVAL

The general format of retrievals on hierarchies is:

```
SELECT <category>  FROM <name>.HIERARCHY;

SELECT <partition> FROM <view name>;
```

Effect:

1. All the v-entity types in the named hierarchy are retrieved.
2. The subsumed v-entity types of the v-entity type specified are retrieved.

All the above operations apply to v-entity types (**type** level) of the hierarchy. In addition, entities themselves (individual records) may be used in hierarchy operations. Operations at the **instance** level are described next.

A.8. INSERTION

To insert entities into the hierarchy we do:

```
INSERT INTO <name>.HIERARCHY
VALUES( <attribute>=<value>, <attribute>=<value>,..);
```

Effect:

1. For insertion of entities, an actual record is inserted into the base entity types determined by the classifier.

Note:

- One of the attributes specified **must** be a key attribute.
- When we are inserting entities into the hierarchy, the attribute names must be given.
- For entity insertion, the normal problems of view updates apply [BANC 81].
- SQUIREL's classifier is invoked on all new entries (instance level).

A.9. DELETION

Corresponding to INSERTION, entities can also be directly deleted from a hierarchy:

```
DELETE FROM <name>.HIERARCHY
WHERE <key attribute>=<value>;
```

Effect:

1. The entity specified will be deleted from all entity types in which the entity is stored.

Note:

- The attribute specified must be a key attribute.

A.10. UPDATES

Updates to entities in hierarchies are made in the following form:

```
UPDATE <name>.HIERARCHY
SET    <attribute>       = <value>
WHERE  <key attribute> = <value>;
```

Effect:

1. The attribute value for the entity referenced in the hierarchy is changed.

Note:

- The key attribute must be given.
- Virtual attributes cannot be updated.
- The user is not able to update any of the key attributes.

A.11. RETRIEVAL

To retrieve information from a hierarchy we do:

```
SELECT <category name>  FROM <name>.HIERARCHY
 WHERE <attribute> = <value>;

SELECT <attribute>,...  FROM <name>.HIERARCHY
 WHERE <attribute> = <value>;

SELECT <partition name> FROM <view name>
 WHERE <attribute> = <value>;
```

Effect:

1. The v-entity type name(s) in which the entity is found is/are returned.
2. The attribute(s) for the entity in the hierarchy is/are returned.

Note:

- The key attribute must be given in the WHERE clause.
- The entity is found in every v-entity type that subsumes it.

A.12. SETARG

The SETARG construct is used to define the types of the arguments used in procedures defined in SQUIREL. The SETARG is used in the following form:

```
SETARG <procedure>.ARG
arg1 CHAR(20), arg2 CHAR(3),...,argN NUMBER;
```

Effect:

1. The system entity type <procedure>.ARG is assigned N arguments of the type stated in SETARG.

Note:

1. Every procedure name must be unique in SQUIREL.
2. The procedure can have any number of arguments of different types and these will be defined in the SETARG construct.
3. There is only one entity (tuple) in each <procedure>.ARG and every time an INSERT is performed to a <procedure>.ARG that entity will be overwritten.

A.13. Algorithm for coalescing E-relations

{This algorithm is used to coalesce the E-relations representing the v-entity types in a hierarchy. An important point is that the E-relation names are not physically changed but rather SQUIREL is able to **internally** equate two or more E-relations. This is because a v-entity type may participate in different hierarchies causing the E-relations of its base entity types to be known by different E-relations. For example, a graduate student is subsumed by a student in a personnel hierarchy but it may also be subsumed by employee in an employment hierarchy. Therefore we would not want to physically set the E-relation GRAD.V to be the E-relation STUDENT.? when we enter GRAD.V into the personnel hierarchy and then switch GRAD.? to EMPLOYEE.? when we enter GRAD.V into the employment hierarchy. Instead SQUIREL should be able to internally reference GRAD.? as either STUDENT.? or EMPLOYEE.?}

1. When the first v-entity type is entered into a hierarchy, the E-relations representing the v-entity type is arbitrarily set ot one of the E-relations.

