RABBIT: AN INTERFACE FOR INFORMATION RETRIEVAL BY REFORMULATION

by

Frederich Nelson Tou

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREES OF
BACHELOR OF SCIENCE
and
MASTER OF SCIENCE
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June, 1982

(c) Frederich Nelson Tou, 1982

The author hereby grants to MIT permission to reproduce and to distribute copies of
this thesis document in whole or in part.

Signature of Author

Department of Electrical Engineering and Computer Science, May 17, 1982

Certified by

Fernando J. Corbató
Thesis Supervisor

D. Austin Henderson, Jr.
Company Supervisor

Michael D. Williams
Company Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Graduate Committee

Archives

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

OCT 20 1982

LIBRARIES
RABBIT: An Interface for Information Retrieval by Reformulation

by

Frederich Nelson Tou

Submitted to the Department of Electrical Engineering and Computer Science on May 17, 1982 in partial fulfillment of the requirements for the Degrees of Bachelor of Science and Master of Science in Computer Science

Abstract

A new kind of user interface for information retrieval has been designed and implemented. This interface, named RABBIT, relies upon a new paradigm for retrieval, retrieval by reformulation, based on a psychological theory of human remembering. To make a query, the user interactively constructs a description of his target item(s) by criticizing successive example (and counterexample) instances. These instances from the database are presented to the user from a well-defined perspective inferred from the user's query description. This restricted presentation, which makes use of the KL-ONE formalism, prevents the user from creating semantically improper query descriptions.

RABBIT is intended for users who work infrequently with a complex database or who must deal with a multitude of databases. A second class of users for which this interface should prove useful are users who approach a database with only a vague idea of what it is that they want and who thus, need to be guided in the (re)formulation of their queries.

Academic Thesis Supervisor: Fernando J. Corbató  
Title: Professor of Computer Science and Engineering

Title: Member of the Research Staff, Xerox Palo Alto Research Centers

Company Thesis Supervisor: Michael D. Williams  
Title: Member of the Research Staff, Xerox Palo Alto Research Centers
Acknowledgements

This thesis would not have been possible without the generous support and encouragement of many people.

I would first like to thank John Tucker and the VI-A Program for providing me the opportunity to work at Xerox PARC.

I would especially like to thank Austin Henderson and Mike Williams, who were always available to answer my many questions and who were a continual source of wisdom, guidance, encouragement, and ideas. Their enthusiasm for this thesis was unbounded.

I am indebted to Richard Fikes and Tom Malone. They, along with Austin Henderson and Mike Williams, have contributed greatly to the design of RABBIT. As a result of many discussions with them, the essential ideas underlying RABBIT were distilled from the many initial bits and pieces.

I would also like to thank John Brown and the members of the Cognitive and Instructional Sciences Group for their support throughout this endeavor.

Special thanks go to Austin Henderson, Mike Williams, Richard Fikes, Tom Malone, Prof. Fernando Corbató, and Steve Weyer, who read early drafts of this thesis and provided many helpful comments. If this thesis is at all clear and understandable, it is a result of their efforts.

Many other people, both at PARC and MIT, have provided moral and technical support and to all of them, my deepest thanks.

Finally, I would like to express my warmest thanks and gratitude to my parents for their continued support and encouragement.
Table of Contents

List of Figures ............................................................ 7

1. Introduction ............................................................... 9

2. The Problem Space .................................................. 11
   1. Incomplete Knowledge ............................................. 11
      1.1 Incomplete knowledge about the descriptive terms ....... 11
      2.2 Uncertainty about the goal .................................. 12
   2. Overabundance of Information .................................... 12
   3. Heterogeneity of the Database ................................. 13

3. An Example ........................................................... 15
   1. The Display ........................................................ 15
   2. Constructing a Simple Query Description ....................... 17

4. Retrieval by Reformulation ........................................ 30
   1. Descriptive Retrieval ............................................ 30
   2. Retrieval by Instantiation ....................................... 31
   3. Retrieval by Reformulation ..................................... 32
      3.1 Retrieval by constructed descriptions ..................... 32
      3.2 Interactive construction of queries ....................... 33
      3.3 Critique of example instances ............................... 33
      3.4 Dynamic perspectives ......................................... 35
   4. The Example Revisited ........................................... 36

5. The KL-ONE Database ................................................. 39
   1. The KL-ONE Epistemology ......................................... 39
   2. KL-ONE: A Medium for Representing Heterogeneous Data ........ 40
   3. The Representation of RABBIT Instance Descriptions in KL-ONE .... 42
   4. The Representation of RABBIT Query Descriptions in KL-ONE ... 42
      4.1 The structure of query descriptions ....................... 43
      4.2 Representing instance classes in KL-ONE .................. 44
      4.3 Representing attribute-value constraint pairs in KL-ONE ... 48
   5. Using Descriptive Information Embedded in KL-ONE ............ 49

6. The User Interface .................................................. 50
   1. The Goals .......................................................... 50
   2. The Display ........................................................ 50
      2.1 The description pane ......................................... 52
      2.2 The example pane ............................................. 52
      2.3 Making selections ............................................ 53
      2.4 Updating the display ......................................... 54
   3. The Basic Operations ............................................. 54
      3.1 Invocation of the operations ................................. 54
      3.2 Require/Prohibit ............................................. 55
      3.3 Alternatives .................................................. 56
         3.3.1 Alternative values for attributes ..................... 56
         3.3.2 Alternative instance classes ......................... 57
         3.3.3 Require any/Require all/Prohibit ..................... 57
      3.4 Specialize .................................................... 65
      3.5 Remove/Clear ................................................ 65
3.6 Other operations: describe, alternative views, retrieve .................................................. 70
4. Semantic Consistency .................................................................................................................. 72

7. Perspectives ................................................................................................................................. 74
1. The Uses of Perspectives .............................................................................................................. 74
1.1 Controlling the presentation of instances .................................................................................. 74
1.2 Facilitating the understanding of instances .............................................................................. 75
1.3 Constraining the search space ................................................................................................. 76
1.4 Managing the non-uniformity of the database ......................................................................... 77
2. Changing Perspectives .................................................................................................................. 80
3. Extended Descriptions .................................................................................................................. 82

8. Embedded Descriptions ................................................................................................................. 86
1. The Uses of Embedded Descriptions ............................................................................................. 86
1.1 Recursive construction of descriptions ....................................................................................... 86
1.2 Inspection of unknown values of attributes .............................................................................. 89
2. Example I: Describing a Value Constraint .................................................................................... 89
3. Example II: Describing an Instance Class ..................................................................................... 94
4. Extended Descriptions .................................................................................................................. 97

1. Increasing the Domain of Expressible Queries ............................................................................ 99
1.1 Creating general boolean expressions ....................................................................................... 99
1.2 Defining additional relationships between an attribute and its values: Set Operators .............. 100
1.3 Other ............................................................................................................................................. 102
2. Extensions to the Functionality of RABBIT .................................................................................. 104
2.1 Browsing ..................................................................................................................................... 104
2.2 Maintaining the semantic consistency ....................................................................................... 105
2.3 Direct user control of the perspective ....................................................................................... 105
2.4 Help facilities ............................................................................................................................. 105
2.5 Assistance for the expert user ..................................................................................................... 106
2.6 Other ............................................................................................................................................. 108
3. Adapting RABBIT to Traditional Data Models ........................................................................... 108

10. Areas for Further Research ........................................................................................................ 109
1. Evaluation ..................................................................................................................................... 109
2. Perspectives ................................................................................................................................... 109
3. Structural Descriptions ............................................................................................................... 112
4. Backtracking .................................................................................................................................. 112
5. Practical Databases ...................................................................................................................... 112
6. What Is an Instance? ..................................................................................................................... 114

11. Related Work ............................................................................................................................... 115
1. Database Query Languages ......................................................................................................... 115
   DIALOG ......................................................................................................................................... 115
   SQUARE and SQL .......................................................................................................................... 115
   THOMAS ....................................................................................................................................... 116
2. Graphical Query Languages ......................................................................................................... 116
   QBE .............................................................................................................................................. 117
   FORAL LP ..................................................................................................................................... 117
   ZOG and PDB ................................................................................................................................. 117
   VIEW ............................................................................................................................................ 118
   FINDIT .......................................................................................................................................... 119
3. Knowledge Representation Languages ........................................................................................ 119
   KRL ............................................................................................................................................... 119
   PIE .............................................................................................................................................. 119
12. Summary .......................................................... 120
 Bibliography ........................................................... 122
 Biographical Note .................................................... 125
List of Figures

2.1 An example relational database ................................................. 14
3.1 The RABBIT interface ......................................................... 16
3.2 The initial display ............................................................. 20
3.3 The menu of operations ....................................................... 21
3.4 The specializations of Entity ................................................ 22
3.5 Requiring the instance class Product ....................................... 23
3.6 Retrieving a new example instance ......................................... 24
3.7 Requiring the manufacturer to be Xerox or Apple ......................... 25
3.8 Retrieving another example instance ...................................... 26
3.9 A sub-window for describing a display .................................. 27
3.10 Specifying a bit-map display ............................................... 28
3.11 The completed query description ......................................... 29
4.1 The interrelationships between the user and the database ................ 37
5.1 A piece of KL-ONE taxonomy ................................................. 41
5.2 ANDing two concepts ......................................................... 45
5.3 ORing-two concepts .......................................................... 45
5.4 Negating a concept ............................................................ 47
5.5 Decompositions of Restaurant ................................................. 47
6.1 The RABBIT interface ......................................................... 51
6.2 An example of a query description ......................................... 52
6.3 An example of an image ...................................................... 53
6.4 Finding alternative values ................................................... 59
6.5 Specializing instance classes ............................................... 66
6.6 The retrieve command ....................................................... 71
7.1 Two perspectives of the Star-8011 ........................................... 78
7.2 Possible implementations of alternative views .............................. 81
7.3 Finding alternative views .............................................. 83
8.1 An inaccessible instance class (Person) ........................................ 88
8.2 Describing the value of an attribute ........................................ 90
8.3 An embedded description within an embedded description .......... 96
8.4 An embedded description of an instance class .......................... 97
9.1 Two reasonable, but inexpressible, queries ............................... 99
9.2 ORed values of an attribute : .............................................. 101
9.3 ANDed values of an attribute .............................................. 101
9.4 A query description in extended RABBIT ............................... 101
9.5 "Definition" of Man .......................................................... 107
9.6 "Definition" of LegalPerson ................................................. 107
10.1 Concept and perspectives hierarchies .................................... 111
Chapter 1 - Introduction

RABBIT is an information retrieval interface which takes a new approach to information retrieval. The design of RABBIT began with an examination of ideas borrowed from cognitive science and knowledge representation. From those ideas, a new paradigm for information retrieval, retrieval by reformulation, has been developed, and a small experimental system based on that paradigm has been implemented in the Smalltalk programming language [Ingalls 78] on the Xerox Dolphin and Dorado personal computers [Lampson 80] and runs over a set of sample databases.

The motivation for designing a new kind of database interface was the unsuitability of existing database interfaces for casual users. Some database interfaces (e.g., SQUARE [Boyce 75] and SQL [Chamberlin 76]) require many hours of instruction to learn; others have a syntax which users find difficult to use and understand (e.g., the boolean expressions of DIALOG [Lockheed 79]). Interfaces based on the relational data model [Codd 70] usually require the user to know in advance which tables and attributes he will be needing, while users of network databases (such as ZOG [Robertson 81]) frequently get lost during the course of their search. (A more complete survey of related work can be found in chapter 11.)

The approach taken in RABBIT was to look at a theory of human remembering (e.g., [Williams 81a]) to learn what were some of the techniques people used in remembering. People are constantly recalling thoughts from their own memories, and it seemed quite plausible that the techniques and strategies which people have developed over time could be applied with good results to the task of retrieving information from an electronic database. Moreover, we believed that an interface modelled after techniques which people use in remembering should be easy to learn and use.

KL-ONE [Brachman 79] was chosen as the medium for storing information because of its rich, well-defined semantics for representing knowledge. In particular, the objects of KL-ONE appeared to be a natural representation for the descriptive terms proposed in the theory of human remembering with which we were working. Furthermore, by embedding the database in a KL-ONE network, we were able to utilize some of the semantics of KL-ONE to prevent the user from constructing certain kinds of meaningless queries.

In the course of exploring the ideas behind RABBIT, we have developed some hypotheses (discussed in chapter 2) regarding the nature of some of the problems of information retrieval. These conjectures concerning the underlying sources of difficulty which people have in retrieving information from databases have, in turn, helped direct our efforts in the development of our proposed solution, the paradigm of retrieval by reformulation.
Our general methodology can be summarized as follows: we first selected a new point in the
design space (i.e., the space of alternative design approaches), developed a partial theory around
that point, built a working model to test the validity and usefulness of that theory and to point out
the directions in which that theory needed improvement or looked promising, and then, began a
new cycle by choosing a new point in the design space along one of the directions pointed out by
the working model. (So far, most of our efforts have been devoted to the design and
implementation phases with only an informal evaluation being conducted at the end of each cycle.
But we have now reached a point where RABBIT can be used to make “useful” queries. Chapter
10 discusses a possible format for testing and evaluating RABBIT.)

In building the working models, which are successive implementations of RABBIT, we have not
been too concerned with issues such as the response time of the interface, size of the database, or
certain details of the user interface such as the optimum placement of windows on the workstation
display and the names and syntax of the commands (however, we do focus on the semantics of
commands). Although these are important issues, they are outside the scope of this thesis. Those
issues have been dealt with only to the extent that they do not significantly distract a user of
RABBIT from the larger problem of learning and using a new technique for retrieving
information—retrieval by reformulation. However, having built several working implementations
of RABBIT, we feel that none of these other issues are intractable.

The intended users of RABBIT are casual users. The term casual user has been defined in many
ways [Cuff 80], but we use that term to refer to users who have had some experience using
computers and bit-map displays, but who are not necessarily programmers or frequent users of a
database retrieval program. We also consider an expert user who works with a multitude of
databases to be a casual user since the amount of time he may spend with any one database is
small. In each of the above cases, the user needs to discover the structure of the database and the
method for retrieving information from that structure. An understanding of the structure should
enable a user to move freely and knowledgeably through the database.
Chapter 2 - The Problem Space

We believe that some of the major sources of difficulty which plague people attempting to retrieve information are incomplete knowledge about the domain and about the particular item being sought, the (over)abundance of information known in the database about any given item, and the heterogeneous structure of the database. These problems seem to be general issues applicable to any kind of information retrieval interface:

1. Incomplete Knowledge

1.1 Incomplete knowledge about the descriptive terms

One source of difficulty for users results from the fact that their initial queries are often incompletely specified due to incomplete knowledge about the descriptive terms. Once a user has formulated his conceptual query (i.e., he knows what he is looking for), he must then learn what the domain-specific terms are for describing that target item to the computer. For example, in a personnel database the relevant descriptive terms might include "name," "salary," and "social security #," whereas in a products database, the appropriate terms might include "name," "price," "color," and "manufacturer."

However, even if he knows what the appropriate terms are, the user may still be unable to specify completely his query. If the terms represent binary relationships between the target item (the item for which the user is searching) and some other items (called the values of those relationships), then the user also needs to know what items are appropriate for filling the value slots of those relationships. Consider again the preceding example of a products database. The appropriate values for filling the "color" descriptor might include "red" and "blue" but not "mauve" or "MasterCharge"; the database might understand the value "$10,000" but not "10,000" for the "price" descriptor. In general, the user needs to be told what the range of legal values is, i.e., what are the possible values which the computer understands.

So there are two kinds of domain-specific knowledge which the user must somehow acquire through the interface in order to construct his query—the names of the appropriate descriptive terms, and for those terms which represent relationships, the names of the appropriate values.
1.2 Uncertainty about the goal

Up till now we have assumed that the user was able to compose a conceptual query in his head, but there are situations where this need not be true; the user may have only a vague idea of what he wants and would really like to browse through the database [Oddy 77]. An example scenario (suggested by Richard Fikes) is the following. After a hard day at the office, a worker, feeling hungry and not wishing to go home and cook, decides to go out to some restaurant. Having heard that the office database contains information about local restaurants, he begins to use that database to find a restaurant. At this point, the extent of the office worker's query is simply the single term 'Restaurant'. The database interface should now somehow lead him to his goal, a restaurant that he might like. For example, the interface could show him the descriptions of some example restaurants and allow him to specify which descriptors of those restaurants he likes or does not like.

This kind of search is not so much random browsing as it is goal-directed browsing; the user is not looking for just anything, but rather for something which matches an initially vague specification derived from an unarticulated, but possibly demanding, intent. Now, as the user browses through the database via the interface, he may notice new information which can enable him to articulate his intention and thus, elaborate in more detail his query specification until he finds what he wants.

2. Overabundance of Information

Even after successfully constructing his query, the user may still become confused if the database has a large amount of information stored about the retrieved item. Presenting everything the system knows about that item could overwhelm the user with too much information, but showing too little information might not answer the user's query to his satisfaction.

For example, suppose the retrieved item is a description of the man John Smith and that the user wishes to find information about Smith's family. Retrieving John Smith's name, age, address, phone number, salary, social security number, spouse's name, and children's names is probably giving the user too much information, but retrieving just the name "John Smith" does not provide enough information.

The basic problem is deciding which and how much of the total information known in the database about a given item should be displayed to the user. For many existing databases, this is not a problem since the amount of information stored about a particular item is not too much even though the total number of items may be quite large. However, for databases which are intended to contain many different kinds of information about any one item (e.g., a database
which treats people as employees, home owners, social beings, and life forms), the large amount of such disparate information may be a source of confusion to users.

3. Heterogeneity of the Database

The structure of the database itself may also be a source of confusion. Since the structure of information in the real world tends to be highly complex ([Kent 76] as reported by [Date 77]), databases which attempt to store "real world" information are also likely to be very complex. The problem is that sufficiently large and complex databases have a non-uniform structure; the items in a database are not all described in the same way. So, for example, people are described using one set of descriptors while products are described in terms of another set.

Many existing databases handle this heterogeneity by partitioning the items of the database into groups which do exhibit uniform structure. For example, a relational database of businesses, products, and people would typically have a fixed relation defined for all businesses, another one for all products, and a third for all people (see Fig. 2.1). The net effect is that the database is really composed of three smaller databases, and switching from, say, businesses to products is akin to switching to a new database. A more important problem is that all items of a given type have the same descriptors. So, for example, all businesses in the database in Fig. 2.1 would have the descriptors "name," "location," "products," and "employees." But suppose we wanted to talk about the type of cuisine served by a restaurant. Adding the descriptor "cuisine" to the Business relation is not a reasonable action since for most businesses, "cuisine" is an irrelevant descriptor. On the other hand, creating a new relation for restaurants seems a bit strange since all the descriptors in the business relation are also applicable to restaurants, i.e., restaurants have those descriptors by virtue of their being a subclass of businesses.

Because of the wide variation in the kinds of information which one might want to store, databases should support a non-uniform shape. Consequently, what the user sees regarding the structure of the database should change depending upon where he is in the database. An interface to such a database should allow the user to navigate through that changing space.
### Business

<table>
<thead>
<tr>
<th>name</th>
<th>location</th>
<th>products</th>
<th>employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerox</td>
<td>Palo Alto</td>
<td>Star-8011</td>
<td>Bill Jones</td>
</tr>
</tbody>
</table>

### Product

<table>
<thead>
<tr>
<th>name</th>
<th>price</th>
<th>color</th>
<th>manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pen</td>
<td>$0.20</td>
<td>Blue</td>
<td>Parker</td>
</tr>
</tbody>
</table>

### Personnel

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
<th>age</th>
<th>social security #</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Smith</td>
<td>$16,000</td>
<td>24</td>
<td>421-05-3718</td>
</tr>
</tbody>
</table>

Figure 2.1: An example relational database
Chapter 3 - An Example

This chapter presents a simple example of using RABBIT to retrieve a product from a database which includes products, businesses, and people. In our discussions, we will refer to the items in a database as instances. But before proceeding with the example, we will first give a brief description of the theory underlying the interface (the next chapter contains a more detailed discussion of our paradigm for retrieval).

