220 02 80 62

LEVEL II (12)

MECHANICAL INTELLIGENCE: RESEARCH AND APPLICATIONS

AD A073127

Final Technical Report
Covering the Period October 1, 1977 to September 30, 1978

August 1979

SRI Project 6891
Contract No. N00039-78-C-0060
ARPA Order No. 3175

Contract Amount: $552,258
Effective Date: December 9, 1977
Expiration Date: September 30, 1978

DDC
RECEIVED
AUG 21 1979
C

By: Robert C. Moore, Computer Scientist
Artificial Intelligence Center
Computer Science and Technology Division

Contributing Authors:

Norman Haas
Kurt Konolige
Robert C. Moore
Ann E. Robinson
Earl D. Sacerdoti
Daniel Sagalowicz

DDC FILE COPY

Prepared for:

Defense Advanced Research Projects Agency
Arlington, Virginia 22209

Approved for public release; distribution unlimited

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government.

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MNP
TWX: 910-373-1246

79 08 20 027

SRI International

MECHANICAL INTELLIGENCE: RESEARCH AND APPLICATIONS

Final Technical Report • 1 Oct 77 - 30 Sep 78
Covering the Period October 1, 1977 to September 30, 1978

August 1979

104 p.

SRI Project 6391
Contract No. N00039-78-C-0060
ARPA Order No. 3175

Contract Amount: $552,258
Effective Date: December 9, 1977
Expiration Date: September 30, 1978

By: Robert C. Moore, Computer Scientist
Artificial Intelligence Center
Computer Science and Technology Division

Contributing Authors:

Norman Haas
Kurt Konolige
Robert C. Moore
Ann E. Robinson
Earl D. Sacerdoti
Daniel Sagalowicz

Prepared for:

Defense Advanced Research Projects Agency
Arlington Virginia 22209

Approved:

Peter E. Hart, Director
Artificial Intelligence Center

David H. Brandin, Executive Director
Computer Science and Technology Division

410 667

ABSTRACT

Since 1976 the Artificial Intelligence Center of SRI International has been conducting a program of research on ways of providing nontechnicians with easy access to complex, distributed data bases of information. This program has emphasized mutually supporting lines of both short-term and long-term research. The short-term research has resulted in an operational computer system for natural-language access to a distributed data base. The LADDER system (Language Access to Distributed Data with Error Recovery) is designed to provide answers to questions posed at the terminal in a subset of natural language regarding a distributed data base of naval command-control information. The system accepts a rather wide range of natural-language questions about the data, and for each question plans a sequence of appropriate queries to the data base management system; determines on which machines the queries are to be processed; establishes links to those machines over the ARPANET; monitors the processing of the queries and recovers from certain errors in execution; and prepares a relevant answer to the original question.

The first-generation LADDER system was completed by September, 1977. In October, 1977, work was begun on a second-generation LADDER system that dramatically extends the capabilities of the first-generation system along several dimensions. This report describes the evolution of the new system.

→ Section I of this report gives some background information on the LADDER system, outlines the changes made to the architecture of the system, and briefly explains the enhanced capabilities produced by those changes. Section II discusses user experiences with the first-generation LADDER system and the response of SRI to the reports of those experiences. Section III describes new user features that have been added to LADDER. Section IV discusses SODA, an improved data access system for LADDER, explaining its new capabilities and the problems of supporting those capabilities in accessing distributed data. Section V describes how the system has been extended to access a heterogeneous data base consisting of both Datacomputer and DBMS-20 data base management systems. Section VI reports on progress to date in bringing the results of our longer-term research into the LADDER system, in the form of a new natural-language processor that will permit a greater range of natural-language questions and lay the groundwork for transporting the system to new data bases and new domains. Section VII lists the publications and presentations by the project staff during the period covered by this report. Finally, Appendix A gives more detail on the new formal query language for data access, and Appendix B describes an experimental French-language version of LADDER.