2. Subsequently, when v-entity types are entered into the hierarchy. The type level classifier is first used to structure the v-entity types.
3. The top level v-entity type's base entity type is chosen. The E-relation of this base entity type is used as the connecting E-relation and all other E-relations of base entity types in the hierarchy are set to this E-relation.
4. If there is more than one E-relation for a top level v-entity type, one is chosen arbitrarily.

Appendix B
ALGORITHMS FOR SQUIREL CONSTRUCTS

A brief description of the constructs is given in braces {} before the algorithm is presented.

B.1. TYPE Level Classifier

{The TYPE LEVEL classification algorithm is invoked when a v-entity type is entered into a hierarchy. It determines where the inserted v-entity type should be placed in the hierarchy by finding out the inserted v-entity type's parent and child v-entity types. When a hierarchy is empty, the first v-entity type is assumed to be the top level v-entity type but as more v-entity types are entered the classifier will make the appropriate changes.}

1. Assume the new v-entity type to be classified is named X.
2. Obtain the attributes attached to each v-entity type in the named hierarchy.
3. Check the attributes of the new v-entity type against those of the v-entity types obtained and classify as follows:
   - V-entity types which contain some of the attributes of X subsume X. In other words v-entity types which are proper subsets of X will subsume X.
   - V-entity types which have all of the attributes of X and more are subsumed by X. In this case, X is a proper subset[19] of some other v-entity type in the hierarchy.
   - V-entity types which have exactly the attributes of X are at the same level as X.
   - V-entity types that have contradictory attributes, i.e. v-entity type A has attribute c while B does not and B has attribute d while A does not, are not linked.

[19]We use subset here in terms of the number of attributes a v-entity type has and a child v-entity type will have all the attributes of its parent v-entity type as well as some attributes unique to itself.

4. After the new links are added to the hierarchy relation, redundant links are removed. For example, if we add (A, X) and (X, D) into the hierarchy relation where we previously had (A, D), then (A, D) is removed from the hierarchy relation.
5. If there are no v-entity types that subsume X, an extra link (X, TOP) is added to the hierarchy relation. Now if A is the present TOP v-entity type that is, (A, TOP) currently exists, X may now subsume A and (A, X) is to be added. Thus (A, TOP) is deleted and (X, TOP), (A, X) are added.

We see that the type level classification algorithm proceeds top-down and follows a backward chaining[20] approach in determining where a v-entity type belongs.

B.2. INSTANCE Level Classifier

{ The INSTANCE LEVEL classification algorithm is used to determine the v-entity types that an entity is found in. It will classify an entity to the lowest level v-entity type in a hierarchy. This is because the v-entity type at a lower level will automatically include its parent v-entity types and the entity will be known to be stored in the low level v-entity type as well as the parent v-entity types of this low level v-entity type.}

1. Obtain the "leaf" v-entity types in the hierarchy using the following query:

```
SELECT  SUB FROM <name>.HIERARCHY
 WHERE  SUB not in
(SELECT SUP FROM <name>.HIERARCHY);
```

2. Get attributes of each of the v-entity types selected.
3. Select the v-entity types which have the attributes referenced in the query.
4. Find the applicable v-entity type by:
   a. Discard the v-entity types that are subsumed by some other in the set.
   b. If one v-entity type is left choose it (SUCCESS).
   c. Otherwise obtain all the v-entity types **above** (subsuming) the ones selected and repeat step 2.

---

[20] Start from the top-level v-entity type, which represents a goal and then go down the hierarchy to check its subsumed v-entity types.

d. If no v-entity type(s) are found from step 4b and we have more than one v-entity type left then the insertion is UNSUCCESSFUL.

5. Differences appropriate to insert or select operations.

   - For insertion queries: Once the v-entity type is found, the data will be entered into the appropriate P-relations corresponding to the base entity types of the v-entity type.
   - For select queries: the data will be retrieved from the appropriate P-relations corresponding to the base entity types of the v-entity type.