In RABBIT, the user makes a query by incrementally constructing a partial description of the target instance, the instance for which he is searching. To assist him in constructing his description, RABBIT provides a description of an example instance, an instance in the database which matches the user's partial description. The description of that instance is the image of that instance with respect to some well-defined perspective inferred from the partial description (more will be said about perspectives in chapter 7). The user can select the descriptors of that image and incorporate those descriptors, or variations of those descriptors, into his partial description. This query-building process is iterative in that the user can at any time request the interface to retrieve a new example instance, one which matches the latest version of his (partial) description, and then use the descriptors of that new image to build up his partial description further.

1. The Display

RABBIT runs on a workstation equipped with a bit-map display, keyboard and mouse (a pointing device for making selections on the display). The screen of the workstation display contains a rectangle, called a window, divided into four sub-windows, called panes (see Fig. 3.1). The text above the top left corner, “Main Description,” is the title of the window. Similarly, each of the panes also has a title. The pane in the upper right corner of the window, labelled “Description,” is called the description pane and contains the current version of the partial description on which the user is working. The pane midway down the window is the example pane, which contains an image of an example instance. This pane is divided into two parts: the top half shows the descriptors of the image while the bottom half presents some purely textual information about the instance. The previous description pane, the pane in the lower left corner, shows the previous partial description (the (preceeding) version of the current partial description which was used in retrieving the instance currently in the example pane) while the pane to its right, the matching examples pane, contains a list of all instances matching the previous partial description, including the instance being displayed in the example pane.
<table>
<thead>
<tr>
<th>Main Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--Attributes of the query--</td>
<td></td>
</tr>
<tr>
<td>Entity</td>
<td></td>
</tr>
<tr>
<td>Product</td>
<td></td>
</tr>
<tr>
<td>Computer</td>
<td></td>
</tr>
<tr>
<td>manufacturer:</td>
<td></td>
</tr>
<tr>
<td>Xerox,</td>
<td></td>
</tr>
<tr>
<td>or Apple</td>
<td></td>
</tr>
<tr>
<td>display: Large-Format-Display</td>
<td></td>
</tr>
<tr>
<td>--- But Not ---</td>
<td></td>
</tr>
<tr>
<td>CPU: 6502</td>
<td></td>
</tr>
</tbody>
</table>

| Example |
| Computer |
| OEM-Product |
| RetailProduct |
| name: TheStar-8011 |
| cost: $13,850, $15,055 |
| manufacturer: Xerox |
| disk: Xerox-10 |
| display: Large-Format-Display |
| memory: Xerox-192-memory |
| CPU: 8000 |

---Information about Star-8011---

<table>
<thead>
<tr>
<th>Previous Description</th>
<th>2 Matching Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>-- -- -- -- -- -- -- --</td>
<td>-- -- -- -- -- -- --</td>
</tr>
<tr>
<td>Entity</td>
<td></td>
</tr>
<tr>
<td>Product</td>
<td></td>
</tr>
<tr>
<td>Computer</td>
<td></td>
</tr>
<tr>
<td>manufacturer:</td>
<td></td>
</tr>
<tr>
<td>Xerox,</td>
<td></td>
</tr>
<tr>
<td>or Apple</td>
<td></td>
</tr>
<tr>
<td>display: Large-Format-Display</td>
<td></td>
</tr>
<tr>
<td>--- But Not ---</td>
<td></td>
</tr>
<tr>
<td>CPU: 6502</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: The RABBIT interface
The query description in the description pane is composed of a set of one or more required
descriptors followed by zero or more prohibited descriptors. The required descriptors specify
properties which the target instance must have, while the prohibited descriptors specify properties
which the target must not have. (Chapter 9, section 1.3 considers the case in which the target
should have the required descriptors instead of must have.) We use the graphic "--- But Not ---"
to separate the required descriptors from the prohibited descriptors. (See the description pane in
Fig. 3.1.) [The term attribute shown in the pane actually corresponds to the term descriptor in the text. We chose to
display the word attribute instead because we felt that attribute is a more meaningful term than descriptor.]

A descriptor of a query description is itself either an instance class or an attribute-value constraint
pair. An instance class, which is shown in boldface on the workstation screen and in the text of
this thesis, denotes a general class of instances to which the target instance must belong, whereas
an attribute-value constraint pair specifies a property which the target must have. In particular,
the target must have that attribute and a value for that attribute which satisfies the value
constraint. An attribute-value constraint pair has two parts, separated by a colon—the name of the
attribute followed by a constraint on the kinds of values which can fill that role. For example, in
Fig. 3.1 the query description in the description pane indicates that the target instance must,
among other things, belong to the instance class 'Computer', be manufactured by Xerox or
Apple, and have a Large-Format display.

The structure of the description of an (actual) instance in the database (i.e., an image of an
instance) is similar to that of the description of a target instance with a few exceptions. One
difference is that there are no prohibited descriptors in the description of an actual instance.
Another is that analogous to the instance classes and attribute-value constraint pairs comprising a
query description are instance classes and attribute-value pairs which comprise the description of
an instance. An attribute-value pair specifies a relationship between that instance and other
instances, called the values, or fillers, of that attribute. Like its counterpart, an attribute-value pair
has two parts, separated by a colon—the name of the attribute (relationship) followed by one or
more values. The null value is denoted by "---". In Fig. 3.1, the example instance 'Star-8011'
belongs to the instance class 'Computer' and has the attribute-value pair 'manufacturer:
Xerox', which relates the instance 'Star-8011' to the instance 'Xerox' by the
'manufacturer:' relationship.

2. Constructing a Simple Query Description

We will now demonstrate how a user creates a simple query. (Chapter 6 contains a more detailed
description of the usage of the various commands supported in RABBIT.) For example, suppose
we wanted to find a Xerox or Apple computer equipped with a bit-map display and a processor
not based on the 6502 microprocessor. [The sample database contains some factual information, but much of it
Initially, the partial description consists solely of the single descriptor 'Entity'. All the instances in the database are considered to be entities, so this description is completely general. Let us suppose that the initial example in the instance pane is the instance 'Shugart', which is an example of a business (see Fig. 3.2).

Since we wish to find a computer, we begin by selecting 'Entity' (in either the partial description pane or the instance pane) with the mouse. The mouse has three buttons on it, one of which, when pressed, causes the RABBIT interface to display a menu of the possible operations which can be performed on the selection (see Fig. 3.3). Since 'Entity' is too broad an instance class, we need to find out what are some of its specializations. Selecting the operation specialize with the mouse makes RABBIT respond with a menu of the specializations of the selection, 'Entity' (see Fig. 3.4). Of those specializations, the instance class 'Product' seems to be the best term for describing a computer, so we select it. Now we are shown another menu of operations and select require, which adds 'Product' to our partial description (see Fig. 3.5).

At this point we can tell RABBIT to retrieve an example instance which matches the partial description constructed so far by pressing a button on the mouse, which causes RABBIT to display a menu containing the operation retrieve. After we select retrieve, RABBIT finds all instances in the database matching the current partial description ('Entity' 'Product') and lists their names in the matching examples pane. [If the set of matching instances is large, only a subset is retrieved.] Then an image of the first instance in that list, 'Atari-400', is displayed in the example pane (see Fig. 3.6). Now we can continue building up our partial description using the descriptors of this new example instance.

One of the descriptors of 'Atari-400' in the example pane is 'Computer', which is exactly what we want. So we can select that descriptor and use the require command to add 'Computer' to our partial description.

The manufacturer of the example is 'Atari', whereas we wanted the manufacturer to be either 'Xerox' or 'Apple'. We would really like to find out what are some of the other manufacturers besides 'Atari', with the hope that two of them are 'Xerox' and 'Apple'. To do this, we select the attribute-value pair, 'manufacturer: Atari', use the mouse to get a menu of operations, and select the operation alternatives. RABBIT then displays a menu of alternative manufacturers, two of which are indeed 'Xerox' and 'Apple'. After selecting both 'Xerox' and 'Apple', we get another menu of operations and select require any (see Fig. 3.7).

None of the other descriptors look very useful to us, so once again we use the mouse to tell RABBIT to perform a retrieval. This time, RABBIT retrieves 'Apple-II-plus' as the example instance (see Fig. 3.8).
The image in the example pane now has a descriptor which we do not want, namely, 'CPU: 6502'. To specify that restriction, we select that attribute-value pair and the operation prohibit, which now constrains all retrieved instances to have a processor other than the 6502 microprocessor. Suppose, however, that we do not know whether a TV display is bit-mapped or not. We can select the attribute-value pair 'display: TV' and the menu operation describe, which displays a new window on the workstation screen containing a description of a TV (see Fig. 3.9). One of the descriptors in the image is indeed 'type: Bit-map', so we select and require it with the mouse (see Fig. 3.10). If we now press a mouse button in the gray space between panes, we get a menu of operations, one of which will merge the current partial description ('Disk' 'type: Bit-map') with the original partial description of a computer in such a way that the computer retrieved will now be required to have a bit-map display. Note that we have created a description to be the value constraint of an attribute.

We have now finished specifying our partial description and can do another retrieve, with the resulting display shown in Fig. 3.11. We can stop here, or if we decide that the current partial description is still not quite right, we can remove old descriptors from or add new descriptors to the partial description and thus, continue the iterative reformulation of our query description.
Figure 3.2: The initial display
Figure 3.3: The menu of operations
Figure 3.4: The specializations of Entity
Figure 3.5: Requiring the instance class Product
### Attributes of Atari-400

<table>
<thead>
<tr>
<th>Entity</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td></td>
</tr>
<tr>
<td>OEM-Product</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RetailProduct</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: The Atari-400</td>
</tr>
<tr>
<td>cost: $600, $850</td>
</tr>
<tr>
<td>manufacturer: Atari</td>
</tr>
</tbody>
</table>

---

**Example**

---

**Previous Description**

---

**29 Matching Examples**

---

Atari-400
Atari-800
TRS-80
Apple-II-plus
Cromemco-Z-80
Star-8011
Star-8012
6502
68000

---

Figure 3.6: Retrieving a new example instance
Figure 3.7: Requiring the manufacturer to be Xerox or Apple
<table>
<thead>
<tr>
<th>Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>Attributes of the query</td>
</tr>
<tr>
<td>Computer</td>
<td></td>
</tr>
<tr>
<td>manufacturer: Apple, or Apple</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Previous Description</th>
<th>3 Matching Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td></td>
</tr>
<tr>
<td>Product</td>
<td></td>
</tr>
<tr>
<td>Computer</td>
<td></td>
</tr>
<tr>
<td>manufacturer: Apple, or Apple</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OEM-Product</th>
<th>RetailProduct</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: The Apple-II-plus</td>
<td></td>
</tr>
<tr>
<td>cost: $1400, $2,040</td>
<td></td>
</tr>
<tr>
<td>manufacturer: Apple</td>
<td></td>
</tr>
<tr>
<td>disk: TV</td>
<td></td>
</tr>
<tr>
<td>display: TV</td>
<td></td>
</tr>
<tr>
<td>memory: Apple-48K</td>
<td></td>
</tr>
<tr>
<td>CPU: 6502</td>
<td></td>
</tr>
<tr>
<td>operatingSystem: AppleSoft</td>
<td></td>
</tr>
</tbody>
</table>

Information about Apple-II-plus

Figure 3.8: Retrieving another example instance
<table>
<thead>
<tr>
<th>Example</th>
<th>Previous Description</th>
<th>3 Matching Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display</td>
<td>TV</td>
<td>Large-Format-Dis</td>
</tr>
<tr>
<td></td>
<td>Concept-100</td>
<td></td>
</tr>
</tbody>
</table>
| Attributes of TV | Display | Attributes of the query-
| cost: $100, $150 |          | Display |
| manufacturer: Sony |      | Attributes of the query-
| name: TVScreen | |          |
| resolution: 300x400 | |          |
| type: Bit-map | |          |
| Information about TV | |          |

Figure 3.9: A sub-window for describing a display
Figure 3.10: Specifying a bit-map display
Figure 3.11: The completed query description
Chapter 4 - Retrieval by Reformulation

The basic principle underlying RABBIT is a new paradigm for information retrieval—retrieval by reformulation. In brief, the user retrieves his target instance by iteratively reformulating his query, expressed as a description of the target instance. To assist him in the construction of that query, RABBIT provides an example instance, which serves as a template and counterexample. Furthermore, RABBIT presents information in the database from well-defined perspectives, which control how much information is displayed to the user and in what directions the user can move within the search space of possible queries. In particular, the perspective for viewing the example instance is inferred from the query description.

This paradigm has been developed starting from ideas borrowed from cognitive science, and in particular, a theory of human remembering. The two techniques of human remembering which RABBIT incorporates are descriptive retrieval and retrieval by instantiation. Our hypothesis was that the methods which people use in remembering could also be applied to the problems of retrieving information from databases. In addition, we felt that an interface employing those methods would be, in some sense, "natural" to use since the interface would be relying on techniques which the users themselves use in recalling thoughts from their own memories.

A second major source of ideas for this paradigm was the KL-ONE epistemology for representing knowledge [Brachman 79]. For reasons to be discussed in the next chapter, KL-ONE was an excellent environment for exploring the ideas borrowed from human psychology.

1. Descriptive Retrieval

One theory of human remembering postulates that people retrieve information from memory by iteratively constructing a description of the target item. Bobrow and Norman [1975] have suggested that the units of memory refer to one another through the use of context-dependent descriptions. They further postulated [1979] that

\[
\text{retrieval starts with a description of the desired information as an initial specification of the records sought from memory. This retrieval description guides the memory search process and helps determine the suitability of retrieved records for the purpose of the retrieval. The initial description can be modified as intermediate information becomes available during the retrieval cycle. [p. 107]}
\]

Williams and Hollan [1981] have extended the idea of accessing memory through descriptions to include the notions of iteration and reconstruction.
Information about the target item is used to construct a description of some aspect of the item. This description is used to recover a fragment of information about the item which is added to what is known. From this information a new description is formed to retrieve still more information, until the particular piece of information sought can be recovered. [p. 118]

According to Williams and Hollan, the retrieval cycle consists of three stages: finding a context (which simplifies the problem of forming a description), searching the memory space accessible from the combination of that context and the available information (the description), and verifying the information retrieved against the original query. “If the information retrieved satisfies the original query, the retrieval terminates at this point. Otherwise, the retrieved information is used to reformulate the description and a new cycle is initiated.” [p. 110]

In addition to being iterative, the retrieval process is also recursive [Norman 79, Williams 81c]. For example, finding a context may itself require one or more retrieval cycles. Similarly, during the course of a search, information may be found which is incomplete and thus, requires further search before it can be understood. The verification phase may also require retrieval cycles during the comparison of the retrieved information against the original query.

In summary, it has been theorized that people retrieve information from memory by iteratively constructing partial descriptions of the desired target item. After a partial description has been specified (which includes finding the context for that description), a search is made for information matching the description. If the retrieved information does not satisfy the original query, then that information is used to reformulate the description, and the cycle is repeated. Furthermore, each stage of the cycle can itself be recursive.

2. Retrieval by Instantiation

The technique of descriptive retrieval outlined above is a general paradigm for retrieving information, but it does not indicate what the fragments of information retrieved on each cycle are or how they are incorporated into the partial description to make a new description. Williams [1981a] has conjectured that the information retrieved on each retrieval cycle is in the form of an instantiation, a description of an example item suggested by the partial description. This idea is based on the observation that when people are trying to recall something, they frequently are reminded of items which are similar to or related to their target items.

The instantiation serves as a template for the description of the target item by providing a set of descriptors which can be incorporated into the partial description. In addition, for those descriptors which represent relationships (which we have been calling attributes), the instantiation also provides access to the possible values filling those attributes by giving example values, namely, the values filling those attributes in the instantiation.
Williams [1981a] goes on to propose a possible database interface based on the ideas of descriptive retrieval and retrieval by instantiation in which the user constructs a description of his target item using an instantiation provided by the computer. We have used Williams' proposal as a starting point for our work.

3. Retrieval by Reformulation

Retrieval by reformulation is our paradigm for information retrieval; it combines descriptive retrieval with retrieval by instantiation and some additional ideas of our own. The four basic components of retrieval by reformulation are: (1) retrieval by constructed descriptions, (2) interactive construction of queries, (3) critique of example instances, and (4) dynamic perspectives.

3.1 Retrieval by constructed descriptions

The user makes a retrieval by constructing a description of the desired instance. Our expectation is that if people do, in fact, use descriptions to access their memory, then an interface in which users create descriptions to access an electronic database will be "natural" to use since the interface will be acting as an extension of their own thinking processes. In addition, if people think in terms of descriptions, then users should find it easier to translate their conceptual descriptions into descriptions understandable by the computer than to translate their thoughts into a more "foreign" representation, such as boolean keyword expressions or predicate calculus statements (which introduce explicitly special operators such as $\land$, $\lor$, $\neg$, $\forall$, and $\exists$).

Descriptions also have the property that they are "multi-dimensional"—each attribute of a description represents a dimension along which that description can be extended. (For example, if restaurants have attributes cuisine, location, and personnel, then each of those attributes represents a different dimension for describing the target instance.) Furthermore, more than one attribute can be changed at a time, so a query can proceed much more quickly and directly than if only one attribute could be changed between retrieval cycles. As a result of this multi-dimensional nature, the user has a broad frontier of possible directions in which the query can be pushed. Since he is not pursuing a single direction, there is a smaller chance that he will get lost. Furthermore, by being able to approach the item being sought from more than one direction, there is a greater chance of reaching that item or at least, some item similar to the target item. This flexibility in creating descriptions also has its drawbacks. For example, there is also an increased chance of the user's overconstraining his query description such that no item matches. Thus, backtracking facilities need to be provided in addition to the query construction operations.
3.2 Interactive construction of queries

The key idea here is that information retrieval is really a process of continual reformulation of the original query. Attempting to formulate a "correct" query on the first try is usually a difficult task that requires much effort; we suggest that a much better method is for the user to devise an initial, partial query and then gradually refine it into a better, more complete one using the information generated by the computer (or memory, in the case of human information retrieval) when it was presented with the partial query. In such a situation, the user does not have to plan out his query in careful detail; after specifying some initial query, he can then proceed to reformulate that query, using information provided back by the interface, until he retrieves his desired information. This ability to reformulate a query is especially useful if the user's initial query description is incomplete or even wrong (in the sense that the user's query description is not a correct description of his intent).

This interaction can be viewed as an exercise in problem solving, which is the way Williams and Hollan [1981] view human remembering. Given an initial problem (namely, specify a description which uniquely determines his target items), the user proposes a possible solution (the initial partial description), and then proceeds to reformulate that solution using feedback from the computer.

Another way of treating this interaction is to think of it as a conversation between two subjects, A and B, where A is trying to explain something to B. As A begins to describe to B what he is thinking, B develops a picture in his mind of what he thinks A is trying to say. B can then tell A what he thinks A has been saying. This feedback allows A to decide whether he can end his discourse (because B's mental image now matches A's mental image) or whether he must reformulate his explanation (because their mental pictures still differ) (see [Oddy 77] for a similar characterization). Furthermore, the nature of B's response can indicate to A which areas of A's explanation need to be better articulated.

3.3 Critique of example instances

The information which the computer provides back to the user upon receipt of the user's partial description is in the form of a description of an example instance, called an image (Williams uses the term instantiation). This instance is an instance (item) in the database which matches the user's partial description. The matching need not be an exact one; for example, in human memory it is often analogical or metaphorical [Bobrow 75]. However, for this thesis we are only considering the simpler case of an exact match.

The example instance is useful in several respects. For one thing, the image of the example instance serves as a template for use in further describing the user's target instance. Since the
example matches the user's description of his target instance, many of the descriptors of the image will be appropriate for further describing the target instance. Thus, the image of an example instance is a partial solution to the problem of chapter 2, section 1; it tells a person what the appropriate terms (which we have here been calling descriptors) for specifying his query are. The image also gives the user example values for those descriptors which denote relationships, via the values filling those descriptors (called attribute-value pairs) in the image. Since the attribute and its value are presumably strongly related, knowing the meaning of one should provide a good hint as to the meaning of the other. In short, example values help the user understand the meanings of the terms used in making query descriptions.