6. If no v-entity type is found then the insertion is UNSUCCESSFUL.

B.3. CREATE algorithm

{The create operator is used to create a hierarchy. It only informs the system to reserve storage for the hierarchy but the hierarchy will contain nothing until v-entity types are inserted into it.}

1. Check the SYSHIER to determine that a unique hierarchy name is used.
2. IF <hierarchy> is not in SYSHIER
   - Update the SYSHIER to include the new hierarchy.
   - Allocate storage for the new hierarchy relation with attributes SUB and SUP.

   ELSE
   - Inform the user that the hierarchy name given is not unique and therefore cannot be used.

B.4. DROP algorithm

{The drop operator is used to drop a hierarchy from the database. When a hierarchy is dropped the v-entity types are not deleted from the database as they may still be used in some other hierarchies.}

1. The named hierarchy with all its v-entity types is removed from the SYSHIER.
2. The hierarchy relation specified is also removed and storage is freed.
3. The partition attribute of each of the v-entity types in the named hierarchy is

removed. The system table storing the definition of the v-entity types is updated to show this.

B.5. INSERT algorithm <TYPE>

{The type insert operator is used to form the hierarchy and v-entity types which are the components of the hierarchy are inserted here.}

1. Classify the v-entity types entered using the TYPE LEVEL classifier.
2. For each v-entity type entered, a partition attribute is attached to its structure. This attribute will have a user defined name or a default name. The system table storing the v-entity type definition will be updated to include this new attribute.
3. The SYSHIER is updated to include the new v-entity types of the hierarchy.
4. The hierarchy relation is updated to include the new v-entity types.
5. V-entity types that cannot be classified by the TYPE LEVEL classifier are printed out as errors.

B.6. INSERT algorithm <INSTANCE>

{The instance insert operator is used to insert entities into the hierarchy. The user need not know the v-entity types or base entity types where the entity is to be inserted but need only know which hierarchy to insert the entity.}

1. Classify the entity into its correct v-entity type using the INSTANCE LEVEL classification algorithm.
2. Check is made to ensure that the entity includes a key attribute.
3. Create new system surrogate for entity named.
4. Using the v-entity type found in step 1, find its base entity types.
5. Using the default entity type, map user given attribute values over the default attributes.
6. Using the E-relation and the P-G relation insert this new entity into the appropriate P-relations of the base entity types found in step 4.

B.7. DELETE algorithm <TYPE>

{The type delete operator is used to delete v-entity types from a hierarchy.}

1. V-entity type specified is removed from the hierarchy relation as well as the SYSHIER relation.
2. The partition attribute of the v-entity type that corresponds to the named hierarchy is removed. The system table storing the v-entity type definition is updated to show this.
3. The TYPE LEVEL classifier is invoked to restructure the hierarchy because of the deleted v-entity type and the hierarchy relation is updated accordingly.

B.8. DELETE algorithm <INSTANCE>

{The instance delete operator is used to delete entities from a hierarchy. Note that attributes of entity types cannot be deleted and only attribute values can be deleted.}

1. Classify the entity into its correct v-entity type using the INSTANCE LEVEL classification algorithm.
2. Determine the base entity types of the v-entity type found using the system table storing the v-entity type definitions.
3. Determine the base entity types which store the attributes that are to be deleted.
4. Delete the entity specified from the E-relations and P-relations of the base entity types that have the attributes specified.
5. At any of the above steps if a v-entity type or base entity type is not found, an error is returned.

B.9. RETRIEVAL algorithm <TYPE and INSTANCE>

{The type and instance select operator is used to retrieve information about the v-entity types in a hierarchy. Information asked for are in two forms, one concerning the structure of the hierarchy, i.e, the subsumed or subsuming v-entity types and the other concerning the attribute values of v-entity types in a hierarchy.}

1. If an entity is specified, find the surrogate key for that entity from its base entity type in the specified hierarchy.
2. Determine if attributes, partitions/categories (v-entity type names) or both is needed.
3. If attributes are needed.
   - Check each 'leaf' v-entity type in the hierarchy specified until the surrogate found in step 1 matches.
   - Return the needed attributes. (Remember that a leaf v-entity type inherits all attributes of its subsuming v-entity types. Therefore all attributes are attached to the leaf v-entity type.)
4. If category/partition is needed
   - Determine which v-entity type(s) the surrogate key found in step 1 lies.
   - Return the v-entity type name(s).