In addition to its use as a template, the image also provides a person with access to alternatives to the descriptors of the image. RABBIT supports this capability by providing operations which allow the user to examine alternative descriptors (e.g., some alternatives to 'cuisine: French' are 'cuisine: Chinese' and 'cuisine: German'). This ability was motivated by the theory of human remembering discussed earlier, which proposes that retrieval is also recursive; during the course of a retrieval, a person may recall information which is incomplete and thus, he may decide to perform an extended (recursive) retrieval using that information as the (new) starting point. The ability to find descriptors which are similar to or related to the original descriptors of the image is a means for dealing further with the problem of incomplete knowledge about the appropriate terms. It is unlikely that all the descriptors of the image are directly applicable to the user's partial description, but it is likely that the terms which the user needs to specify further his query are related to the descriptors in the image.

As we noted earlier, the image is a description of an example instance, an actual instance in the database which instantiates the abstract specification of the query description. But perhaps more importantly, the example instance serves as a counterexample. Although the retrieved example instance matches the user's partial query description, it is very unlikely, at first, that the example is the instance for which he is searching. In that case, the example instance is really a counterexample to his intentions, a near-miss recollection.

Viewed as a counterexample, the instantiation makes the user aware that his query description is still incomplete and in addition, may point out to him some of the areas and directions in which the description needs to be further refined. (For a somewhat different use of counterexamples see Winston [1977].) For example, if the user is given a (counter)example whose value for a given attribute is different from the value he had intended, then he now knows that he should reformulate his partial description to include that attribute with the particular value(s) he desires. Here, the counterexample is forcing the user to make explicit the implicit assumptions he had been making about the value of that attribute.

This technique of prodding the user to articulate more completely his partial description, particularly, the values filling the attributes, is probably most effective when the counterexamples
are “extreme”; i.e., the example instances are maximally dissimilar from the “typical” instance matching the partial description, for example, because of extreme or uncommon values filling the attributes. In those cases, the differences between the current partial description and the “correct” description of the target instance are most evident. A specification by the user of the desired value(s) will now sharply reduce the search space since instances in the database with attributes filled by values outside the specified range will no longer be considered in subsequent retrievals.

In addition, a counterexample can serve to make the user aware that his partial description is incomplete. After constructing a description, the user may feel that the description is now complete and not realize that that description may still be too general. “[People] tend to examine facts that can only confirm their hypotheses rather than choosing to examine facts capable of disconfirming their hypotheses” ([Wason 72] as cited in [Thomas 77]). Williams [1981b] refers to this problem as “functional fixedness” on the part of the user; after specifying all the attributes of his target that he can think of, the user may feel that he is done. But now if a subsequent retrieval produces a counterexample, then he is forced to realize that there are additional attributes which need to be included in (or removed from) the partial description.

In summary, the image of the example instance performs the following functions:

1) It serves as a template of descriptors.
2) It provides access to alternative descriptors.
3) It gives the user a concrete example of an actual instance in the database.
4) It serves as a counterexample that indicates that parts of the partial description need correction and in addition, specifies which parts those are.

3.4 Dynamic perspectives

The fourth important characteristic of retrieval by reformulation is the idea of “dynamic” or “constructed” perspectives. Norman and Bobrow [1979] characterize a perspective as something which “provides a way of describing an event or item from a particular viewpoint.” [p. 113] In RABBIT a perspective specifies which descriptors (instance classes and attribute-value pairs) should be included in the image of the instance presented to the user. For example, a particular restaurant, say “The Little Hsi Nan,” when viewed from the perspective of “a restaurant,” has descriptors ‘cuisine: Chinese’ and ‘reservations: Unnecessary’, but from the perspective of “a business,” it has descriptors ‘location: PaloAlto’ and ‘manager: Louie’. In addition, we have extended that notion of perspectives to include change; the perspective from which the user views the instances in the database changes depending upon his partial description and on where he is within the database. We use perspectives as a mechanism for controlling the presentation of instances in the database, for preventing the formation of semantically anomalous queries (by controlling the space of the database immediately accessible to the user), and for dealing with the
non-uniformity of the database (by presenting to the user only “relevant” information). A more detailed explanation and discussion of perspectives is deferred to chapter 7.

Fig. 4.1 shows a diagram portraying the interrelationships between the partial description, the perspective, the database, and the user. The database can be thought of as a lattice in which the leaves are the instances (i.e., the items of immediate interest to users) while the rest of the structure represents organizational information. The set of instances in which the user is currently interested is delimited by the partial description being constructed by the user. The partial description also determines the perspective from which instances in the database are viewed. A perspective is like a filter over a window into the database; the only way to see instances in the database is through some perspective. When the user tells RABBIT to retrieve an (example) instance, RABBIT presents an image of that instance from a perspective inferred from the partial description. A given instance may be seen from different perspectives and thus, can have different images. Similarly, the same perspective may be used to view several instances.

4. The Example Revisited

Now, we will re-examine the example given in Chapter 3 and show how retrieval by reformulation is used in the specification of that example query.

The initial partial description (see Fig. 3.2) was 'Entity', so the matching instances were all instances in the database. The particular instance 'Shugart' was chosen at random (as far as the user is concerned) from the database to be the initial example instance. The perspective from which the example was presented was determined by the partial description. In this case, 'Shugart' was described from the perspective of "an Entity"; only those attributes common to all entities were shown in the image. (In our database, the only attribute shared by all entities is 'name:'). In addition, certain instance classes peculiar to 'Shugart' were also shown, namely, the instance class 'Business'. (The details of how perspectives are produced can be found in chapter 7, sections 1.1 and 1.3.)

The target of our retrieval was a Xerox or Apple computer equipped with a bit-map display and a processor other than the 6502 microprocessor. Computers are entities, but so are many other things (and in fact, everything); what we really needed to do was to reformulate our partial description. Since 'Entity' is such a broad instance class, it was very likely that it had (more restrictive) subclasses, of which one or more might have included computers. When we examined the specializations of 'Entity' to get 'Product', we were, in effect, doing an "extended retrieval" on the descriptor 'Entity', to find instance classes which were subclasses of the given instance class.
Figure 4.1: The interrelationships between the user and the database
After we added the concept 'Product' to the partial description and did a retrieval, RABBIT found a new example instance (see Fig. 3.6). Since the partial description had now changed, the perspective for viewing instances had also changed. In particular, we were shown the example instance 'Atari-400' viewed as both "an Entity" and as "a Product." Note that 'manufacturer:' and 'cost:' are attributes which all products have but which are not attributes of all entities.

Two descriptors of the image, 'Computer' and 'manufacturer: Atari', seemed appropriate for incorporation into our partial description. 'Computer' was immediately applicable, but 'manufacturer: Atari' was the "right" attribute with the "wrong" value. Once again, we performed an extended retrieval on that attribute to find the alternative values, 'Xerox' and 'Apple', which we wanted.

None of the other attributes seemed relevant to our target description. The reason for this was that the remaining properties in which we were interested, namely, the display and the processor, are not properties of all products but only of computer products. However, since we added the descriptor 'Computer' to our partial description, a subsequent retrieval revealed the descriptors peculiar to computers, as can be seen in Fig. 3.8. Now, we had access to the display and processor properties via the 'display:' and 'CPU:' attributes, respectively.

The CPU in the image was exactly what we did not want, so we prohibited it, but we were unsure if the type of display possessed by the example instance (a TV) was what we desired. So we performed an extended retrieval in order to learn what were the attributes of a TV display and in order to describe what kind of display we wanted. That last operation completed our query specification.
Chapter 5 - The KL-ONE Database

The database which RABBIT accesses is a KL-ONE network. KL-ONE [Brachman 79] is a knowledge representation language developed at BBN. The particular dialect of KL-ONE which we are using is KloneTalk [Fikes 81b], an implementation of KL-ONE in the Smalltalk programming language [Ingalls 78]. KL-ONE has proved to be a formative environment for exploring our ideas concerning description-based retrieval.

1. The KL-ONE Epistemology

KL-ONE is actually two sublanguages—a description language and an assertion language. The description language allows one to create “description terms” and assemble them into (KL-ONE) descriptions; the assertion language is used to make statements about the world using those description terms. Our database presently makes use of the description language only.

The description terms of KL-ONE are called concepts. There are two kinds of concepts—generic and individual. Generic concepts express generic knowledge (i.e., they describe classes of objects) and are organized into a specialization structure in which a given generic concept can have multiple superconcepts and subconcepts. The topmost element in this lattice of generics is the generic concept ‘Entity’. ‘Entity’ is the only generic concept with no superconcept; all other generics are specializations (i.e., subconcepts) of it or of one of its specializations. (In this thesis we will show generic concepts in boldface.)

Individual concepts are unique instances of some generic concept. An individual concept may be an individual of more than one generic while a generic may have zero or more individuals. For example, the individual concept ‘Boston’ could be an individual of the generic concept ‘City’, which in turn could be a specialization of the generic concept ‘Entity’. If ‘A’ is an individual of ‘B’ and ‘B’ is a subconcept of ‘C’, then ‘A’ is also an individual of ‘C’. So, in the preceding example, ‘Boston’ is an (indirect) individual of ‘Entity’ although it is not an immediate individual of ‘Entity’. We will use the phrase “an individual of” to refer to the transitive closure unless we explicitly state otherwise.

Both generic and individual concepts have an internal structure composed of roles and structural descriptions. The meaning of a concept is determined from the meanings of its superconcept(s) in the specialization structure, its roles, and its structural descriptions.

A role describes a relationship between the concept owning that role and other concepts in the
network, called the values, or fillers, of that role. Each role can have a set of values (which are individual concepts). Other kinds of information that can be associated with a role include constraints on the type of values (the role's value restriction, which is a generic concept) and on the number of values. An important property of roles is that they are inheritable. Each concept has a collection of local roles, which are part of that concept's definition. But in addition, a given concept also inherits the roles of all its superconcepts. The names of roles are not unique across all concepts; thus, the meaning of a role name depends upon the concept owning that role.

Structural descriptions are constraints on the values that can fill different roles of the same or different concepts. For example, a particular structural description could specify that two roles of a concept must have the same filler.

Fig. 5.1 shows an example KL-ONE network. 'Entity' has three subconcepts: 'Business', 'Product', and 'ElectricalDevice'. 'Computer' has two superconcepts: 'Product' and 'ElectricalDevice'. Two other generic concepts in the lattice are 'Restaurant' and 'Manufacturer', both of which are subconcepts of 'Business'. These generic concepts have various roles. For example, 'Business' has roles 'location:' and 'income:'. 'Star-8011' is an individual of 'Computer' and consequently, it inherits all the roles of its ancestors in the KL-ONE network. The figure shows that 'Xerox' is the value of the 'manufacturer:' role of 'Star-8011' and that 'Manufacturer' is the value restriction (V/R) of the 'manufacturer:' role of 'Product'.

2. KL-ONE: A Medium for Representing Heterogeneous Data

One of the main reasons for choosing KL-ONE is that the KL-ONE epistemology easily supports our conception of databases as being non-uniform. The KL-ONE taxonomy allows very disparate information to be classified in a lattice structure. This classification scheme is a lattice and not a tree since "real world" information can seldom be classified according to a strict hierarchy. KL-ONE concepts allow uniform information to be gathered together in one place while the role inheritance mechanism allows that information to be shared. This ability to distribute information and control the inheritance of that information is what enables KL-ONE to support non-uniformly structured databases. So, for example, heterogeneous classes like Business, Product, and Person, and their subclasses, such as Manufacturer, Retail Business, and Computer, can all be represented in the same database. Instances of those classes, i.e., particular businesses, products, and persons, contain only that information which distinguishes them from other instances in the same class. In effect, we have a classification scheme consisting of interrelated taxonomies. Movement from one class or instance to another is easily made by following the (specialization, role value, and role value restriction) links which connect all the objects in the network.
Figure 5.1: A piece of KL-ONE taxonomy
3. The Representation of RABBIT Instance Descriptions in KL-ONE

As we saw in chapter 3, instance descriptions (called images) are simply collections of descriptors, where a descriptor is either an instance class or an attribute-value pair. The correspondence between KL-ONE concept descriptions and RABBIT instance descriptions is straightforward. The instance classes comprising RABBIT descriptions are represented by KL-ONE generic concepts, while attribute-value pairs are implemented as KL-ONE role-value pairs. The instances themselves correspond to individual concepts.

An instance is described by a particular instance description if and only if the following are true:

1. The individual concept (call it ICI) corresponding to that instance is an individual of the generic concepts corresponding to the instance classes in the instance description (i.e., the instance is a member of all the instance classes given in the instance description).
2. For every attribute of each attribute-value pair in the instance description, ICI has the corresponding role (i.e., the attributes given in the attribute-value pairs are all attributes of the instance).
3. For each attribute-value pair, the individual concept(s) corresponding to the value(s) given for that attribute are fillers of the corresponding role of ICI.

4. The Representation of RABBIT Query Descriptions in KL-ONE

A consequence of KL-ONE's strong epistemology for representing descriptive information is that there is also a natural correspondence between (most of) the query descriptions of our paradigm and KL-ONE concepts, which are themselves (partial) descriptions. In most cases, a KL-ONE concept can be found in the network (or a new one added to the network) which describes exactly those objects matching the query description. In those cases, retrieval of instances matching the query description is equivalent to finding all individuals of that possibly new concept.

The present implementation of RABBIT does not actually construct these new concepts given a query description, but the existence of such concepts presents several interesting possibilities. For example, in future implementations the user could be allowed to name and save his query description as a new (generic) concept in the network for future use. Moreover, if new individual concepts are subsequently added to the network using a KL-ONE classifier [Lipkis 81], then those individuals will be subsumed by that constructed concept (i.e., those individuals will become individuals of that concept) if and only if they match the corresponding query description.

Before we discuss further the correspondence between query descriptions and KL-ONE concepts, we will first describe more precisely the structure of a RABBIT query description.
4.1 The structure of query descriptions

In chapter 3 we defined a query description to be a collection of required and prohibited descriptors, where a descriptor is either an instance class or an attribute-value constraint pair. More precisely, a query description is an implicit boolean combination of descriptors. In its simplest form, the “required” section of a query description (i.e., the required descriptors) consists of the (logical) AND of one or more instance classes, ANDed with the AND of zero or more attribute-value constraint pairs. Similarly, the “prohibited” section (i.e., the prohibited descriptors) is generally the AND of zero or more instance classes, ANDed with the AND of zero or more attribute-value constraint pairs. A complete query description is then the AND of the required section and the prohibited section, where the prohibited section is optional. (For several examples of query descriptions, see chapter 6.)

We decided not to consider a description consisting solely of a prohibited section to be a valid query description although one could create that effect by constructing a query description whose only required descriptor is the instance class ‘Entity’. The reason for this is that in a large heterogeneous database, a query based only on an enumeration of properties which the target instance should not have is not a very efficient or effective way of finding instances. Thus, we wished to discourage, but not prohibit, such a strategy.

AND is the principal connective between descriptors, but a limited form of (logical inclusive) ORing is also available. In particular, any occurrence of an instance class in the above framework of ANDed instance classes can be replaced with the OR of two or more instance classes. For the case of attribute-value constraint pairs, the value constraint can be a list of one or more (ORed) instances, or a description. And finally, any occurrence of an instance class can be replaced with a description. These descriptions which can replace an instance class or constitute the value constraint of some attribute-value constraint pair are themselves query descriptions, so the definition of a query description is recursive. We refer to these “sub-descriptions” as embedded descriptions.

A query description is interpreted to specify a set of instances in the database; each of those instances is said to match the description. An instance matches a query description if the following are true:

(1) For each ANDed instance class in the query description, the retrieved instance is a member of that class.
(2) For each group of ORed instance classes, the retrieved instance is a member of at least one of those classes.
(3) For each ANDed instance class in the prohibited section of the query, the retrieved instance is not a member of that class.
(4) For each group of ORed instance classes in the prohibited section, the retrieved instance is not a member of any of those classes.

(5) For every attribute-value constraint pair in the query, the retrieved instance has that attribute and a value for that attribute which satisfies the value constraint. Note that the names of attributes are not unique across all instance classes, so two attributes having the same name but belonging to two different instance classes may not match because of differing semantics.

(6) For every attribute-value constraint pair in the prohibited section of the query, the retrieved instance has that attribute [the reason for this will be given in chapter 7], but none of the values filling that attribute satisfy the value constraint.

So, for example, if the query description is

\[
('\text{Entity}') \\
and ('\text{Business}') \\
and ('\text{ChineseRestaurant' or 'FrenchRestaurant}') \\
and ('\text{location: PaloAlto or MountainView}') \\
and ('\text{payment: Visa}') \\
and ('\text{payment: MasterCharge}') \\
and not ('\text{reservations: Necessary}')
\]

then an instance matches that description if it is an entity, a business, and either a Chinese restaurant or a French restaurant, is located in Palo Alto or Mountain View, accepts both Visa and MasterCharge, and does not require reservations.

Given the structure of descriptions as described above, we will show in the following sections that, in most cases, a KL-ONE concept description can be created in the network which subsumes exactly those individual concepts which correspond to the instances specified by the query description.

4.2 Representing instance classes in KL-ONE

Suppose we have two instance classes 'A' and 'B', which have corresponding generic concepts 'A' and 'B'. A given query description might contain the AND of those two instance classes, the OR of those two instance classes, or the AND of one with the NOT of the other. The equivalent KL-ONE structure is the AND of the corresponding concepts, the OR of the two concepts, and the AND of one concept with the NOT of the other.

To form the generic concept corresponding to the AND of the two concepts 'A' and 'B', we need to create a generic concept, 'A-and-B', which is a subconcept of both concepts 'A' and 'B', and then make individual concepts which are individuals of both 'A' and 'B' be individuals of 'A-and-B' (see Fig. 5.2). By inheritance (actually, the transitivity of "specialization of"/"individual of" links), any individual of 'A-and-B' is both an individual of 'A' and an individual of 'B'.
Figure 5.2: ANDing two concepts

Figure 5.3: ORing two concepts
Similarly, the OR of 'A' and 'B' can be constructed by finding the first common ancestors, say 'X' and 'Y', of 'A' and 'B' (one such common ancestor must exist since 'Entity' is the ancestor of all concepts) and creating a new concept 'A-or-B', which is a subconcept of those common ancestor concepts and superconcepts of both 'A' and 'B' (see Fig. 5.3). We define the first common ancestors of 'A' and 'B' to be those concepts which are superconcepts of 'A' and 'B' and which have no subconcept which is also a superconcept of 'A' and 'B'. But there is a slight problem here since we could still create an individual which is an individual of 'A-or-B' but not an individual of either 'A' or 'B'. A solution is to define a KloneTalk decomposition. The subconcepts of a given concept can be divided up into partitions (the decompositions) which are specified by the network designer to be complete/incomplete and disjoint/non-disjoint. A complete decomposition is a group of subconcepts of a given concept such that any individual of the given concept is also an individual of at least one subconcept in the decomposition. A disjoint decomposition of a concept is a group of subconcepts such that each individual of the concept is an individual of at most one subconcept in the group. In this case, we define a complete decomposition of 'A-and-B' which contains the subconcepts 'A' and 'B'. Now, all individuals of either 'A' or 'B' are also individuals of 'A-or-B' and similarly, all individuals of 'A-or-B' are individuals of 'A' or 'B' or both.

The representation of NOT 'A' is a bit more difficult since negated descriptors specify which instances do not match as opposed to which instances do match. The method for creating NOT 'A' is as follows. Suppose 'A' has only one superconcept, 'X'. Then create a new subconcept of 'X' named 'X-ButNot-A', and a new decomposition, complete and disjoint, which contains both 'A' and 'X-ButNot-A'. Finally, make individuals of 'X' which are not individuals of 'A' into individuals of 'X-ButNot-A' (see Fig. 5.4a). Since the decomposition is both complete and disjoint, any individual of 'X' which is not an individual of 'A' must be an individual of 'X-ButNot-A'. If 'A' had happened to have two superconcepts, repeat the preceding process for each superconcept and then create the OR of the resulting new concepts (see Fig. 5.4b). Note that we have not created 'Not-A', the concept whose individuals are all individual concepts which are not individuals of 'A', but rather, the relative complement of 'A'. The operations for constructing partial descriptions in RABBIT are defined such that the user can not add NOT 'A' (i.e., add the instance class 'A' as a prohibited descriptor) to a query description without first adding a superclass of 'A' to the description as a required instance class (see chapter 6). So there always exist some superclass 'X' of 'A' in the query description. The reason for constraining the user to specify 'X' (say, 'Product') before he can prohibit a subclass of 'X' (say, 'Computer') is that "not a computer" is not a very specific query description. For example, people and restaurants match the query description "not a computer," but it is unlikely that the user who created such a query description is searching for a particular person or restaurant. What is more likely to be the case is that the user is thinking of some context which includes computers, and the instance for which he is searching is something in that context which is not a computer.
Figure 5.4: Negating a concept

(a) $X$ \quad $X \text{-} \text{but-not-A}$

(b) $X \text{-or-} Y$  
\quad $X \text{-but-not-A}$  
\quad $Y \text{-but-not-A}$

Figure 5.5: Decompositions of Restaurant
4.3 Representing attribute-value constraint pairs in KL-ONE

Attribute-value constraint pairs are composed of an attribute and a constraint on the values which are allowed to fill that attribute in the matching instance. The current implementation supports two kinds of constraints, represented as either a set of one or more instances or as a (recursively defined) embedded description. If the value constraint is a set of instances, then an instance matches a query description containing that attribute-value constraint if the individual concept corresponding to that instance has (or inherits) the role corresponding to that attribute and in addition, has a value (an individual concept) filling that role which corresponds to one of the instances enumerated in the value constraint. If the value constraint is an embedded description, then the role corresponding to that attribute must have a value filling that role which (recursively) matches the description comprising the value constraint. (More elaborate constraints are discussed in chapter 9.)

A KL-ONE value restriction for a role is similar to our notion of a value constraint for the corresponding attribute, but value constraints can not be implemented as value restrictions. The reason is that a value restriction is a constraint which must be satisfied by all fillers of a role, whereas a value constraint (as defined above) must be satisfied by at least one of the fillers of that attribute.

One way of examining this difference is to view a value constraint as a data structure containing two parts: a specification of the potentially valid fillers, and a relation which indicates how the actual fillers must be related to the potential fillers. If the value constraint is an enumerated list, then the first part is the list of instances (which can be represented by a concept whose individuals are those individuals corresponding to the instances in the list) while the second part is the relation “non-empty intersection.” A similar result is obtained if the value constraint is a description; in that case, the first part is that description (which can also be represented by a concept denoting those instances which match that description) and the second part is again the relation “non-empty intersection.” According to this view, a KL-ONE value restriction (which is a generic concept) has a first part which is the concept and a second part which is the relation “subset,” i.e., the actual fillers must be a subset of the potential fillers. So in some sense, KL-ONE value restrictions are a special case of this value constraint mechanism. One possible solution to the problem of representing value constraints in KL-ONE is to extend KL-ONE value restrictions to include our notion of value constraints.

But for the most part, constructing query descriptions is analogous to creating a new generic concept in the KL-ONE network. Instance classes, instances, and attributes have simple analogs in the KL-ONE description language, while value constraints in the query description can be viewed as a generalization of KL-ONE value restrictions.
5. Using Descriptive Information Embedded in KL-ONE

Since KL-ONE is a knowledge representation language, we are able to use some of the descriptive information which KL-ONE employs in its classification of knowledge. For example, KL-ONE maintains information regarding the type of information which can fill a role via the role's value restriction. By enforcing that type constraint in the interface to the user, we can prevent certain kinds of semantically anomalous descriptions from being formed. (This is discussed further in chapter 8.)

Another example is the KL-ONE mechanism for defining decompositions of concepts. A given concept may have an arbitrary number of subconcepts, but the network designer is allowed to partition those subconcepts, i.e., the designer can group related subconcepts together. For example, if 'Restaurant' has subconcepts 'ChineseRestaurant', 'FrenchRestaurant', 'BostonRestaurant', and 'CambridgeRestaurant', then one partition could contain those subconcepts of 'Restaurant' differentiated by the ethnic origin of the cuisine while another could contain the subconcepts differentiated by the location of the restaurant (see Fig. 5.5). The interface can use that knowledge in the following manner: if the user is seeing the description of a French restaurant and wants to know what are some of the alternatives to that kind of restaurant, RABBIT can initially show him alternatives within the same partition since the members of that partition are more similar to the term 'FrenchRestaurant' than other terms outside that partition.

Probably the strongest influence of KL-ONE has been in motivating and supporting the development of the idea of dynamic perspectives for viewing instances in the network/database. Perspectives are discussed in detail in chapter 7.

In general, KL-ONE was formative in developing the paradigm of retrieval by reformulation. It provided a firm foundation for the implementation of our abstract notion of query descriptions and moreover, posed a number of issues (such as the need to control the amount of information which should be displayed), and solutions (such as the use of dynamic perspectives).
Chapter 6 - The User Interface

The user interface of RABBIT was designed to support the ideas proposed in our paradigm of retrieval by reformulation. Consequently, the inclusion of most of the features, properties, and operations of the user interface was motivated by the kinds of abstract operations presented in the theory of human remembering discussed earlier. Another objective was to develop an interface which would be easy to learn and use. However, it should be kept in mind that the implementation was intended primarily for use as a test-bed of the ideas behind retrieval by reformulation.

1. The Goals

The design of the user interface was undertaken with several goals in mind. Firstly, the interface should be highly interactive; whenever the user invokes an operation which changes the (partial) query description, the workstation display should be immediately updated to reflect that change. Moreover, it should be clear to the user where those changes in the display occurred. Also, the partial description should at all times be visible to the user and accessible for modification. Secondly, the interface should provide a set of simple, but powerful, operations for creating and modifying descriptions. Thirdly, RABBIT should provide automatic semantic checking which would use some simple heuristics and knowledge about the KL-ONE network to help prevent users from constructing semantically improper query descriptions.

Section 2 discusses some of the details concerning the RABBIT display. The basic operations provided in RABBIT for (re)formulating query descriptions are described in section 3. The last section, section 4, covers some of the techniques which RABBIT uses to help maintain the semantic consistency of query descriptions.

2. The Display

As was mentioned in the example user session in chapter 3, the workstation display presents a window divided into four panes (see Fig. 6.1): the description pane, the example pane, the previous description pane, and the matching examples pane. The display of information in those panes is modelled after the Smalltalk system user interface.
Figure 6.1: The RABBIT interface
2.1 The description pane

The description pane contains the query description. The descriptors of the query description are displayed in a vertical listing with instance classes and the instances comprising value constraints placed on separate lines. The names of instance classes are shown in boldface while attribute-value constraint pairs are represented as the name of the attribute, followed by a colon, followed by the value constraint. The value constraint itself is either a description (see chapter 8) or a list of the names of one or more instances. Between each descriptor of the query description is an implicit AND which is not displayed. If the description contains any prohibited descriptors, then all those prohibited descriptors are grouped together at the end of the list of required descriptors and separated from the required descriptors by the graphic "--- But Not ---". For example, the partial query description in Fig. 6.2 describes an inexpensive French or Chinese restaurant located in either Palo Alto, Mountain View or Los Altos which accepts both Visa and MasterCharge and which does not require reservations. [The meaning of the term 'Expensive' is somewhat ambiguous, so that instance would probably have descriptors of its own indicating the actual range of prices and possibly a date to control for inflation.]

--- Attributes of the query ---
Entity
Business
StoreFrontBusiness
Restaurant
FrenchRestaurant, or ChineseRestaurant
location: PaloAlto, MountainView, or LosAltos
payment: Visa
payment: MasterCharge
--- But Not ---
price-range: Expensive
reservations: Required
--- Attributes of the query ---

Figure 6.2: An example of a query description

2.2 The example pane

The example pane is divided into two parts. The top half of the pane contains an image of an instance while the bottom half contains some purely textual information about that instance. Note that images do not contain ORed or prohibited descriptors. The values of an attribute-value pair are listed immediately after the attribute in an image, as in the case of the 'payment:' attribute in Fig. 6.3. [If an attribute has more than four values, then only the first four values are shown, followed by an ellipsis ( . . ).] The reason for putting an attribute-value pair on one line is so that it is clear what the distinct attributes of that instance are. As a result, the image looks more like a well-ordered template than like a collection of descriptors.
Attributes of LittleHsi-Nan

Entity
Business
StoreFrontBusiness
Restaurant
ChineseRestaurant
PaloAltoRestaurant
ame: TheLittleHsi-Nan
location: PaloAlto
cuisine: Chinese
region: Szechuan
payment: Visa, MasterCharge, PersonalCheck
price-range: Moderate
reservations: Unnecessary

Attributes of LittleHsi-Nan

Figure 6.3: An example of an image

Note that 'region:' is shown indented with respect to 'cuisine:'. The reason is that 'region:' is a "subordinate" attribute, i.e., an attribute which modifies another attribute, in this case, 'cuisine:'. The information that one attribute modifies another attribute can be represented in the KL-ONE network by the network designer. Such pairs of attributes are displayed together with the sub-attributes indented with respect to the main attribute.

The bottom portion of the example pane is generally used to display any textual information stored with the example instance. For example, in a restaurants database, that space might be used for reviews or ratings. Both the Smalltalk editor and interpreter can be invoked from within that half of the pane, so that area of the example pane could also be used to run simulations and execute Smalltalk programs.

2.3 Making selections

Selection of descriptors (and menu items) is made with a pointing device called a mouse. To make a selection, the user moves the mouse (which moves a cursor on the workstation display) until the cursor is positioned over the desired text. Then depressing a button on the mouse will cause the text beneath the cursor to be selected. Typically, selection of a descriptor (or menu item) on the screen is indicated by displaying in reverse video the line of the pane (or menu) containing the selected text (i.e., a black stripe is painted over the text; in Fig. 6.1, 'Computer' has been selected). In the description pane, the user can individually select an instance class, attribute-value constraint pair, a single value of a value constraint, or an embedded description. In the example pane, he can individually select an instance class or an attribute-value pair. (Access to the individual values of an attribute-value pair is provided by the alternatives command, described in section 3.3.) The user can also deselect a selection by reselecting it with the mouse. (A useful extension to the current implementation of selection is the ability to select several items (e.g., an ORed group of instance classes) at once. Chapter 9 discusses such a capability.)
2.4 Updating the display

The contents of the description pane are updated incrementally, each time a descriptor is added to or removed from the partial description. The contents of the remaining panes are updated only after a retrieval from the database. When the user asks the interface to make a retrieval based on his current partial description, RABBIT first copies the current partial description from the description pane into the previous description pane and then retrieves all instances in the database matching that description. A list of the names of those matching instances is put into the matching examples pane while a description of the first instance in that list is displayed in the example pane. Subsequently selecting any instance in the matching examples pane will cause the description of that instance to be displayed in the example pane. In effect, the user can browse through the descriptions of the matching instances. In accordance with the goal that changes in the display of descriptions should be apparent to the user, the attributes comprising the description of the newly retrieved instance are put into the same order that the corresponding attributes of the previously retrieved instance were in; those attributes of the new image not present in the description of the previous instance are appended at the end of the description of the new instance.

Since the partial query description is not changed by the act of selecting another instance in the matching examples pane, the perspective for viewing those instances is not changed. Thus, selecting another instance in the matching examples pane will not change what attributes are being shown in the example pane; only the fillers of those attributes will change. (This last remark will become clearer in the next chapter.)

3. The Basic Operations

3.1 Invocation of the operations

Following the examples of ZOG [Robertson 81] and Smalltalk [Tesler 81], we have chosen selection from menus as the principal mode of interaction between the user and the system. Typically, the user will select some descriptor of a description (in either the description pane or the example pane) and then use the mouse to display a “pop-up” menu on the screen which contains the operations appropriate to the selection. Selecting an operation may cause some action to take place, or it may cause another menu of options to appear on the screen. In a sense, the description in the example pane is itself one giant menu which is displayed anew when a retrieval is made and whose “menu items” are the descriptors comprising the description.

One advantage of menus is that no special syntax needs to be learned; the user does not have to remember the exact names and spelling of the operations or terms (descriptors). Instead, the user
just needs to be able to recognize the operations and terms displayed in a pop-up menu. So instead of having to learn how to type, the user must now learn how to point with a mouse, a skill which is readily learned. But more importantly, pop-up menus allow RABBIT to tell the user exactly what his options are at any given point. For example, there are many operations available in the user interface, but only a subset of them may be relevant at any one time. By using pop-up menus as a filtering template [Goldberg 79], the interface can prevent the user from trying to apply an inappropriate operation to his selected descriptor.

A potential disadvantage of menus is that menu selection may be inconvenient for an expert user who already knows the terms needed to specify his query. For a reasonable typist, the bandwidth provided by typing is several times greater than that provided by menu selection [Robertson 81]. Future implementations should allow the user to type in the names of descriptors from the keyboard, but adding this capability raises the question of how typing should be integrated with menu selection.

After selecting a descriptor (in either the description pane or example pane), the user can press a button on the mouse to get a menu of operations (see Fig. 6.1). The following subsections describe the functions of those operations in detail.

3.2 Require/Prohibit

require and prohibit are the most basic operations available in the interface. require simply adds the selected descriptor to the partial description and specifies that all retrieved instances must have that descriptor. Similarly, prohibit adds the selected descriptor to the partial description, but specifies that none of the retrieved instances can have that descriptor. Note that require and prohibit specify what descriptors the retrieved instance must and must not have. A natural extension is to allow operations which indicate preferences: should have and should not have. (This option is discussed further in chapter 9, section 1.3.) require and prohibit can be used from both the description and example panes.

If the descriptors in the image of the example instance are exactly what the user wants or exactly what he does not want, then a partial description can be built up by using just require and prohibit. However, it is much more likely that the image will contain descriptors which are "sort of what the user wants" but are not quite the "right" (or "wrong") descriptors. In that case, what the user really needs to be able to do is to find out what are some of the descriptors which are similar to the descriptors given in the image. This need to perform an extended retrieval is handled by the alternatives, specialize, and describe commands.
3.3 Alternatives

Alternatives gives the user access to descriptors which are similar to the given descriptor. In particular, invoking alternatives after selecting a descriptor presents the user with a menu of alternative values (if the selected descriptor was an attribute-value/attribute-value constraint pair) or alternative instance classes (if the selection was an instance class). Like require and prohibit, alternatives can be invoked from either the description or example panes.

3.3.1 Alternative values for attributes

The set of alternative values displayed by alternatives for a given attribute is the set of those instances which can fill that attribute. In deciding what values are "alternative values," RABBIT checks what the value restriction associated with the role corresponding to that attribute is; only instances which correspond to individual concepts which satisfy the value restriction are meaningful values. However, the value restriction is only an upper bound on what the possible alternative values are. A second, more restrictive, definition of "alternative values" for a given attribute is "the set of those instances which actually fill that attribute for some item in the database." In fact, the current implementation goes a step farther and defines the "alternative values" to be those values which actually fill that attribute for some item in the matching examples pane. The advantage of such a restrictive definition is that the user is guaranteed that there is some item in the database matching his previous partial description which has that value filling that attribute. As long as he reformulates his partial description without removing or prohibiting any previously required descriptors, the space of alternative values will be monotonically non-increasing, and he can zero in on his target item very quickly.

If the user does remove or prohibit previously required descriptors, then the list of instances in the matching examples pane may appear to bear little relationship to the current partial description. As a result, the alternatives command may produce alternatives which are counter-intuitive to the alternatives one would expect based on the current partial description. For example, if the descriptor 'PaloAltoRestaurant' were removed from the description of a restaurant in the description pane, the alternatives for 'location:' would still be just 'PaloAlto' until another retrieval is made. For that reason, RABBIT displays a pane which contains the previous partial description (i.e., the partial description which was the basis for retrieving those examples and hence, describes the matching examples) for reference.

A fourth option which we considered is the following: define the alternatives to be the (subclasses corresponding to the) subconcepts and immediate (instances corresponding to the) individuals of the value restriction. If the number of instances (immediate and indirect) is large (> 10), then a menu of that many items is difficult to read, and for very large numbers of instances, the menu may not even fit on the screen. [In the current implementation, if there are >10 menu items, RABBIT will
inform the user that there are too many items and not display the menu.] This fourth option reduces the number of menu items greatly compared to the other options presented above plus, it presents a complete decomposition of the value restriction. But this technique has the drawback that the actual value filling the selected attribute in the instantiation will not be displayed unless that value was an immediate instance of the value restriction; in other words, the value given for the attribute in the image may not be listed as an alternative value. Another disadvantage is that the alternatives will now include instance classes in addition to instances, which may be confusing. For those reasons, that option was not chosen. In short, RABBIT's alternatives command shows alternative values (another command, describe, provides the user with access to instance classes).

The third method above (only show fillers of attributes of instances in the example pane) also has its disadvantages. For large databases, it may be impractical to have to find all the matching instances and determine what values fill their attributes. Some possible solutions to this problem are to use some combination of the third and fourth methods or to use a hierarchy of menus.

### 3.3.2 Alternative instance classes

alternatives, when applied to an instance class, shows a list of instance classes which are similar to the selected instance class. One way of defining what constitutes a "similar instance class" is to say that all instance classes which are siblings of the selected instance class are similar to the selection. In the KL-ONE hierarchy, the siblings of a given concept would be all the subconcepts of all superconcepts of that concept. However, the number of such siblings can be quite large, and it is very likely that some siblings are more closely related to the given concept than others. KL-ONE allows such information to be expressed in the network via the ability for the network designer to give decompositions of the subconcepts of a concept. Accordingly, we define the "alternative concepts" to be those concepts which lie in the same decomposition(s) as the selected concept. For example, 'ChineseRestaurant' would be considered an alternative to 'FrenchRestaurant' while 'BostonRestaurant' would not (see Fig. 5.5). If the selected concept happens to belong to more than one partition, then the concepts in all those partitions are shown, grouped by partition.

### 3.3.3 Require any/Require all/Prohibit

Having selected one or more alternatives, the user now needs some way of incorporating those alternatives into his partial description. In the case of alternative values for an attribute, an attribute-value constraint pair with that attribute and the selected alternative values is added to the partial query description while in the case of alternative instance classes, the selected alternative instance classes are added. The obvious way to incorporate those alternatives into the partial description is to use the require and prohibit commands described earlier. If only one alternative
was selected, then require and prohibit perform the actions described earlier. But if more than one alternative was selected, the user could be intending either that the selected alternatives should all be descriptors of his target item or only that at least one of them should be a descriptor. So instead of offering him a choice of the menu items require and prohibit, RABBIT gives him a menu containing the options require any, require all, and prohibit if more than one alternative has been selected. require any specifies that any of the selected alternatives can be descriptors of his target and that one of them must be; require all specifies that they must all be descriptors of his target.

For example, in Fig. 6.4a, the user has selected the descriptor 'manufacturer: Atari' in the example pane and is asking for alternatives. RABBIT responds to that command with a menu containing the names of all firms which manufacture the products listed in the matching examples pane (see Fig. 6.4b). Suppose that the user wants the manufacturer to be either Xerox or Apple. He can now select those two items and then select outside the menu to get a menu containing the operations which he can perform on the selected alternatives (see Fig. 6.4c). Since he wants either Xerox or Apple, as opposed to both of them or neither of them, he selects require any, with the resulting display of the partial description updated as shown in Fig. 6.4d. Figs. 6.4e and 6.4f show the partial query descriptions resulting if the user had selected require all and prohibit instead, respectively.

As some readers may have noticed, RABBIT supports commands like require any and require all but only one kind of prohibit. The natural counterparts to require any and require all are prohibit any (meaning \(\neg(A \lor B)\)) and prohibit all (meaning \(\neg(A \land B)\), where A and B are descriptors). However, in English prohibit any and prohibit all are not usually distinguished; e.g. “I don’t like spinach or broccoli” and “I don’t like spinach and broccoli” both mean (to most people) “I don’t like spinach, and I don’t like broccoli.” Given that observation, we decided that it would be much less confusing to have a single command, prohibit, which took a list of attributes, A, B and C, and produced \((\neg A \land \neg B \land \neg C)\). (The user can still create \((\neg A \lor \neg B)\) indirectly, by using describe (see chapter 8) to create the boolean equivalent \((\neg(A \land B))\).