B.10. UPDATE algorithm <TYPE>

{The type update is only used to update the partition attribute name of a v-entity type in a given hierarchy.}

1. The SYSHIER is checked to determine that the v-entity type specified exists in the named hierarchy.
2. If the v-entity type exists, the partition attribute name is changed and the system table that stores the v-entity type definition is updated to show this.
3. An error is reported if the v-entity type does not exist.

B.11. UPDATE algorithm <INSTANCE>

{The instance update is used to update attribute values of v-entity types in a hierarchy. Note that the number of attributes cannot be updated through the hierarchy but only the value of attributes can be updated.}

1. Classify the entity into its correct v-entity type using the INSTANCE LEVEL classification algorithm.
2. Determine the base entity types of the v-entity type found using the system table storing the v-entity type definitions.
3. Determine the base entity types which store the attributes that are to be updated.
4. Update the P-relations of the base entity types that have the attributes specified.
5. At any of the above steps if a v-entity type or base entity type is not found, an error is returned.

B.12. TRIGGER algorithm

{This is also the interface procedure used in conjunction with the trigger construct. The trigger construct can be used in three ways, in conjunction with any SQUIREL operation, used to invoke a procedure directly or to attach a procedure to a base entity type where every time that base entity type is accessed, the procedure will be invoked.}

1. If TRIGGER is used with a SQUIREL operation (SELECT, DELETE, etc.) or if it is used directly like a *call* then the procedure table will be checked for the existence of the procedure.
2. The procedure is invoked through an interface with a users' program and the arguments to the procedure are passed.
3. Errors which apply to procedures, e.g., wrong arguments are caught by the procedure.
4. If the TRIGGER is used to attach a procedure to a base entity type the procedure table is updated to show this.

B.13. Pseudo Code for the PLAYBACK rule

```
PLAYBACK:  PROC OPTIONS(MAIN)
           DCL PLAYBACK.ARRAY(100)        CHAR  (20),
               I,J                        FIXED (3),
               HOLD,NEXT,SUB,SUP,HNAME    CHAR  (20);
 /**********************************************/
 /* using embedded SQL, we first get the arguments*/
 /* HNAME will hold the hierarchy name and NEXT   */
 /* will hold the v-entity type whose path is to  */
 /* be found                                      */
 /**********************************************/
          EXEC SQL DECLARE PLAYBACK.ARG TABLE
                          (HNAME  CHAR(20),
                           ETNAME CHAR(20) );

          EXEC SQL SELECT HNAME, ETNAME
                   INTO  :HNAME,:NEXT

          HNAME = HNAME||'.HIERARCHY';
          I = 1;
 /**********************************************/
 /* we now find the path using the hierarchy    */
 /* relation. Remember that the hierarchy       */
 /* relation has two arguments SUB and SUP which */
 /* are the SUBORDINATE and SUPERORDINATE       */
 /* v-entity types respectively. The TOP level  */
 /* v-entity type will have a SUP of TOP        */
 /**********************************************/
       DO WHILE NEXT ~= 'TOP';
          EXEC SQL DECLARE :HNAME TABLE
                           (SUB CHAR(20), SUP CHAR(20));

          EXEC SQL SELECT SUP
                   INTO  :SUP
                   FROM  :HNAME
                   WHERE  SUB = :NEXT;

          PLAYBACK.ARRAY(I) = SUP;
          NEXT = SUP;
          I = I + 1;

          END;
 /**********************************************/
 /* the path is now printed out                 */
 /**********************************************/
         DO J = I TO 1 BY -1
         PUT SKIP LIST(PLAYBACK.ARRAY(J));
         END;
```

```
END PLAYBACK;
```

Appendix C

Pseudo code for the OAD rule

```
OAD.RULE:     PROC OPTIONS(MAIN);
              DCL PNO                  FIXED(8),
                  TLC                  FIXED(2),
                  FVC, MMF, RV/TLC FIXED(3);
                           .
                           .

      EXEC SQL DECLARE OAD.V VIEW
               (PNO    NUMBER (8), TLC NUMBER (2),
                FVC    NUMBER (3), MMF NUMBER (3)
                RV/TLC NUMBER (3))

      EXEC SQL SELECT PNO, TLC, FVC, MMF, RV/TLC
                INTO :PNO,:TLC,:FVC,:MMF,:RV/TLC
                FROM  OAD.V

/*******************************************************/
/* rules to determine the certainty factors are shown */
/*******************************************************/
      IF TLC < 15 AND TLC > 10
         THEN TLCERTAIN = 0.21
          .
          .
      IF MMF > 15 AND TLC > 0
         THEN MFCERTAIN = 0.13
          .
          .
      IF FEV1 < 20
         THEN FCERTAIN = 0.1
          .
          .
      TOTALCERTAIN = TLCERTAIN + MFCERTAIN + FCERTAIN

PRINT 'The reason this patient might have an OAD
       (OBSTRUCTIVE AIRWAYS DISEASE) is because
        THE TLC IS ,'TLC'
        THE MMF IS ,'MMF'
        THE FVC IS ,'FVC'
        which makes the certainty that he has an
```

```
       OAD, 'TOTALCERTAIN')
       .

/****************************************************/
/* we now obtain the subsumed v-entity types of OAD*/
/* under the PARTITION of OAD.V which is POSSIBLE  */
/****************************************************/

EXEC SQL DECLARE X CURSOR FOR
         SELECT POSSIBLE
         FROM   OAD

PRINT 'The possible kinds of OAD for patient', PNO,'are:'

 EXEC SQL OPEN X:
     DO WHILE (more records)
       EXEC SQL FETCH X INTO POSSIBLE
       PRINT POSSIBLE
     END;
 EXEC CLOSE X;

/*****************************************************/
/* the user is prompted for more information which  */
/* will help in further classification              */
/*****************************************************/
PRINT" Could you please tell me "PNO"  RV/TLC:"
                                  (ENTER RV/TLC)

 /****************************************************/
 /* an update is performed on the OAD.V which will  */
 /* automatically update the hierarchy and lead to  */
 /* other rules being invoked depending on which    */
 /* v-entity type is updated                        */
 /*                                                 */
 /* in this particular case if RV_TLC is 25, the    */
 /* v-entity type SEVERE.V is updated and the rules */
 /* attached to SEVERE.V (SEVERE.RULE) will be fired*/
 /****************************************************/
   EXEC SQL UPDATE OAD.V
            SET RV/TLC = :RV_TLC
            WHERE  PNO = :PNO;

END;
```

References

[ABITE 87] Abiteboul S & Hull R.
IFO: A Formal Semantic Database Model.
ACMTODS 12(4), Dec, 1987.

[AIKINS 84] Aikins J.S.
A representation scheme using both frames and rules.
Rule Based Expert Systems :424-440, 1984.

[BANC 81] Bancilon F & Spyratos N.
Update Semantics of Relational Views.
ACMTODS 6(4), 1981.

[BOBROW 77] Bobrow D.G & Winograd T.
An Overview of the KRL-O, A Knowledge Representation Language.
Cognitive Science 1(1), Jan, 1977.

[BRAC 79] Brachman R.J.
On the Epistemological Statues of Semantic Networks.
In *Associative Networks-The Representation of Knowledge in Computers.* Academic Press, New York, 1979.

[BROD 86] Brodie M, Jarke M.
On Integrating Logic Programming and Databases.
In *Expert Database Systems.* The Benjamin Cummings Publishing Company, 1986.

[BROWNSTON 86]
Brownston L, Farrell R, Kant E & Martin N.
Programming Expert Systems in OPS5.
Addison-Wesley, 1986.

[BUCHANAN & SHORTLIFFE 84]
Buchanan B.G. & Shortliffe E.H.
Rule Based Expert Systems.
Addison-Wesley, 1984.

[CADIOU 76] Cadiou J.M.
On Semantic Issues in the Relational Model of data.
In *Proc. of 5th Symp. on Math. Foundations of Comp. Sc.*, pages 23-28. Springer-Verlag, 1976.