The purpose of the alternatives command was to provide the user with access to descriptors “similar” to the ones immediately available to him in the example and description panes. Given those alternative descriptors, the user then needs to be able to merge those descriptors into his query description according to some operation. The present implementation of RABBIT allows the user to AND or (inclusive) OR those descriptors into his query description. Thus, alternatives supports ORing between instance classes and between the values of attributes, but it should be noted that the ability to OR is very limited at present; alternatives allows ORing only over related descriptors and not between arbitrary descriptors. So, for example, there is presently no way to say: “Find a restaurant which is either in Palo Alto or accepts Visa.” The current implementation of alternatives is also limited in that only ANDing and inclusive ORing are
provided. So, for example, the user can not specify that the alternative values selected for an attribute in the partial description are to be a superset of the set of instances actually filling that attribute for some (matching) item in the database (in other words, a KL-ONE-style value restriction). (See chapter 9 for a discussion of generalized ORing and of extensions to the alternatives command.)
<table>
<thead>
<tr>
<th>Main Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-- Attributes of the query</td>
<td>-- Attributes of the query</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
</tr>
<tr>
<td>Product</td>
</tr>
<tr>
<td>Computer</td>
</tr>
<tr>
<td>OEM-Product</td>
</tr>
<tr>
<td>RetailProduct</td>
</tr>
<tr>
<td>name: The Atari-400</td>
</tr>
<tr>
<td>cost: $600, $850</td>
</tr>
<tr>
<td>manufacturer: Atari</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Information about Atari-400</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Previous Description</th>
<th>6 Matching Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>Product</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>Atari-400</td>
<td></td>
</tr>
<tr>
<td>Atari-800</td>
<td></td>
</tr>
<tr>
<td>Apple-II-plus</td>
<td></td>
</tr>
<tr>
<td>Cromemco-Z-</td>
<td></td>
</tr>
<tr>
<td>Star-8011</td>
<td></td>
</tr>
<tr>
<td>Star-8012</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.4b: The alternative manufacturers
Figure 6.4c: Selecting Xerox and Apple
Figure 6.4d: Updated query after using require any
Figure 6.4e: Updated query after using require all
### Main Description

---Attributes of the query---

**Entity**

**Product**

**Computer**

--- But Not ---

**manufacturer:**

Xerox, or Apple

---Information about Atari-400---

<table>
<thead>
<tr>
<th>Previous Description</th>
<th>6 Matching Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entity</strong></td>
<td>Atari-400</td>
</tr>
<tr>
<td><strong>Product</strong></td>
<td>Atari-800</td>
</tr>
<tr>
<td><strong>Computer</strong></td>
<td>Apple-II-plus</td>
</tr>
<tr>
<td></td>
<td>Cromemco-Z-</td>
</tr>
<tr>
<td></td>
<td>Star-8011</td>
</tr>
<tr>
<td></td>
<td>Star-8012</td>
</tr>
</tbody>
</table>

---

Figure 6.4f: Updated query after using prohibit
specialize is another operation, available in both the description and example panes, which gives
the user access to descriptors similar to the selected descriptor. Selecting an instance class in the
description or example panes and invoking specialize shows a menu containing the specializations
of the selected instance class. We have defined the specializations of an instance class to be all the
subconcepts of the (corresponding) concept in the KL-ONE hierarchy. (There is currently no
specialize command for attribute-value pairs. The describe command, described later in this
chapter, allows specialization of the value to create a value constraint, but specializing an attribute
is not supported. Such a command could be useful since KL-ONE roles can have modifiers and
differentiators, e.g., 'chairman:' could differentiate 'officer:'.)

After the user has selected one or more instance classes from the menu of specializations and
selected outside the menu, he is shown another menu. This menu lists the various actions which
the user is allowed to take at this time. If he had selected only one specialization, then his options
are to add that instance class to his partial description (via require and prohibit), find alternatives
to that instance class, or further specialize that instance class. If he had selected more than one
specialization, then his options are to require any, require all, or prohibit those instance classes.

For example, in Fig. 6.5a, the user has selected 'StoreFrontBusiness' in the description pane
and asked for its specializations. In Fig. 6.5b, the user has selected the specializations
'RetailBusiness' and 'ServiceBusiness' and gotten back a menu of operations. Selecting
require any adds those instance classes to his partial description (see Fig. 6.5c). Fig. 6.5d shows
the menu of operations for the case of the user's selecting just the single specialization
'RetailBusiness'.

3.5 Remove/Clear

remove and clear are commands for removing a descriptor (or part of a descriptor) from the
partial description and for clearing the partial description, respectively. Unlike the preceding
operations (require, prohibit, alternatives, and specialize), which are available in both the
description and example panes, remove and clear are only available in the description pane.
clear erases a query description, leaving behind 'Entity' as the sole descriptor. remove can be
used to remove an instance class, an attribute-value constraint pair, a single value of a value
constraint (removing the last value removes the entire attribute-value constraint pair), or an
embedded description. Note that if the user is removing an instance class which has subclasses
present in the query description, then those subclasses should also be removed unless they have
another superclass present in the query description. For example, if the query description includes
the descriptors 'Restaurant' and 'ChineseRestaurant', then attempting to remove
'Restaurant' should have the side effect of removing 'ChineseRestaurant' too; if the user
does not want a restaurant, then he certainly does not want a Chinese restaurant. [This feature has not yet been implemented.]

Figure 6.5a: Looking for specializations of StoreFrontBusiness
<table>
<thead>
<tr>
<th>Main Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td></td>
</tr>
<tr>
<td><strong>Entity Business</strong></td>
<td></td>
</tr>
<tr>
<td><strong>StoreFront Business</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Personal Services Business</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Retail Business</strong></td>
<td></td>
</tr>
<tr>
<td><strong>WholeSale Business</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Example**
- **name:** The Xerox Store
- **phoneNumber:** 225-7121
- **owner:** Xerox
- **personnel:** Mike Ward, Sally Shup, 
- **location:** Palo Alto
- **payment:** VISA
- **address:** Xerox Store, Palo Alto, CA

---

**Information about Xerox Store PA**

---

**Previous Description**

**10 Matching Examples**

<table>
<thead>
<tr>
<th>Entity Business</th>
<th>StoreFront Business</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComputerLand</td>
<td>RadioShack PA</td>
</tr>
<tr>
<td>DigitalDeli</td>
<td>RadioShack ST</td>
</tr>
<tr>
<td>MicroAgeMV</td>
<td>XeroxStore PA</td>
</tr>
<tr>
<td>RadioShackMV</td>
<td></td>
</tr>
<tr>
<td>RadioShackPA</td>
<td></td>
</tr>
<tr>
<td>RadioShack ST</td>
<td></td>
</tr>
</tbody>
</table>

---

**Figure 6.5b: Selecting two specializations**
<table>
<thead>
<tr>
<th>Previous Description</th>
<th>10 Matching Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity Business StoreFrontBusiness</td>
<td>ComputerLand DigitalDeli MicroAgeMV RadioShackMV RadioShackPA RadioShackST XEROXSTOREPA</td>
</tr>
</tbody>
</table>

Figure 6.5c: Updated query after using require any
<table>
<thead>
<tr>
<th>Previous Description</th>
<th>10 Matching Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entity</strong></td>
<td><strong>ComputerLand</strong></td>
</tr>
<tr>
<td><strong>Business</strong></td>
<td><strong>DigitalDeli</strong></td>
</tr>
<tr>
<td><strong>StoreFrontBusiness</strong></td>
<td><strong>MicroAgeMV</strong></td>
</tr>
<tr>
<td></td>
<td><strong>RadioShackMV</strong></td>
</tr>
<tr>
<td></td>
<td><strong>RadioShackPA</strong></td>
</tr>
<tr>
<td></td>
<td><strong>RadioShackST</strong></td>
</tr>
<tr>
<td></td>
<td><strong>XeroxStorePA</strong></td>
</tr>
</tbody>
</table>

Figure 6.5d: The possible operations for one specialization
3.6 Other operations: describe, alternative views, retrieve

In addition to the above operations, there are two more operations which affect the partial description: describe and alternative views. describe allows the user to inspect the description of a particular value of an attribute in the image of the example instance or to (recursively) describe what values he wants to fill a particular attribute in his partial description. Using describe, the user can create a description and insert that description as the value constraint of an attribute-value constraint pair. describe can be invoked from both the description and example panes. (See chapter 8 for a complete explanation of the describe facility.)

Unlike describe, alternative views is available only in the example pane. alternative views is a means for discovering other ways of looking at a particular instance. Consider the following example: a property tax assessor is viewing information about a particular restaurant from the perspective of a taxable piece of property and feeling hungry, decides that he would also like to see that restaurant viewed as a place which serves food. An earlier implementation required the user to construct a new partial description top-down from 'Entity', but with the alternative views command, the user can change the perspective for viewing an instance directly. [We are indebted to Laura Gould [1981] for first pointing out to us the need for this notion of giving the user direct access to alternative perspectives for any given instance.] (alternative views is described more fully in chapter 7.)

An operation which does not affect the partial description, but which we have mentioned frequently, is the retrieve command. retrieve is the option on a menu obtained by moving the mouse into the "gray" space between panes and pressing a button on the mouse (see Fig. 6.6). [Future implementations should probably replace the retrieve menu with a permanent button on the display.] retrieve performs the action of retrieving a new example instance and updating the panes of the display.

In summary, RABBIT attempts to provide a set of operations for constructing and modifying descriptions which mirrors the kinds of operations which we believe people use in reformulating descriptions in their own heads. require and prohibit are the basic operations, corresponding to "yes, I want this" and "no, I don't want that." alternatives, specialize, and describe allow the user to perform extended retrievals to find similar or related descriptors. alternative views allows the user to change contexts, and remove and clear allow the user to backtrack or start over, respectively.

From the point of view of KL-ONE, the various operations permit the user to move through a KL-ONE network. specialize allows movement downward; upward movement is provided indirectly by the remove command (removing a concept, in effect, generalizes the description);
lateral movement over a short distance is provided by alternatives; and lateral movement over large chunks of the network is made by moving up and then down a different path.

Figure 6.6: The retrieve command
4. Semantic Consistency

One of the goals in the design of RABBIT has been the automatic maintenance of the semantic consistency of query descriptions (for similar work in semantics checking, see [Fikes 81a] and [Senko 77]). Two approaches one can take to attain this goal are: (1) force the user to create only queries which are semantically consistent, or (2) permit the user to create a potentially anomalous description and then automatically remove any inconsistencies. In practice, RABBIT uses both approaches to varying degrees. Examples of the first approach which have already been mentioned include allowing the user to invoke only “appropriate” operations via pop-up menus, and forcing the user to choose alternative values for a role which satisfy the value restriction of that role.

Another example of the first approach occurs when the user recursively invokes specialize. For example, the user could select 'Entity', specialize 'Entity' to get (among other things) 'Product', specialize that to get (among other things) 'Computer', and then require 'Computer'. RABBIT will add not only 'Computer' as a required descriptor of the partial description but also add 'Product' and 'Entity'. The reason for doing this is that the user should have an explicit path (i.e., a chain of instance classes connected by specializations links) visible to him from the highest (most general) instance class(es) in his description to the lowest (most specific) instance class(es). With an explicit path, he is less likely to become lost in the network (a problem which afflicts users of ZOG [Robertson 81]). In addition, backtracking is easily performed by removing the lower concepts and starting off on a new path. But there is an even stronger argument for wanting a path if the last specialization, in this case, 'Computer', is prohibited. There are likely to be many things in a database which are not computers, so the specification “an entity which is not a computer” is not an especially useful description. However, it seems very reasonable to infer that the user was implicitly requiring the instance classes along the path which he took to reach 'Computer'; i.e., the user wants “a product other than a computer.” We could have chosen the simpler technique of simply requiring the superconcept of 'Computer', except that 'Computer' might be a subconcept of several other concepts, such as 'ElectricalDevice', and we would not know which path the user wants. It is important to know which path since the instance classes comprising that path are added to the query description and as we will see in the next chapter, the perspective for viewing example instances is determined by the query description.

An example of the second approach to maintaining the semantic consistency—removing inconsistencies already present—is the following: if the user prohibits a previously required descriptor, then that descriptor is made a prohibited descriptor and removed from the list of required descriptors. The assumption is being made that the most recent command supersedes all preceding commands. (An earlier implementation supported an undo operation which allowed the user to reverse the effects of the last operation invoked, so that accidental invocation of
operations could be easily undone. That command has not yet been re-implemented.) Similarly, requiring a previously prohibited descriptor adds that descriptor to the list of required descriptors and removes it from the list of prohibited descriptors. The preceding consistency check also applies when the user prohibits (requires) a previously required (prohibited) value in the value constraint of some attribute-value constraint pair.

If the descriptor happens to be an instance class, then requiring or prohibiting that descriptor may have additional side effects since an instance class may imply certain attribute-value pairs (via the roles and values attached to the corresponding concept) which may now contradict the query description. For example, the instance class 'PaloAltoRestaurant' might have an attribute-value pair 'location: PaloAlto' associated with it. If both 'PaloAltoRestaurant' and 'location: PaloAlto' had been previously required, then prohibiting 'PaloAltoRestaurant' will remove both 'PaloAltoRestaurant' and 'location: PaloAlto' from the required part of the query description. By extension, if an instance class is removed, then any of its attributes which are also present in the partial description as part of some attribute-value constraint pair will also be removed. For example, if both 'Computer' and 'disk: T300-disk' had been required descriptors in the query description, then removing 'Computer' will also cause 'disk: T300-disk' to be removed since that attribute-value constraint pair is meaningless outside the context of 'Computer'. (The name of an attribute may name other attributes in other contexts, but the attribute itself is only meaningful in the context of the instance class owning it.)

Chapters 7 and 8 contain additional cases of attempts by RABBIT to maintain the semantic consistency of query descriptions.

Although some of the information used in checking the semantics of a description is syntactic in nature, most of the information is derived from the KL-ONE network itself. KL-ONE allows for a very rich representation of knowledge, with the advantage that some of that (meta-)knowledge of representation can be used in the user interface.
Chapter 7 - Perspectives

As mentioned earlier, in chapter 4, a perspective is a way of describing an instance in the database from a particular viewpoint. The inclusion of perspectives in RABBIT was originally motivated by the implementation of RABBIT in KL-ONE; however, we now believe that perspectives are a fundamental part of any interface to a database modelling the "real world." Unlike the perspectives in other knowledge representation languages (e.g., KRL [Bobrow 77] and PIE [Goldstein 80]) or the "views" of relational databases [Date 77], the perspectives of RABBIT are dynamic—they change as the partial description is being reformulated and thus, reflect the user's evolving conception of what he is looking for.

1. The Uses of Perspectives

1.1 Controlling the presentation of instances

One of the key ideas behind RABBIT is that the user is presented with a description of an example instance. But in implementing a working prototype, we had to be much more precise as to what "a description of an instance" is. In an earlier implementation, we decided that the description of an instance was the set of all its attributes, both local and inherited, and all its superclasses. However, that scheme produced instance descriptions which were very bulky and difficult to read, despite the fact that our original test database had only a few tens of instance classes, with a couple of attributes per instance class. In a database in which instances have hundreds of attributes, there would be so much information being presented in the description of an example instance that the user would have a very difficult time deciding which descriptors were relevant to his partial description. In such a situation, the user would probably be better off trying to guess what the appropriate descriptors were.

The problem seemed to be a lack of a definite perspective for viewing items in the database. When people think of items in their heads, they are actually "seeing" that item with respect to some (possibly default) context. For example, a geologist's description of the moon would differ substantially from the descriptions provided by an astronomer or a poet. Consequently, we have added the notion of perspective to our interface; whenever a person looks at instances in a database, he is actually looking at that instance with respect to some perspective. In other words, the user can not examine an item in a database without first specifying a perspective for viewing that item. Thus, perspectives behave like filters between the user and the database which screen out "extraneous" information in the database. They provide coherent subviews of the total view of an
instance.

We have implemented perspectives using the concepts of the KL-ONE hierarchy. Each (generic) concept is said to define a perspective for viewing individuals of that concept. The perspective specified by a concept consists of that concept and its local roles. Since KL-ONE allows multiple inheritance, a perspective is, more generally, a collection of one or more concepts and the local roles of those concepts. The description of an individual concept from the perspective of some collection of concepts consists of those concepts, the roles which that individual concept inherits from those concepts, and the values filling those roles for that particular individual concept. Similarly, in the RABBIT description domain the description of an instance from the perspective of some collection of instance classes consists of those instance classes, the attributes which that instance inherits from those instance classes, and the values filling those attributes for that particular instance. We refer to that description of an instance as the image of that instance from that perspective. Note that a given instance may have several different images, i.e., it may be viewed from many different perspectives. Similarly, a given perspective may be applied to many different instances to produce various images which all share the same attributes but have different values filling those attributes.

Having realized the need for perspectives, we needed to decide what perspective should be used in displaying the example instance to the user. How an instance is presented in the example pane should be determined by how the user wants to see that instance, and since the partial description represents what the user wants to find in the database, it seemed reasonable that the partial description should also be used to determine the perspective for viewing the retrieved example instances. In particular, the local attributes of the required instance classes in the user's partial description are very likely to be the attributes in which he is interested. Therefore, RABBIT describes the example instance in terms of the required instance classes and their local attributes.

Since the perspective for viewing instances is a function of the partial description, the perspective will change as the partial description changes. In other words, perspectives are dynamic; they change to reflect the user's evolving conceptual description of the target instance as expressed in his (partial) query description. Because of the dynamic nature of perspectives, the user can construct his own perspectives out of the primitive perspectives (i.e., the generic concepts) originally supplied by the network designer.

1.2 Facilitating the understanding of instances

Besides controlling the presentation of instances, perspectives also provide a framework for understanding instances. Since the attributes in an image of an instance are defined by some perspective, the general meanings of those attributes should be related. For example, the attributes of a business viewed as an investment might include 'assets:', 'debts:',


'revenue:', and 'profit:', whereas the attributes of a business viewed as a manufacturer might include 'products:', 'employees:', and 'location:' . Each set of attributes provides a fairly coherent and cohesive view of a business because the set arises from the definition of a concept/instance class which presumably has semantic integrity. The name of an attribute standing alone might be ambiguous, but since an attribute commonly appears with other, related, attributes within some context (the perspective), the meaning of that attribute is much more likely to be understood. (In addition, the value filling that attribute in an image might also provide a clue to the meaning of that attribute.)

1.3 Constraining the search space

In chapter 6, we talked about some of the automatic checks in RABBIT for maintaining the "meaningfulness" of query descriptions. One of the strongest semantic checks in RABBIT is the use of perspectives as a means for constraining the search space (i.e., the space of the database accessible to the user as he makes his query). The constraints imposed through perspectives prevent the user from even beginning to construct certain kinds of semantically anomalous queries.

For example, since instances are always being viewed from some perspective inferred from the partial description, the user does not have access to a particular attribute unless he has first made the instance class owning that attribute a required descriptor of his partial description. Thus, only attributes which are "appropriate" or "relevant" to his partial description are available to the user for inclusion in his query description. For example, if 'Book' and 'Science-Fiction' are the required instance classes in the user's partial description, then there would be no way for the user to add the attributes 'employees:' or 'CPU:' to his partial description since these are not attributes of either 'Book' or 'Science-Fiction'.

The fact that the attributes in the partial description must be owned by some instance class in the partial description also has the benefit that it simplifies the process of finding instances in the database which match the partial description. If an instance which is an instance of all the required instance classes is found, then we immediately know that the instance has all the attributes given in the partial description; only a matching of the (actual) fillers of those attributes against the (allowable) fillers specified by the value constraints of the attribute-value constraint pairs needs to be done.

Limiting what attributes are accessible to the user is just one of the ways in which perspectives constrain the directions in which the partial description can be elaborated. The perspective defined by the partial description also specifies the next layer of instance classes, which are the specializations of the currently required instance classes. The image of the example instance consists of the required instance classes, their local attributes and the values filling those attributes, and the specializations of the required instance classes. (Actually, the specializations given in the image are
not all the specializations of the required instance classes, but rather, all specializations which have the retrieved example as an instance. This layer of instance classes represents the possible directions in which the partial description can be further refined. The user already had access to those instance classes via the specialize command, but by explicitly showing the specializations, we immediately remind the user of what paths he can take in the network. So instead of having to perform exploratory probes and extended retrievals with specialize, the user can proceed directly with the reformulation of his query description.

Although the extra level of instance classes allows for easier and quicker construction of partial descriptions, it also can be confusing in that the user may forget that not all the instance classes in the image have been added to the partial description. However, we believe that the benefits outweigh the disadvantages. Careful observations of users of RABBIT will need to be made to determine the validity of this decision.

1.4 Managing the non-uniformity of the database

As databases attempt to represent knowledge about the real world, their structures become increasingly complex and begin to take on a non-uniform shape (see chapter 2); i.e., the shape of the database changes depending upon where one is within the database. Perspectives can be used to manage this non-uniformity since they can be used to support the notion of a changing space of information (i.e., the amount of information in the database accessible to the user changes). We have already seen an example of a reduction in the amount of available information—over the course of several reformulations of the query description, the number of alternatives given by the alternatives command decreases. Perspectives, on the other hand, are a scheme for increasing the amount of domain-specific information available to the user.

As the user refines his partial description, he will often add new instance classes to his query as required descriptors of his target instance. Since each instance class defines a perspective, the net effect of adding new instance classes is the addition of new perspectives, and hence, the appearance of new attributes in the description of the example instance. Consequently, the user does not need to concern himself with the shape of the database as expressed in the attributes of instance classes; all he must do is decide on the conceptual shape (e.g., is the user looking for a business, a product, or something else), and the appropriate attributes are made available to him automatically. Furthermore, the attributes are made available only when he needs access to them, as indicated by his choosing the instance class owning those attributes.

For example, if the user has only specified that he is searching for a product, then only the attributes common to all products are visible to him. But if he now further specifies that he wants a computer product, then those attributes common to computer products are now accessible to the user in addition to the attributes common to all products (see Fig. 7.1).
Figure 7-1a: The image of the Star-8011 viewed as a Product
<table>
<thead>
<tr>
<th>Main Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes of the query</td>
<td>Entity</td>
</tr>
<tr>
<td>Product</td>
<td></td>
</tr>
<tr>
<td>Computer</td>
<td>Attributes of the query</td>
</tr>
</tbody>
</table>

**Example**

```plaintext
RetailProduct
name: TheStar-8011
cost: $13,850, $15,055
manufacturer: Xerox
disk: Xerox-10
display: Large-Format-Display
memory: Xerox-192-memory
CPU: 8000
operatingSystem: Star-system

Information about Star-8011
```

<table>
<thead>
<tr>
<th>Previous Description</th>
<th>6 Matching Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity Product Computer</td>
<td>Apple-II-plus Atari-400 Atari-800 Cromemco-Z-80 Star-8011 Star-8012</td>
</tr>
</tbody>
</table>

Figure 7-1b: The image of the Star-8011 viewed as a Computer
An advantage of this layering of information is that the user does not have to know what the various templates are (as expressed via the attributes of the image). He can browse though the database, constructing his own "personalized" templates through the dynamic addition of perspectives to his partial description. In short, perspectives provide a natural partitioning of the information in the database into uniform regions and allow the user to move freely from one region to another via the hierarchy of instance classes.

2. Changing Perspectives

Perspectives are normally constructed implicitly during the course of building up a partial description. But occasionally, the user might want direct control over the specification of the perspective. For example, a user viewing a particular instance with respect to some perspective might wish to find out what are the other ways of viewing that instance. That ability is provided by the alternative views command, which presents the user with a menu containing alternative perspectives [actually, the instance classes which denote the alternative perspectives] and allows the user to choose one or more of them. The instance classes corresponding to the selected perspectives are added to the partial description, and the instance currently being displayed in the example pane is redisplayed with respect to the perspective specified by the new partial description. In addition, all other instances in the database which match the new partial description are retrieved as well and listed in the matching examples pane. The assumption is being made that the user wants to examine the current example instance from a new perspective. If, in fact, the current example no longer matches the new partial description, then the redisplay of the current instance and retrieval of new instances are aborted. However, the new perspective instance classes are still added to the partial description, so that the user can now explicitly retrieve the instances matching the new partial description using the retrieve menu command.

In deciding what instance classes comprise the "alternative views" of an instance, RABBIT imposes the constraint that the alternative views displayed are, in fact, alternative views of the current instance in the example pane. [Note that these views need not be valid perspectives for viewing the other instances in the matching examples pane.] Given that constraint, there are still several options available in deciding what the alternative views should be. They include:

1) showing all the instance classes in the lattice of instance classes between ‘Entity’ and the example instance
2) displaying those specializations of each required instance class which lie within the lattice above the example instance
3) showing those instance classes which have the example instance as an immediate instance (i.e., there is a direct link in the KL-ONE hierarchy between the instance and those instance classes)
4) choosing one of the above options but excluding those instance classes which are already required descriptors of the partial description.

(Fig. 7.2 shows a fragment of a KL-ONE network and indicates which instance classes would be included in the alternative views as defined in options (1) through (3). The ovals labelled "Required" are the required instance classes.)

---

**Figure 7.2:** Possible implementations of alternative views
The first option has the advantage that all possible views of the given instance are being shown to the user. However, the total number of instance classes in the lattice above the example instance could be quite large, and a menu containing that many items would be difficult to display or read. Furthermore, all the instance classes in the lattice are interrelated; hence, selecting an instance class from that menu may imply that one or more other instance classes in the menu should also be considered selected. We would then need to decide which, if any, of those implied instance classes should be added to the partial description. The third option is useful in that it allows a bottom-up construction of the partial description, but there are likely to be many attributes of the selected instance class which were inherited from superclasses and hence, these attributes, although relevant to the instance class, would not be included in the perspective defined by that instance class unless the corresponding superclass is also added to the partial description. This problem of not knowing which instance classes should be implicitly added to the partial description does not occur in the case of the second option since the instance classes comprising the alternative views are directly connected to the current partial description, i.e., each of those instance classes has a superclass already present in the partial description. In addition, the number of alternative views is not likely to grow very large. For those reasons, that option was implemented in the current version of RABBIT. The modifications introduced by the fourth option, which removes redundancy from the list of alternative views, were also adopted so that the alternative views are, in fact, new views of the given instance.

The decision to implement the second and fourth options is only a tentative one, for there are several potential drawbacks with these options. For example, the user can not jump immediately to a radically different point of view; he can only move there gradually through successive invocations of alternative views. Experiments with real users will need to be performed to determine which of those options is best or if none is, what are the problems which need to be resolved.

Fig. 7.3 shows an example of using alternative views. In Fig. 7.3a the user has pressed a button on the mouse in order to display the alternative views command. When the user selects that command, RABBIT displays the alternative views for the example instance in a menu. After the user has selected one or more views (see Fig. 7.3b), RABBIT redisplays the example instance from a new perspective which includes those added view(s) and retrieves all matching instances (see Fig. 7.3c).

3. Extended Descriptions

In this chapter, we have mentioned two ways in which a perspective can be constructed: by adding instance classes to the query description using the require command and by using the alternative views command. A third way is to incorporate partial descriptions of related instances
into the image of the example instance to create an extended description of the example instance. But before we can discuss this form of constructing a perspective, we must first introduce the idea of embedded descriptions.

Figure 7.3a: The alternative views command
Figure 7.3b: Choosing the new view InvestmentBusiness
### Example

- Personnel: Mike Ward, Sally Shup, San
- Location: Palo Alto
- Payment: VISA
- Address: Xerox Store, Palo Alto, CA
- Annual Revenue: $1,000,000
- Equity: $250,000
- Return on Investment: 11%

---

### Information about Xerox Store PA

---

### Previous Description

<table>
<thead>
<tr>
<th>Entry</th>
<th>Business</th>
<th>Store Front Business</th>
<th>Investment Business</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>Xerox Store PA</td>
<td>Advanced Computer</td>
<td>Byre Shop</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Computer Land</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Digital Deli</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MicroAge MV</td>
</tr>
</tbody>
</table>

---

Figure 7.3c: The updated display
Chapter 8 - Embedded Descriptions

Embedded descriptions are simply descriptions which are embedded within another description. The motivation for adding the capability to construct embedded descriptions is a theory of human memory, which proposes that the retrieval process is not only iterative, but also recursive. The ability to construct embedded descriptions allows the user to describe what a descriptor of the partial description should be and to inspect the descriptions of those attribute values whose names he does not know or understand.

1. The Uses of Embedded Descriptions

1.1 Recursive construction of descriptions

Williams and Hollan [1981] have noted that "one important aspect of the retrieval cycle is that each major stage is recursive." [p. 111] (Norman and Bobrow [1979] make a similar observation p. 115.) Accordingly, we have added a facility to RABBIT which allows the recursive construction of descriptions. As the user constructs his partial description, he may decide that the descriptors of the image of the example instance or of the partial description are not quite the descriptors that he desires. Earlier, we mentioned that he can use the alternatives and specialize operations to find similar or related descriptors. But if the alternatives and specializations are unfamiliar terms or are too numerous to be shown using alternatives and specialize, the user really needs to be able to describe the descriptor that he wants. Even if the alternatives/specializations are few in number and known to him, the user may still wish to describe those alternatives/specializations since a description is often more general and more meaningful than an enumerated list of alternatives/specializations. For example, if the user wanted to find "a store which sells items manufactured by Apple," then the sub-description "items manufactured by Apple" is much more desirable than an actual list of the items manufactured by Apple. Furthermore, new items manufactured by Apple which are subsequently added to the database will be automatically subsumed under the sub-description "items manufactured by Apple," whereas the enumerated list would have to be re-enumerated.

The operation in RABBIT which allows the user to create an embedded description is the describe command. Applying describe to an instance class presents a new window for constructing descriptions on the workstation display and allows the user to create a description within that window which can later be merged back into the (main) partial description in the original window. Similarly, applying describe to an attribute-value pair displays a new window
and allows the user to build up a description which can then be used as the value constraint of the attribute.

The user may create embedded descriptions within his main partial description at any time. Those embedded descriptions are descriptions in their own right, so the user can recursively embed descriptions within embedded descriptions. However, the user can not embed an arbitrary description within another description. For example, if the user selects the instance class 'City' in order to describe it, then he is not allowed to describe something which is not a city. Similarly, if he is describing the fillers of the attribute 'location:', which has a value restriction of 'City', then he is not allowed to describe values which are not cities. RABBIT enforces this constraint by initializing the partial description in the new window to contain a required instance class more specific than 'Entity'. In the case of describing an instance class, the initial required instance class is that instance class; in the case of an attribute-value pair, the initial required instance class is the value restriction of the attribute. Those initial required instance classes are called distinguished roots—instance classes which can not be removed or prohibited by the user. By using distinguished roots in the partial description, RABBIT can prevent semantically meaningless queries like “Find a computer store in a city which is a person with social security number 264-45-1234” from being constructed. [The distinguished root of the main (original) partial description is 'Entity'.]

Normally, the perspective defined by the partial description includes only the required instance classes and their local attributes. However, in the case of embedded descriptions, there is typically only a single required instance class (the root instance class) in the partial description. Consequently, the only attributes shown in the image of the example instance are the attributes local to that instance class. Moreover, the user does not have access to the inherited attributes of that instance class. To remedy this lack of attributes, RABBIT implicitly assumes that all superclasses of the root instance class have been previously required, so that all their attributes are also present in the perspective. But since the user is not allowed to prohibit or remove those superclasses, they are not shown in the partial description. This choice is still not very satisfactory since we are now potentially showing the user too much information, most of which is probably irrelevant or even confusing to the user's intended description. (Chapter 10 discusses some possible remedies for this problem of 'conceptual jar'.)

After implementing embedded descriptions and distinguished roots, we encountered a slight problem: there were perspectives, and thus, attributes, applicable to the retrieved instances which were still inaccessible to the user via specialize or alternatives. For example (see Fig. 8.1), an 'Owner' is a 'Person-or-Corporation' but not all 'Person's or 'Corporation's are 'Owner's. If the root instance class of the partial description is 'Owner', then the image of the instance 'John' will include all attributes inherited from 'Owner', 'Person-or-Corporation', and 'Entity', but no attributes from 'Person'. Since 'Owner' is the only explicit instance class of the partial description, the user can not reach 'Person' using specialize or alternatives, but...
the alternative views command does give him access to ‘Person’ and hence, to the local attributes of ‘Person’. Thus, alternative views can serve an important function in the construction of embedded descriptions.

Figure 8.1: An inaccessible instance class (Person)
1.2 Inspection of unknown values of attributes

Besides allowing the user to describe instance classes or the values of attributes, the describe command can also be used to inspect the descriptions of the values of attributes. For example, suppose that the user sees the attribute-value pair ‘disk: Xerox-10’ in the image and that the name Xerox-10 is meaningless to him. Asking for alternatives to Xerox-10 might also yield a collection of names, none of which are meaningful to the user. But if the user selects that attribute-value pair and invokes describe, the example instance in the new window will initially be the value of the selected attribute-value pair, namely, Xerox-10. Thus, the user will be shown a description of the Xerox-10. Now that he knows how a “typical” disk is described (and in particular, the Xerox-10), he can proceed to describe the exact disk which he desires. Or if the user is now unfamiliar with some value of an attribute of the Xerox-10, he can continue to use describe on the attribute in question.

If the attribute-value pair had more than one value, then the first value is the initial example instance. But the user can still examine the other values simply by selecting their names in the matching examples pane. If the attribute-value pair had no values, then the first instance found which satisfies the value restriction is the starting example.

2. Example I: Describing a Value Constraint

Suppose the partial description is in the state shown in Fig. 8.2a and that the user is searching for a computer with either a 256K byte or a 512K byte 5-1/4" floppy disk. Furthermore, suppose that the user is completely unfamiliar with the names of commercially available disks. Normally, the user would select ‘disk: Xerox-10’ in the example pane and ask for alternatives, but in this case, the list of alternative disks would be completely meaningless to him. So instead, he selects ‘disk: Xerox-10’ and asks to describe it (see Fig. 8.2b). RABBIT responds by displaying a new window on the screen with the partial description initialized to the value restriction of ‘disk:’, namely, the instance class ‘Disk’, and the example instance initialized to ‘Xerox-10’ (see Fig. 8.2c).

The user can now proceed to examine the descriptions of the disks listed in the matching examples pane or, in this case, create a description of the particular disk that he wants; all the basic operations for constructing and modifying a query description in the original window are also available in this window. For example, to get the desired disk size, the user selects ‘capacity: 10-megabytes’, asks for alternatives, selects both ‘256K-bytes’ and ‘512K-bytes’, and requires that any of them be true. Similarly, using the alternatives command on ‘type: Winchester’ produces ‘5-1/4-floppy’ as one of the choices, which can then be required. (The resulting partial description is shown in Fig. 8.2d.)
Figure 8.2a: A query description
Figure 8.2b: Describing the value of an attribute
### SubDescription for disk:

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes of the query:</td>
</tr>
<tr>
<td>Disk</td>
</tr>
<tr>
<td>Attributes of the query:</td>
</tr>
</tbody>
</table>

### Example

--- Attributes of Xerox-10 ---

- **Disk**
- **Cost:** $2500
- **Manufacturer:** Shugart
- **Name:** TheXerox-10
- **Capacity:** 10-megabytes
- **Type:** Winchester
- **Average Access Time:** 50msec

--- Attributes of Xerox-10 ---

--- Information about Xerox-10 ---

### Previous Description

#### Disk

---

### 6 Matching Examples

---

- Xerox-10
- Atari-810
- Datatrak-8
- Shugart-5
- T300-disk
- Xerox-29

---

Figure 8.2c: The embedded description window for describing a disk
## SubDescription for disk:

### Attributes of the query:

**Disk**
- Capacity: 256K-bytes, or 512K-bytes
- Type: 5-1/4-floppy

---

## Description

### Attributes of Xerox-10

**Disk**
- Cost: $2500
- Manufacturer: Shugart
- Name: The Xerox-10
- Capacity: 10-megabytes
- Type: Winchester
- Average Access Time: 50msec

---

## Example

---

**Information about Xerox-10**

---

## Previous Description

<table>
<thead>
<tr>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerox-10</td>
</tr>
<tr>
<td>Atari-810</td>
</tr>
<tr>
<td>Datatrak-8</td>
</tr>
<tr>
<td>Shugart-5</td>
</tr>
<tr>
<td>T300-disk</td>
</tr>
<tr>
<td>Xerox-29</td>
</tr>
</tbody>
</table>

---

## 6 Matching Examples

---

### Figure 8.2d: The completed embedded description
Having constructed the (embedded) description he desires, he can do one of three things (see Fig. 8.2d): retrieve all instances matching his description (retrieve), merge his description back into the original partial description (require any, require all, prohibit), or exit the window (and its partial description) without affecting his original partial description (abort). [Name is unimplemented at this time; its intended function is explained in chapter 9.] The options for embedding his description as the value constraint of the original attribute, 'disk:' are the same as the ones for assigning multiple alternative values to an attribute; selecting require any specifies that one of the values filling the attribute 'disk:' in the target instance must match the embedded description while selecting prohibit specifies that none of the values may match the embedded description. [Require all has not been implemented at this time. As it turns out, require all is ambiguous: it could specify that all instances which match the embedded description must be values of the attribute (i.e., the potential values (specified by the value constraint) must be a subset of the actual values), or that all values of the attribute must satisfy the embedded description (i.e., the actual values must be a subset of the potential values). Since both interpretations are reasonable, require all should probably be broken up into two operations, say, require all of and require all must be.]

At the beginning of this example, we said that the user was looking for a computer with a 256K or 512K byte 5-1/4" floppy disk (i.e., the user is looking for a computer with any disk which is a 256K or 512K byte floppy). Thus, the user selects require any, which adds the description in the new window to the original partial description and erases the new window from the workstation display. Now he can do a retrieval to get the new matching examples, shown in Fig. 8.2e. If the user later wishes to modify the embedded description, he can select the embedded description in the partial description and invoke describe, which will open a new window with the partial description initialized to be that embedded description. [In the current implementation, the user can not select an individual attribute of an embedded description. Chapter 9 discusses an extension which would allow such access.]

Since the reformulation of descriptions is recursive, the user could have created an embedded description within the embedded description. For example, if he had further wanted the disk to be manufactured by a firm incorporated in California or Delaware, he could have used describe on the 'manufacturer: Shugart' attribute-value pair in Fig. 8.2c. Fig. 8.3 shows the resulting partial description and retrieved example instance.

3. Example II: Describing an Instance Class

The preceding example was an example of using describe on an attribute-value pair. Executing describe for an instance class differs in a few minor ways. One difference is that the retrieve/return menu of the new window has require in place of require any and require all. Another difference is that if the originally selected instance class was a required (prohibited)
Figure 8.2e: The completed query description
Figure 8.3: An embedded description within an embedded description
descriptor in the partial description, then that instance class is replaced by the embedded description if require (prohibit) was selected on returning from the new window back to the original window. As before, selecting the embedded description in the partial description initializes the new window's partial description to be that embedded description. An example query using an embedded description of an instance class is the following: "Find a restaurant in Palo Alto which is either a Chinese restaurant, or a French restaurant which accepts Visa" (see Fig. 8.4).

- Attributes of the query -
  Restaurant
  ChineseRestaurant, or
  |FrenchRestaurant
  payment: Visa
  location: PaloAlto

- Attributes of the query -

Fig. 8.4: An embedded description of an instance class

4. Extended Descriptions

In Figs. 8.2e and 8.3, the reader may have noticed that the attribute-value pair 'disk: Shugart-5' in the example pane was followed by more attribute-value pairs, shown indented with a vertical bar to the left. This collection of attribute-value pairs is an extended description of the filler of 'disk:', 'Shugart-5', and serves to describe further what a Shugart-5 is. The reason for including the extended description in the image is that the user chose to describe the fillers of 'disk:' (using an embedded description) as opposed to enumerating the fillers. If, for example, the user chose to describe the fillers because he did not understand the names of the fillers, then simply showing him the names of the fillers in the image tells him nothing about the meanings of those fillers or the reasons why those particular fillers were chosen in the image. Thus, it seems reasonable to include in the image the descriptors of the embedded description which specified the filler(s). In Fig. 8.3, the user may not know what a Shugart-5 is, but the important thing as far as he is concerned is that the Shugart-5 has a capacity of 256K bytes and that it is manufactured by a firm incorporated in California.

In deciding how much of the embedded description should be included in the extended description, we had several options. We could:

1) show all the descriptors, both implicit and explicit, of the embedded description [In the example above, the extended description of 'Shugart-5' would include all the descriptors of a Disk.]
2) show all the the descriptors explicitly stated in the embedded description [In the example, 'Shugart-5' would be described with the descriptors 'Disk', 'capacity: 256K-bytes', and 'type: 5-1/4-floppy'.]

3) show all the explicit attribute-value pairs [In that case, 'Shugart-5' would be described as in option 2), but without the instance class 'Disk'.]

4) show those explicit attribute-value pairs which have ambiguous values, i.e., the embedded description does not specify a unique value for the attribute [This is the case shown in the example for 'Shugart-5' in Fig. 8.3.]