[CHANG 86] Chang C.L & Walker A.
PROSQL: A Prolog Programming Interface with SQL/DS.
In *Expert Database Systems*, pages 233-246. The Benjamin Cummings Publishing Company, 1986.

[CLOCKSIN & MELLISH 82]
Clocksin W.F & Mellish C.S.
Programming in PROLOG.
Springer-Verlag, 1982.

[CODD 70] Codd E.F.
A Relational Model of Data For Large Shared Data Banks.
CACM 13(6):377-387, June, 1970.

[CODD 79] Codd E.F.
Extending the Database Relational Model to Capture More Meaning.
ACMTODS 4(4):397-434, 1979.

[DATE 86] Date D.J.
An Introduction to Database Systems, VOL 1.
Addison-Wesley, 1986.

[FIKE 85] Fikes R & Kehler T.
The Role of Frame-Based Representation in Reasoning.
CACM 28(9):904-920, Sept, 1985.

[FORGY 81] Forgy C.L.
OPS5 Users Manual.
Technical Report, Carnegie-Mellon University, 1981.

[GERV 87] Gervarter B.W.
The Nature and Evaluation of Commercial Expert System Building Tools.
IEEE Computer 13(6):24-40, May, 1987.

[HAN 86] Han J, Li Z.
DQL -- The Deductive Augmentation of Relational Database Query Language QUEL.
AUTOMATED ANALYSIS OF LEGAL TEXTS :861-874, 1986.

[HARANDI 86] Harandi M.T, Schang T & Cohen S.
Rule Base Management Using Meta Knowledge.
In *Expert Database Systems.* The Benjamin Cummings Publishing Company, , 1986.

[IBM DB2] IBM Form No GC 26-4082.
IBM Database 2.
Technical Report, IBM, , .

[IBM SQL/DS] IBM Form No G 320-6590.
SQL/Data System for VSE, A Relational Data System for Application Development.
Technical Report, IBM, , .

[KEL 86a] Keller Arthur.M.
The Role of Semantics in Translating View Updates.
IEEE Computer , January, 1986.

[KEL 86b] Keller Arthur.M.
Choosing a View Update Translator by Dialog at View Definition Time.
In *Proceedings of Twelfth Int. Conference on VLDB.* , August, 1986.

[KUNZ 78] Kunz G.
PUFF: A Pulmonary Disease Fact Finder.
In *Proceedings of the Conference on Rule Based Systems.* , March, 1978.

[REITER 78a] Reiter R.
On Closed World Databases.
In *Gallaire H, Minker J (eds), Logic and Databases.* Plenum New York, 1978.

[REITER 78b] Reiter R.
On Reasoning by Default.
In *Proceedings 2nd Symposium on Theoretical Issues in Natural Language Processing.* July, 1978.

[REITER 80] Reiter R.
A Logic for Default Reasoning.
Artificial Intelligence 13:81-132, June, 1980.

[SMITH 77] Smith J.M & Smith D.C.P.
Database Abstractions: Aggregation and Generalization.
ACMTODS 2(2), June, 1977.

[STON 76] Stonebraker M, Wong E, Kreps P & Held G.
The Design and Implementation of INGRES.
ACMTODS 1(3), Sept, 1976.

[STON 85] Stonebraker M.
Adding Semantic Knowledge to a Relational Database System.
Springer Verlag , 1985.

[ULL 82] Ullman Jeffrey D.
Principles of Database Systems.
Addison-Wesley, 1982.

[ULL 85] Ullman Jeffrey D.
Implementation of Logical Query Languages for Databases.
ACMTODS 10(3):289-321, September, 1985.

[VESONDER 83] Vesonder G.T, Stlofo S.J, Zielinski J.E, Miller F.D, & Copp D.H.
ACE: An Expert System for Telephone Cable Maintenance.
In *Proceedings of 8th IJCAI*, pages 116-121. 1983.

[WOODS 83] Woods W.A.
What's important about knowledge representation.
IEEE Computer 15(10):22-29, 1983.

[ZLOFF 77] Zloof M.M.
Query-By-Example: A Database Language.
IBM SYSTEM JOURNAL 16(4):324-343, 1977.