We chose option 4) for several reasons. First, option 1) has the drawback that the number of descriptors may be quite large. In fact, there could be more descriptors in the extended description than in the rest of the top-level partial description with the result that the user's attention would be diverted from the top-level partial description to the extended description. Options 2) and 3) show descriptors whose presence can be implicitly determined. For example, since the embedded description specifies that the disk must be a 5-1/4" floppy, we know that the Shugart-5 is a 5-1/4" floppy disk. This redundancy need not be bad, since it could serve to emphasize the relationship between the actual fillers and the corresponding embedded description; however, for now we are removing that redundancy until we have a chance to perform experiments involving people actually using the interface.

Since the extended description makes new attributes visible to the user, the extended description is actually a part of the perspective from which the user is viewing instances. Thus, the display of extended descriptions as a result of constructing embedded descriptions is a second mechanism for constructing perspectives (the first one was described in the preceding chapter). The attributes in the extended description are not immediate attributes of the matching example instances, but changing the values filling those attributes does affect what instances will match the partial description as a whole. However, the attributes in the extended description differ from the attributes in the rest of the partial description in that all the attributes in the extended description are already present in the partial description. Since one of the uses of a perspective is to provide new information (through new attributes), it would seem that option 1) above is the better choice if we wish to treat the extended description as part of the perspective. Moran [1981] has suggested that the user should be allowed to suppress the display of descriptors in the instantiation. If such a facility is implemented, then the problem of overly detailed descriptions can be avoided.
Chapter 9 - Future Extensions

The first two sections of this chapter describe possible additions to the RABBIT interface. These extensions can be divided into two major areas: improvements in the expressibility of RABBIT and enhancement of RABBIT's flexibility and functionality. The last section of this chapter discusses the application of the retrieval by reformulation paradigm to traditional data models.

1. Increasing the Domain of Expressible Queries

1.1 Creating general boolean expressions

The descriptions of RABBIT are disguised boolean expressions in which ANDing is implicit (and the default case) while ORing and negation are explicit. But the operations provided do not allow arbitrary boolean expressions to be created (subject, of course, to the constraints imposed by KL-ONF). As a result, certain kinds of useful queries are difficult to express or modify while other kinds are inexpressible in the current implementation. For example, there is no easy way to specify "a restaurant or a hotel in Palo Alto" or to change the value constraint of an attribute-value pair from "Visa or MasterCharge" to "Visa, MasterCharge, or PersonalCheck" or to "Visa or PersonalCheck." Examples of descriptions which can not be expressed at all are the following:

(a)  
Restaurant
   location:
   |PaloAlto
or
   |City
   |near: PaloAlto

(b)  
Restaurant
   payment:
   |PersonalCheck
or
   |Visa
   |MasterCharge

Figure 9.1: Two reasonable, but inexpressible, queries

where the first description describes "a restaurant in or near Palo Alto," and the second specifies "a restaurant which accepts either personal checks or both Visa and MasterCharge."

One of the problems here is that the user can not, in general, replace one term with another. In the present implementation, the only way to replace a term is to AND in the replacement and then delete the old term. This technique, of course, only works when the user is replacing an entire descriptor and not just a part of a descriptor (such as a value in a value constraint). So a
special replace operator could be provided to allow general replacement of one term with another.

To allow almost arbitrary boolean expressions to be created, RABBIT could provide two additional operators: and and or, which AND and OR the selection with another descriptor (or value of a value constraint), respectively. A limited form of ORing is now handled by alternatives, which actually ANDs in a set of ORed terms, as opposed to ORing in new terms. The current operations of require, prohibit, and alternatives apply to the query description as a whole, whereas and and or would apply to selectable pieces of the description, such as a descriptor or a value of a value constraint.

The replace, and, and or commands increase the domain of expressible queries, but they can only be applied to those pieces of the description which the user can select with the mouse. An example of a description fragment which the user is presently unable to select is a single descriptor of an embedded description; the user can only select an embedded description as a single unit. One way of handling this limitation is to provide a "structure" editor, which would allow the user to select successively larger fragments of a description. [Our notion of a structure editor derives largely from early versions of an Interlisp-D structure editor designed and implemented by Beau Shell of Xerox PARC.] For example, suppose 'City' were selected in Fig. 9.1a. With a structure editor, reselecting 'City' would select the immediately enclosing "descriptive chunk"—'City near: PaloAlto'. Successive reselections would select "PaloAlto or City near: Palo Alto," "location: PaloAlto or City near: PaloAlto," and finally, the entire description. The structure editor is, in effect, aware of the structure of the description and allows the user to edit that structure.

With the addition of the new commands replace, and, and or, and a structure editor, all the query descriptions mentioned earlier in this section are easily expressed.

1.2 Defining additional relationships between an attribute and its values: Set Operators

The current implementation provides only one relationship between an attribute and the values which are constrained to fill that attribute—at least one of the instances specified by the value constraint must be a filler of that attribute in the retrieved matching example(s). For example, the following query description specifies all restaurants which accept either Visa, MasterCharge, personal checks, or some combination of the three forms of payments:
—Attributes of the query—
Restaurant payment:
  Visa,
  MasterCharge,
  or PersonalCheck
—Attributes of the query—

Figure 9.2: ORed values of an attribute

The effect of ANDing values is created as shown in the following description, which describes a restaurant which accepts all three forms of payment. [When we say ANDing values, we really mean the AND of the attributes with those values and not the (uninteresting) description “Visa and MasterCharge and PersonalCheck,” which specifies items which are both Visa, MasterCharge, and PersonalCheck all at the same time.]

—Attributes of the query—
Restaurant payment: Visa
payment: MasterCharge
payment: PersonalCheck
—Attributes of the query—

Figure 9.3: ANDed values of an attribute

If we let \( v \) denote the set of actual values filling an attribute in a matching instance and let \( p \) denote the set of potential values specified by the value constraint of the attribute-value pair, then in the current implementation, the relationship \( S \) between \( v \) and \( p \) is \((v \cap p \neq \emptyset)\). A useful extension would be to allow other kinds of relationships, e.g.,

\[
\begin{align*}
|v \cap p| &= 1 \\
v &= p \\
v &\subseteq p \\
v &\supseteq p \\
v \cap p &= \emptyset \\
v - p &\neq \emptyset.
\end{align*}
\]

\((|v \cap p| = 1)\) specifies that exactly one of the values in \( p \) is in \( v \) (in effect, an exclusive OR). \((v = p)\) indicates that the actual fillers are exactly the instances in \( p \). \((v \subseteq p)\) corresponds to interpreting the value constraint as a value restriction in the KL-ONE sense while \((v \supseteq p)\) is equivalent to ANDing values. \((v \cap p = \emptyset)\) and \((v - p \neq \emptyset)\) are both forms of negation: \((v \cap p = \emptyset)\) is equivalent to the present notion of a prohibited attribute-value pair and indicates “values not in \( p \)”; \((v - p \neq \emptyset)\) specifies that the attribute must have “values other than \( p \),” which could still include values in \( p \).

Note that the preceding discussion assumes that the value constraint denotes a set of instances which are handled as a group, whereas the extension proposed in section 1.1 handles the instances specified by the value constraint individually, according to the kind of connective (AND or OR).
between instances. Thus, there are queries (e.g., Fig. 9.1b) which are expressible in section 1.1 but inexpressible here, and vice versa (e.g., “Find all businesses with officers who are all under 50 years of age”). However, both extensions can be merged together if we treat the value constraint as being a list of descriptions, where a description \( p_i \) can be a boolean combination of instances and descriptions. Then the relationship \( S \) specifies which of the \( p_i \) are to be compared with \( v \) while the actual comparison between \( p_i \) and \( v \) is done according to the method given in section 1.1. For example, Fig. 9.4 describes “a restaurant which accepts either Visa, a credit card issued by Bank America, or both Master Charge and personal checks” (\( S \) is \( v \land (p \neq \emptyset) \)).

---Attributes of the query---
Restaurant
payment (at least one of):
Visa,
| CreditCard
| issuer: BankAmerica,
MasterCharge and PersonalCheck
---Attributes of the query---

Figure 9.4: A query description in extended RABBIT

1.3 Other

Naming descriptions. Another useful extension to RABBIT is the ability for the user to name (embedded) descriptions and thus, re-use them elsewhere in his (main) description; in effect, descriptions can be shared. Alternatively, descriptions could be copied so that changes to one copy would not effect the other. The potential problem with re-using descriptions elsewhere is that it may be difficult to enforce the semantic constraints which previously prevented users from creating semantically improper descriptions. Currently, the only constraint between an embedded description (serving as a value constraint) and its associated attribute is that the embedded description must satisfy the value restriction. However, if more elaborate constraints (such as the structural descriptions to be described in chapter 11) are allowed, then enforcement may be nontrivial.

In addition to being re-used, named descriptions could also be used as “virtual” instance classes/concepts in the KL-ONE network [Actually, as pointed out in chapter 5, not all descriptions can be represented as concepts]. Then in subsequent sessions with the database, the user could move through the lattice of instance classes directly to a previously defined virtual instance class without having to specify intermediate instance classes and attribute-value constraint pairs.

Linking variables. A natural extension to named descriptions, which are, in effect, constants, is to allow variables which link together different parts of a description in much the same way as the example variables of QBE [Zloof 75]. KL-ONE role value maps [Fikes 81b] are a natural mechanism for representing a variable linking the values of two KL-ONE roles (e.g., the filler of
one role is the same as the filler of another role). An example of a useful description which could be expressed with variables is “a restaurant whose owner is its manager.” Role value maps are themselves a special case of a more general class of constraints, KL-ONE structural descriptions, which are discussed at greater length in the next chapter.

Inverse roles. An inverse role (in the KL-ONE domain) is simply a back pointer from a value restriction (value) of a role to the concept (individual) owning that role. For example, if 'Business' has the role 'location:' and the value restriction 'City', then 'City' has an implicit corresponding role 'location-of-business:' with value restriction 'Business'. Similarly, corresponding to 'McDonalds512' with role 'location: Cambridge' would be 'Cambridge' with role 'location-of-business: McDonalds512'. Inverse roles could be explicit (e.g., 'employer:' and its inverse 'employee:'), but in general, they are implicit since they represent (in the opinion of the database designer) a minor relationship or property which is only occasionally useful. With inverse roles, it would be possible to describe "a city which contains a manufacturer of computers" and "all cities containing a McDonalds."

One way of giving the user access to the inverse roles associated with a given individual is to create a new perspective for viewing that individual, which specifies the relationships in which that individual participates as the value. Then invoking alternative views could include this new perspective as one of the alternative ways of viewing the given individual.

Inverse roles (e.g., 'location-of-business:') could be implemented as roles generated automatically whenever their counterparts (e.g., 'location:') are created or when they are needed (e.g., when alternative views is invoked) but marked such that they are normally invisible to the user or forgotten after they have been used. However, unlike "normal" roles, inverse roles will frequently have no value. For example, 'State' might have the role 'capital: ' which is a 'City', but not all cities are the capitals of some state. Thus, most individuals of 'City' will have no filler for the inverse role 'capital-of-state:'.

An alternative solution, which avoids extraneous roles, is to implement inverse roles using qua concepts [Freeman 81, Fikes 81b]. A qua concept is a concept associated with a role which is a subconcept of the value restriction of that role. Any individual concept filling that role is an individual of that qua concept, and any individual of that qua concept must be a filler of that role for some (other) individual concept owning that role. In the above example of businesses and their locations, a qua concept, 'City-which-is-the-location-of-a-business', could be created and associated with the 'location:' role of 'Business'. Then exactly those cities which are the locations of businesses are subsumed under that qua concept. The inverse role is implicit in the qua concept.

Indicating preferences. The operations of require and prohibit do just that: they require the presence of a descriptor or prohibit its presence. This suggests the possibility of preferences, i.e.,
probabilistic matching (see [Oddy 77]). The user could prefer that the retrieved instances have a given descriptor or prefer that they do not. One method of implementing preferences is to retrieve the matching instances using only the required and prohibited descriptors, and then sort the retrieved examples such that examples with the preferred descriptors and without the unpreferred descriptors are closer to the front of the final list of matching examples. Another method could be to retrieve the “preferred example instances” first and then retrieve a random sampling of instances matching the query description formed by the original description minus the preferred/unpreferred descriptors.

**Manipulating numbers.** Numbers are presently treated like any other individual concept in the KL-ONE network; no special meaning, such as the fact that numbers can be added or compared, is associated with numbers. However, since a particular number is represented only once in the network, a test for equality can be performed. Future implementations will need to understand what a number is and how they can be manipulated and compared. Furthermore, numbers will have to be handled in a special way since it is clearly impractical to have an individual concept in the network for every number.

2. Extensions to the Functionality of RABBIT

2.1 Browsing

RABBIT supports a restricted form of browsing through the matching examples and through the perspectives for viewing a particular example instance, but not browsing from one part of the database to another. However, the ability to create embedded descriptions (see chapter 8) is very much like a browser between instance classes/instances linked by attributes, and with some slight modifications, could be turned into a browsing facility similar to PDB [Cattell 80] and ZOG [Robertson 81].

One change which needs to be made is the removal of the link between the new description window and the original partial description window since the new description being constructed will not be merged back into the original partial description. Another change that needs to be made is the inclusion of an inverse role in the initial partial description. For example, if the example instance in the original window is the particular business 'McDonalds512' and the user wants to browse through the list of employees given in the 'employees:' attribute, then in the new (browsing) window, only the employees of that particular McDonald's should be shown in the matching examples pane and not, for example, all people in the database who are employed. Having the inverse attribute-value pair 'employees-of-business: McDonalds512' in the partial description will ensure that only the employees of that particular business are initially retrieved and displayed. Now the user can construct a description and make retrievals as before
or select an attribute-value pair in the image and open a new browser. To move back to another
window, the user simply moves the mouse into that window. Since browsers are not automatically
"closed" (i.e., removed and erased from the workstation display) when the user goes to another
window, he will also need to be able to close windows explicitly.

2.2 Maintaining semantic consistency

The present implementation only uses KL-ONE decompositions for determining the alternatives to
an instance class (see chapter 6, section 3.3). But decompositions can also be used as a semantic
check. For example, if an instance class belonging to a disjoint (i.e., mutually exclusive) partition
is a descriptor of the partial description, then an attempt by the user to add another instance class
from that same partition should result in a warning message to the user that the concept he is now
trying to add is inconsistent with an instance class already present in the description. Similarly,
attempting to prohibit all instance classes of a complete decomposition should also result in a
warning message.

2.3 Direct user control of the perspective

Our notion of perspectives allows us to present instances in a coherent fashion which avoids much
of the confusion arising from showing all known information about an instance. But for
sufficiently complex instance descriptions, even the information presented from a perspective may
be too detailed and confusing to the user. One solution is to allow the user to control which
attributes and instance classes should and should not be displayed. For example, the user might
specify that an attribute should no longer be displayed on the screen because the value filling that
attribute is immaterial to him or because he has finished specifying the value constraint and does
not intend to modify it again.

2.4 Help facilities

Although RABBIT provides access to the terms (i.e., instance classes, attributes, and instances)
necessary for making a query, it is usually unable to explain the meaning of those terms [vanLehn
81]. Two cases in which the user can determine the meaning of unknown terms are: (1) if the
user does not understand the meaning of a value of an attribute, he can use describe to see a
description of that value, and (2) if he does not understand the meaning of the name of an
attribute, he can examine the possible values which can fill that attribute or compare that attribute
with other attributes in the perspective. But in general, there is, as yet, no mechanism in RABBIT
for examining the definition or description of an arbitrary term.
Some possible techniques for explaining the meanings of instance classes using information in the KL-ONE network are the following:

1) KL-ONE associates a comment field with each instance class, instance, and attribute. The simplest technique would be to allow the user to see that comment upon request. However, the network designer is now burdened with providing the definitions of instance classes, instances, and attributes.

2) Alternatively, RABBIT could define an instance class to a user, upon request, in terms of the superclass(es) of that instance class and the value restrictions and/or values which that instance class assigns to the attributes inherited from those superclass(es). For example, if the instance class 'Person' has attribute 'sex:', and the subclass 'Man' fills 'sex:' with the value 'Male', then a user asking the meaning of 'Man' could be told that "a Man is a Person with sex Male" on the basis of the KL-ONE structure (see Fig. 9.5). In this example, the explanation is a definition; however, in many cases, a similar explanation would only be a description. For example, "an Elephant is a Mammal with a trunk," but not every mammal with a trunk is necessarily an elephant.

3) A third way is to define an instance class in terms of its superclass(es) and its subclass(es). For example, suppose we have the instance class 'LegalObject' with subclasses 'Person', 'Corporation', and 'LegalPerson', and that 'Person' and 'Corporation' are subclasses of 'LegalPerson'. Then a 'LegalPerson' could be described as "a LegalObject which is a Person or a Corporation" (see Fig. 9.6).

Another type of help facility needed is the ability to query the interface regarding the function and usage of the various menu commands. A possible solution is to adopt the same technique used in Interlisp-D [Interlisp 81] where selecting a menu command for a period of several seconds causes the system to display on the screen a message explaining the use of the selected command.

2.5 Assistance for the expert user

RABBIT's ability to remind users of descriptive terms via the images of example instances makes it well-suited for casual users, but we feel that expert users can also benefit from this interface. The main emphasis in the development of RABBIT has been oriented towards assisting the casual user, but certain properties of the retrieval by reformulation paradigm should also prove useful for expert users. For example, RABBIT's semantic checking capabilities can provide diagnostic help to an expert user who discovers that his query has resulted in no matching instances. The ability to name descriptions, described earlier, should also prove useful to expert users since they could then create and save away templates for commonly used query descriptions. Selection from menus will most likely prove too slow an input medium for experts, and thus, future implementations
should accept typing in from a keyboard. However, cleanly integrating typing with menu selection such that the user can switch from one mode to another will take careful design.

Figure 9.5: "Definition" of Man

Figure 9.6: "Definition" of LegalPerson
2.6 Other

Some other useful extensions include: an undo command (which was supported in earlier versions) which reverses the effects of the last command executed, a backup command in the description pane (or re-install command in the previous description pane) which initializes the partial description to be the previous partial description, and automatic window management (currently, the user must decide for himself where he wants windows to be placed on the screen).

3. Adapting RABBIT to Traditional Data Models

One important difference between traditional relational databases and KL-ONE databases is that a KL-ONE database is object-oriented—everything is an object (although some objects, such as names and numbers, have no internal structure and are, for all practical purposes, equivalent to the values of a relational database). An individual concept in a KL-ONE database is approximately equivalent to a collection of tuples in a relational database where the roles of an individual correspond to the attributes (fields) of the tuples. Similarly, a KL-ONE concept corresponds roughly to a relation. The value restriction for a role is equivalent to the domain (type) of the corresponding tuple attribute. The views of a relational database [Date 77] are analogous to the perspectives of RABBIT, but in general, relational databases do not structure their views. Nor are database views dynamic as in the case of RABBIT perspectives. If one were to impose a hierarchical (actually, heterarchical) ordering over the relations and incorporate an inheritance mechanism for relation fields and their associated domains, then there would be enough support for implementing RABBIT as it has been described in this thesis.

The Entity-Value-Relationship (EVR) data model [Cattell 82], which combines properties of both the relational data model and the entity-set model [Chen 76], already supports the notion of entities organized into a network and appears to be a particularly good candidate for use with RABBIT in its present form.
Chapter 10 - Areas for Further Research

1. Evaluation

We currently only have a demonstration model of an interface incorporating the ideas of our paradigm, and hence, at this time we can only make conjectures about the effectiveness of those ideas and RABBIT. Observations of actual users will need to be performed to determine what makes RABBIT better (or worse) than other existing interfaces and how much better (or worse).

The standard technique for evaluating a query language is to administer written tests to people. However, since a major feature of RABBIT is its highly interactive nature, the best approach to evaluating RABBIT is probably to videotape people using the interface. Some of the questions which need to be answered are:

1) learnability:  is the interface easily learned
2) expressiveness and usability:  can users express all the queries they might have, and is the (re)formulation of their queries easily performed, or do users require an “excessive” amount of time to find their desired items
3) errors and error recovery:  do users reach “dead ends” frequently, and if so, are they able to backtrack easily
4) relearnability:  is the interface easily relearned after a period of non-use

([Reisner 81] contains a more complete list of evaluation criteria for database query languages.)

In addition to determining the overall efficacy of RABBIT, we also need to determine how much each part of the interface (e.g., the instantiation) contributes to the utility of the interface as a whole. A possible approach to determining the contributions of the various pieces of RABBIT is to remove some piece from the implementation and observe how well users do without that feature.

2. Perspectives

Perspectives constitute an important part of the interface, but we have not as yet defined an epistemology for the construction and meaning of perspectives. In the current implementation, perspectives are represented as concepts in the KL-ONE conceptual hierarchy. However, in the course of building some sample databases, we have encountered situations in which the conceptual hierarchy does not adequately express the intended hierarchy of perspectives. For example, suppose that OEM products are products with OEM prices and that all computer related products
have OEM prices (actually, this is not true, but for the purposes of this example, suppose that it is true), then 'Computer-Related-Product' should be a subconcept of 'OEM-Product', which, in turn, is a subconcept of 'Product' (see Fig. 10.1a). Although 'OEM-Product' subsumes 'Computer-Related-Product' in the KL-ONE hierarchy, the user should not have to know about the concept 'OEM-Product' in order to reach 'Computer-Related-Product' since a 'Computer-Related-Product' is not inherently an 'OEM-Product'; that is to say, a 'Computer-Related-Product' is not by definition an 'OEM-Product'; it just happens that all 'Computer-Related-Product's are 'OEM-Product's. The hierarchy of perspectives might be better represented in Fig. 10.1b, where 'Computer-Related-Product' has been made an explicit subconcept of 'Product', but that newly introduced link is redundant with respect to inheritance [and is normally removed automatically by the KloneTalk system]. Further work needs to be done before we can answer questions such as what is a perspective, when does a perspective need to be created, how are perspectives related, and how do perspectives differ from concepts. The answers to those questions should allow a computer to generate a hierarchy of perspectives automatically, given a collection of instances.

Another area of future research is the possible application of perspectives to a problem of ZOG-like networks which Brown [1981] terms "conceptual jar." In jumping from node to node within a semantic network, the context used by the interface for displaying one node may differ radically from the context from which the preceding node was shown and thus, clash with the user's conceptual expectations. For example, suppose a surgeon is examining a description of the structure of the heart and then jumps to a description of the lungs. It is very likely that the surgeon would like to (and expect to) see a description of the structure of the lungs and not, for example, a functional description.

A possible solution to this problem, using perspectives, is to find the instance class which is the first common ancestor of the current node (instance) being viewed by the user and the preceding instance (i.e., the last instance viewed). The instance classes comprising the perspective for viewing the preceding instance and lying between 'Entity' and the common ancestor (in the KL-ONE hierarchy) can then be made instance classes of the initial perspective for the current instance. But a different strategy is needed for deciding which instance classes, if any, below the common ancestor and above the current instance should be included in the perspective. If a well-defined epistemology of perspectives can be developed, then it may be possible to compare the perspectives defined by different instance classes or by different decompositions (collections of instance classes) and thus, specify exactly the appropriate perspective for viewing the current instance. In this way, perspectives could be used to minimize or remove conceptual jar.
a) A hierarchy of concepts

b) A hierarchy of perspectives

Figure 10.1
3. Structural Descriptions

Presently, the only constraints between the parts of a description are those specified by the network designer (e.g., the value restriction of a role). More generally, a user might wish to express his own constraints within his partial description. Linking variables (see chapter 9, section 1.3) are an initial step in that direction. Role value maps, which are a possible means for implementing linking variables, are a special case of a more general KL-ONE construct, structural descriptions, which can be used to represent almost arbitrary constraints between the values of roles. The major issue here is that it may be extremely difficult to check whether the constraints imposed by the user are consistent. Previously, the interface was able to prevent a semantically improper description from being constructed, but now, the best that the interface may be able to do is to signal "no matching instances found," even if the reason why no matching example instances were found is that the query description was inconsistent; i.e., there could never be a matching instance. Some other important issues include deciding how the user should tell RABBIT his constraints, how the constraints should be represented on the workstation display, and how the user can modify previously entered constraints.

4. Backtracking

The current implementation of RABBIT does not support backtracking very gracefully. For example, when RABBIT can find no matching example instances, it simply indicates that fact to the user without giving him any advice as to what he can or should do next. What is really needed are some strategies for helping the user discover how he can modify his description such that there will be matching instances. One possible strategy is a histogram indicating the importance of each descriptor of the description [Williams 81a]. In particular, the user could be told for each descriptor how many instances would match the partial description if that descriptor had been removed. An examination of the techniques people use in backtracking might provide some insight into this problem.

5. Practical Databases

RABBIT, as it stands now, only supports information retrieval; the user can not add or delete information using RABBIT. KloneTalk, the language in which RABBIT has been implemented, already supports an interface for making changes to the contents and structure of the database, but that interface is unsuitable for casual users. However, it should be possible to modify RABBIT to allow the user to make changes to the database by running the interface "in reverse." For example, if the user wants to change the value of some attribute of some instance, he could simply retrieve that instance and then, somehow, change the value of the particular attribute in which he
is interested. If he wants to add a new instance to the database, he could retrieve an instance in
the database which looks very much like the new instance to be added except for the values filling
the attributes (i.e., the retrieved instance is an instance of the same instance classes as the new
instance should be) and then change those values accordingly and rename the newly created
instance. However, if the new instance does not resemble anything in the database, then either
the new instance needs to be made an instance of a new combination of instance classes, not used
by any other single instance in the network, or there are instance classes (and hence, attributes)
not present in the network which need to be added.

So far, RABBIT has only been used with very small ad hoc databases (on the order of 200 to 300
individual concepts and approximately 50 generic concepts). Further work needs to be done to
adapt the ideas of RABBIT for practical databases, where a large amount of data (i.e., several
thousand instances) is being changed and shared. For large databases, the current method of
presenting alternatives and specializations in menus is likely to be inadequate. As we noted earlier
in chapter 6, one possible solution is to use hierarchical menus (i.e., selecting from a menu of
items causes a second menu of specializations of the selection to appear), or if the number of
items is not too great, a scrollable menu.

Another area of the present RABBIT interface affected by a growth in size is the retrieval of all
matching instances in order to make a list of those matching instances available to the user in the
matching examples pane and to provide the domain from which the alternative values are found.
The alternative values could be drawn from the value restriction instead and thus, the search
would be independent of the number of matching instances; however, this solution also had its
disadvantages (as pointed out in chapter 6). With respect to retrieving the matching examples, a
possible strategy is to retrieve some subset of the matching examples which are “good”
(counter)examples. As we noted in chapter 4, well-chosen (counter)examples can assist a user
greatly, so RABBIT could concentrate its efforts on finding a few good (counter)examples. This is
an area in which an ability to infer additional information about the user’s intent (e.g., via the
history of the interaction or a pre-session questionnaire) and the current state of the database
would be most useful.

As the size and complexity of the database increases, one characteristic of the retrieval by
reformulation paradigm should begin to prove essential, namely, the notion of perspectives. With
so much information available in the database, some method of controlling the presentation of that
information to the user is a necessity.
6. What Is an Instance?

Although RABBIT relies heavily on the existence of individual concepts, we have not defined exactly what an instance is or how it differs from a (generic) concept. The sample databases of businesses, products, and people with which we have been working all seem to have well-defined individuals. However, the difference between a concept and an individual can sometimes be quite fuzzy; e.g., the car model 'Ford-Mustang' is an individual of the concept 'Car', but the particular car 'Jim's-Ford-Mustang' seems to be an individual of 'Ford-Mustang', which can not be true since individuals can not themselves have individuals. Deciding what are individuals, what are concepts, and how they are related is a general problem of knowledge representation.
Chapter 11 - Related Work

The ideas behind RABBIT have been drawn primarily from a theory of human remembering [Williams 81a, Williams 81c, Norman 79, Bobrow 75] and from KL-ONE [Brachman 79]. Related work on database interfaces includes the database query languages DIALOG, SQUARE, SQL, and THOMAS, and the graphical interfaces QBE, FORAL LP, ZOG, PDB, VIEW, and FINDIT. In particular, FINDIT was an early ancestor of RABBIT. Related work on the idea of perspectives includes KRL and PIE.

1. Database Query Languages

**DIALOG.** DIALOG [Lockheed 79] is a query language in which a user makes a query by constructing a boolean expression of keywords. Although boolean expressions are very expressive and precise, people often have trouble dealing with explicit boolean expressions [Cuff 80]. The query descriptions of RABBIT are also boolean expressions, but are closer in appearance to descriptions than to boolean expressions, in part, because the most commonly used boolean operator, AND, is implicit instead of explicit. A major difference between DIALOG and RABBIT is that DIALOG does not provide the user with a template of terms; a user of DIALOG simply uses any English words in his boolean expression and the system retrieves the texts (e.g., journal abstracts) containing those words. Thus, the user is presumed to have a reasonably good idea of what the relevant terms are. Otherwise, he must guess the appropriate terms. Although users of DIALOG are encouraged to formulate their queries in advance [Miastkowski 81] (one of the reasons being the charge for connecting to the database), in practice, users adopt an iterative approach—they formulate a query, retrieve some abstracts, and then use the (additional) keywords in those abstracts to reformulate their queries [Williams 81a].

**SQUARE and SQL.** SQUARE [Boyce 75] and SQL (formerly SEQUEL) [Chamberlin 76] are database query languages based on the relational data model [Codd 70]. Examples of SQUARE and SQL statements for the query “Find the names of employees in department 50” are:

SQUARE:

```
NAME EMP
('50')
```

SQL:

```
SELECT NAME
FROM EMP
WHERE DEPTNO = 50.
```

Both SQUARE and SQL have a syntax and semantics which require a significant amount of training to learn, on the order 12 to 14 hours of instruction [Riesner 75]. In addition, the user
must know in advance exactly which tables (relations) and columns (attributes) he will need for his query and what the legal table entries are.

THOMAS. THOMAS [Oddy 77] is an information retrieval interface which “[tries] to come to grips with the problem of serving a library user who is not able to formulate a precise query, and yet will recognize what he has been looking for when he sees it.” [p. 1] The basic mode of interaction is for the user to create a query consisting of a list of keywords by retrieving document references matching his current query and adding keywords selected from the retrieved references to his query. Thus, THOMAS is similar to RABBIT in its recognition of the problem of the possible inability of the user to formulate a precise query and in its solution of presenting examples to be used in reformulating the query. However, there are some important differences. For one thing, a retrieved reference consists of a title, author, publication information, and a list of keywords. Consequently, there is no structure to the information; the user is simply being given a list of keywords. The user also does not have access to alternatives to the given keywords. Another difference is that the permissible operations on a keyword are to specify that it is to be used in the subsequent retrieval process or that it is not to be used. In other words, saying “yes” to a keyword only increases the likelihood that that keyword is in the next reference retrieved while “no” only decreases the likelihood that it is present. In addition, when keywords are added to his query, they are implicitly ANDed together; there is no ORing. These differences arise from the fact that THOMAS is intended for users who are unable to specify exactly their queries and who thus, need to make their queries by browsing through the database.

An interesting property of THOMAS is its underlying metaphor of a dialogue between two parties. THOMAS makes strong inferences about the user's conceptual model (i.e., the user's conception of what he is searching for). If the user's model begins to differ substantially from THOMAS' model of what the user wants (as evidenced by the user's continuous rejection of retrieved references), then THOMAS backs up to an earlier state in the user's session with THOMAS and shows a retrieved reference which the user had approved of earlier. However, the burden of deciding that “something is wrong” is left to THOMAS since the user is not shown the query (i.e., the list of keywords) he has constructed so far and thus, the user can not explicitly remove previously accepted or rejected keywords.

2. Graphical Query Languages

Graphical interfaces are interfaces which make use of a graphical display (e.g., CRT or bit-map). They take advantage of the fact that recognition is usually easier than recall for the user [Cattell 80]. Furthermore, graphical images present an added dimension of information not available in simple text.
QBE. QBE [Zloof 75] is a graphical query language which utilizes a two-dimensional format. The user queries a relational database by filling in slots of tables displayed on a CRT. The tabular representation is, in effect, a template for constructing a description. ANDing and ORing are represented in the structure of the query and not by explicit keywords. A notable feature of QBE is its use of linking variables to express constraints. Such a mechanism is not provided in RABBIT [although it could be supported, as mentioned in chapter 9]. However, as in the cases of SQUARE and SQL, the user must still know in advance which tables and attributes he will be needing and what values are legal entries in those tables. In a study of QBE, the mean training time varied between two hours and two hours and 55 minutes [Thomas 75]. At the end of that amount of instruction, the subjects of that experiment still created incorrect queries about 33 percent of the time.

It should be noted that the “example” in Query by Example is not an example item in the database but rather “an example element [corresponding] to what is elsewhere called a variable, which gets bound to some specific value during a pattern matching or unification process” [Attardi 82].

FORAL LP. FORAL LP [Senko 77] is also a two-dimensional graphical interface, but one in which input from the user is made with a light pen. The user makes a query by touching nodes and arcs of a binary semantic network displayed on the screen. By constraining the user to select paths from a given network, FORAL LP reduces the possibility of non-meaningful queries. FORAL LP also requires that the user choose a “FORAL Context” which provides the context for the names of attributes. For large networks, the Context also serves to indicate what attributes are immediately visible, namely, the attributes of the Context node. However, since the network is not an inheritance network, there is no layering of information. In effect, the network is simply a set of relations represented in a network form. Another problem with FORAL LP is that it presents a view of the data schema, i.e., the data types and relationships, and not the actual data [Cattell 80]. Consequently, the user is forced to understand those relationships before he can gain access to the data.

ZOG and PDB. ZOG [Robertson 80] is an interface to a large network which relies on menu selection. “In ZOG, communication from man to computer is by discrete selection of semantically meaningful options [presented in menus]. Communication from computer to man is by visual display of natural language text in a structured format.” PDB [Cattell 80] is an entity-based database interface developed in connection with ZOG. The nodes of the network are the items, or entities [Chen 76], in the database. The edges of the network are the relationships (tuples) between entities. The user browses through the network a frame at a time, where a frame corresponds to an entity in the database and consists of the name of the entity and the relationships in which it takes part. To move to another frame, the user simply selects one of those relationships.
Both ZOG and PDB provide example instances to the user. The major difference between these two interfaces and RABBIT is that they allow a user to move in only one direction at a time; to move from one frame to the next, the user selects a single relationship, whereas a user of RABBIT can manipulate several relationships at once, simply by modifying more than one descriptor in his query description before performing the next retrieval of matching instances. The reason for this difference stems primarily from the fact that ZOG and PDB were designed to facilitate browsing as opposed to querying.

A problem observed with ZOG, although not unique to ZOG, is that users readily get lost. Another, possibly related, problem is that each frame holds so little data that moving to a new frame is an act of total replacement. There is also the problem mentioned earlier of "conceptual jar" in moving from one frame to the next.

VIEW. The VIEW System [Wilson 80], built upon SDMS [Herot 80], is a three-dimensional graphical interface which "combines techniques of graphics and knowledge representation to provide more complete and more perspicuously organized answers to user queries." The database consists of a semantic network representation of a set of relations and (pre-defined) predicate calculus rules for specifying constraints in the database. Information in the database is presented to the user based on semantic knowledge in the database and the user's context. The user's context consists of relations derived from the relations representing the database and are the relations to which the user has access. However, unlike RABBIT, where the user's context (perspective) is constructed and inferred, a user of VIEW must specify explicitly his desired context, chosen from a set of pre-defined contexts. VIEW also has the ability to retrieve information which is relevant to the user's query but not specifically requested. This information is actual data, whereas in RABBIT, it is structural information specifying the dimensions in which a new query can be described. For example, if a user of VIEW requested information about computer scientists working on database management systems, he would be shown not only information about computer scientists working on DBMS, but also non-computer scientists working on DBMS, computer scientists working on other projects, and non-computer scientists working on other projects.

The information in the database is presented as graphical icons with some text on a set of color, raster-scan displays. The user browses through the retrieved information by manipulating a joy stick which moves the display, viewed as a window, over the "data surface" of icons and text. The interface for making queries to the VIEW system is the underlying query language of the DBMS.
FINDIT

FINDIT [Weyer 76, Weyer 82a] is an early ancestor of RABBIT. In FINDIT, the user makes a query by filling in a form (template) and then using that filled-in form to retrieve matching examples. The user can then select keywords in the retrieved examples and copy them into the form for the purpose of performing another retrieval. What FINDIT does not provide is a structure to the keywords in the form and hence, there is no access to alternative keywords which can be used in the form (later versions of FINDIT did explore the possibility of providing a mechanism similar to RABBIT’s “alternatives” [Weyer 82b]). In addition, keywords can only be ANDed and ORed together; there is no negation. Finally, the structure and contents (i.e., fields) of the forms are fixed. So if the information is highly complex in structure and content, the associated form will similarly be very complex.

3. Knowledge Representation Languages

KRL. KRL [Bobrow 77] is a knowledge representation language which makes heavy use of perspectives. Bobrow and Winograd express their belief that “the description of a complex object or event cannot be broken down into a single set of primitives, but must be expressed through multiple views.” RABBIT extends that notion to include the idea of change based on the user’s conceptual model (as expressed in his partial description).

PIE. PIE [Goldstein 80] is not really a knowledge representation language but is rather an experimental personal information environment for supporting interactive development of programs, documents, etc. Information is represented as nodes in a network. Each node in turn can be assigned several perspectives. However, in PIE there is no structure between perspectives.
Chapter 12 - Summary

This thesis describes an information retrieval interface called RABBIT which relies on a new paradigm for information retrieval, *retrieval by reformulation*, based on a psychological theory of human remembering. The four main ideas underlying this paradigm are:

1) retrieval by constructed descriptions
2) interactive construction of queries
3) critique of example instances
4) dynamic perspectives

To make a query, the user interactively constructs a *description* of the item(s) sought. A description is a sequence of descriptors and thus, is multi-dimensional; each descriptor represents a dimension along which the target item can be described. To assist him in the construction of that query description, RABBIT provides an *example instance*, an item in the database which matches the user's current (partial) query description. This example serves as a *template* of descriptors which are likely to be relevant to the description of the target item. In addition, RABBIT provides operations which give the user *access* to alternative descriptors and which thus, reveal the structure of the descriptors. However, if the example is not the desired target item, as is likely to be the case initially, then the retrieved example instance is actually a *counterexample*, which makes the user aware not only of the fact that his partial description is incomplete or incorrect, but also of the directions in which the description needs to be further modified. Thus, by iteratively criticizing successive example (and counterexample) instances, the user can *reformulate* his query from an initially very vague characterization to a very precise one.

In addition to being an iterative process, retrieval by reformulation is also recursive. If in the course of his reformulation, the user encounters a descriptor which he does not understand, he can examine the description of that descriptor or recursively *describe* what the descriptor should be.

The instances in the database are potentially very complex in structure, especially if real world information is being stored and retrieved. So instances are always presented from some definite *perspective*, inferred from the user's query description. This restricted presentation, which makes use of the KL-ONE epistemology, reminds the user of terms likely to be relevant to his query description, enhances the user's understanding of the meanings of given terms, and prevents the user from creating semantically meaningless query descriptions.

Since RABBIT provides the user with a template and access to the descriptors, it facilitates the interactions of users who approach a database with only a vague idea of what it is that they want and who thus, need to be guided in the reformulation of their queries. RABBIT can also provide
valuable assistance to casual users who have limited knowledge about a given database or who must deal with a multitude of databases.

An experimental implementation of RABBIT has been written in the Smalltalk programming language and runs on the Xerox PARC Dolphin and Dorado personal computers. It is a graphical interface in which user input is through selection from pop-up menus. This working model looks very promising, but only usage by real users can determine the effectiveness and usefulness of the paradigm of retrieval by reformulation.
Bibliography


Biographical Note

Frederich N. Tou was born July 28, 1959, in Lafayette, Indiana and grew up in Gainesville, Florida. In 1977 he entered MIT as a freshman. At MIT he was a member of Eta Kappa Nu and Tau Beta Pi and actively participated in the Concert Band, Lecture Series Committee, and Musical Theatre Guild. His master's thesis research was conducted at the Xerox Palo Alto Research Centers in Palo Alto, California under the auspices of the MIT VI-A Cooperative Program.