CONTENTS

LIST OF ILLUSTRATIONS iv

I INTRODUCTION 1

A. EVOLUTION OF A NATURAL-LANGUAGE INTERFACE TO COMPLEX DATA 1

B. BACKGROUND INFORMATION ON LADDER 3

C. OVERVIEW OF THE SECOND-GENERATION SYSTEM 5

II EXPERIMENTS WITH LADDER 10

A. THE DATA AND OUR RESPONSE 10

B. THE VALUE OF EXPERIMENTATION 11

C. CONCLUSIONS 13

D. QUESTIONS HANDLED BY THE NEW LADDER 14

III NEW USER FEATURES OF THE LADDER SYSTEM 17

A. QUESTIONS INVOLVING CALCULATIONS 17

B. DISPLAYING DATA GRAPHICALLY 18

C. EXTENDING THE RANGE OF QUESTIONS 18

D. ELLIPTICAL QUESTIONS AND COMMANDS 20

E. MONITORING SYSTEM BEHAVIOR 21

IV HANDLING COMPLEX QUERIES IN A DISTRIBUTED DATA BASE . . . 23

A. INTRODUCTION 23

B. EXPRESSING COMPLEX QUERIES IN SODA 25

C. EXPANDING VIRTUAL RELATIONS AND ORDERING ACESSES TO RELATIONS 30

D. PROBLEMS IN DISTRIBUTED PROCESSING OF COMPLEX QUERIES 33

E. DISTRIBUTED QUERY PROCESSING IN SODA 39

F. LIMITATIONS AND POSSIBLE EXTENSIONS OF SODA 43

V ACCESSING A CODASYL DATA BASE SYSTEM: DBMS-20 47

A. INTRODUCTION 47

B. COMPILING SODA QUERIES 48

C. COMPARING RELATIONAL TO CODASYL DBMS'S FOR INTERACTIVE QUERYING 60

D. CONCLUSION 61

VI TOWARD A GENERAL NATURAL-LANGUAGE INTERFACE 63

A. OVERVIEW 63

B. THE DIAMOND PARSER 64

C. SEMANTIC PRIMITIVES 68

D. CONCEPTUAL SCHEMA 70

E. GENERATING SODA QUERIES 76

F. PORTABILITY 80

G. STATE OF IMPLEMENTATION AND PRELIMINARY RESULTS . . . 81

VII PUBLICATIONS AND PRESENTATIONS 82

A. PUBLICATIONS 82

B. PRESENTATIONS 83

APPENDICES

A FORMAL DEFINITION OF THE SODA QUERY LANGUAGE 87

B AN EXPERIMENTAL FRENCH-LANGUAGE LADDER 94

REFERENCES 96

Accession For
NTIS GRA&I
DDC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or special

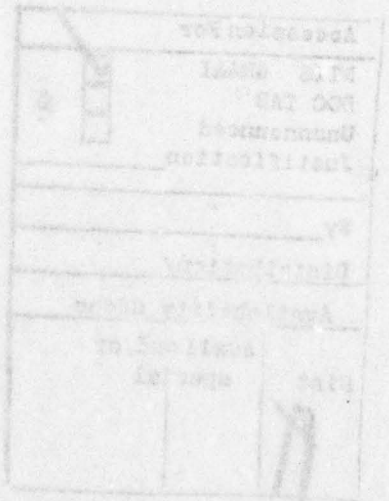
ILLUSTRATIONS

1 Sample Noun Phrase Rule 66

2 Conceptual Schema Taxonomy 74

3 Conceptual Schema Delineations 75

4 Generation of a SODA Query from the Conceptual Schema . . . 78



I INTRODUCTION

A. EVOLUTION OF A NATURAL-LANGUAGE INTERFACE TO COMPLEX DATA

Since 1976 the Artificial Intelligence Center of SRI International has been conducting a program of research on ways of providing nontechnicians with easy access to complex, distributed data bases of information. This program has emphasized mutually supporting lines of both short-term and long-term research. The short-term research has resulted in an operational computer system for natural-language access to a distributed data base. The LADDER system (Language Access to Distributed Data with Error Recovery) is designed to provide answers to questions posed at the terminal in a subset of natural language regarding a distributed data base of naval command-control information. The system accepts a rather wide range of natural-language questions about the data, and for each question:

(1) Plans a sequence of appropriate queries to the data base management system.

(2) Determines on which machines the queries are to be processed.

(3) Establishes links to those machines over the ARPANET.

(4) Monitors the processing of the queries and recovers from certain errors in execution.

(5) Prepares a relevant answer to the original question.

Work on LADDER is being carried out in support of the Advanced Command Control Architectural Testbed (ACCAT) program under the sponsorship of the Defense Advanced Research Projects Agency (DARPA). The ACCAT program is intended to provide a facility for transferring emerging information processing technology to Navy command-control applications. While the direct application of SRI's effort has been to develop prototype systems to aid in naval command and control, the software tools that have been created and the concepts underlying them

offer potential aid to decision makers in the other services, government, and industry as well.

The first-generation LADDER system was completed by September, 1977, and has been extensively described in the literature [1] [2] [3] [4] [5] [6] [7]. In October, 1977, work was begun on a second-generation LADDER system that dramatically extends the capabilities of the first-generation system along several dimensions. This report describes the evolution of the new system.

The remainder of Section I gives some background information on the LADDER system, outlines the changes made to the architecture of the system, and briefly explains the enhanced capabilities produced by those changes. Section II discusses user experiences with the first-generation LADDER system and the response of SRI to the reports of those experiences. Section III describes new user features that have been added to LADDER. Section IV discusses SODA, an improved data access system for LADDER, explaining its new capabilities and the problems of supporting those capabilities in accessing distributed data. Section V describes how the system has been extended to access a heterogeneous data base consisting of both Datacomputer [8] and DBMS-20 [9] data base management systems (DBMSs). Section VI reports on progress to date in bringing the results of our longer-term research into the LADDER system, in the form of a new natural-language processor that will permit a greater range of natural-language questions and lay the groundwork for transporting the system to new data bases and new domains. Section VII lists the publications and presentations by the project staff during the period covered by this report. Finally, Appendix A gives more detail on the new formal query language for data access, and Appendix B describes an experimental French-language version of LADDER.

The work decribed in this report reflects efforts by an integrated group at SRI performing research in natural-language access to data bases along a broad spectrum from the creation of demonstration systems to advanced research in computer understanding of natural language. Members of the group during the period covered by this report include

Barbara J. Grosz, Norman Haas, Gary G. Hendrix, Jerry R. Hobbs, Kurt Konolige, Robert C. Moore, Staffan Lof (now at the KVAL Institute for Information Science, Stockholm), Nils J. Nilsson, Gordon S. Novak, Jr. (now at the University of Texas), Ann E. Robinson, Jane J. Robinson, Earl D. Sacerdoti, Daniel Sagalowicz, Jonathan Slocum (now at the University of Texas), and B. Michael Wilber. Staffan Lof participated in the research program as an International Fellow at SRI under sponsorship of the National Defense Research Institute in Sundbyberg, Sweden.

B. BACKGROUND INFORMATION ON LADDER

1. Implementation

The LADDER system is written in INTERLISP [10], and the current version uses SRI's proprietary LIFER package [4] [5] for building natural-language interfaces. LADDER has been operational since June, 1976, and has been installed on a PDP-10 in the ACCAT facility at the Naval Ocean Systems Center (NOSC) since January, 1977. As of October, 1978, LADDER was also installed on three hosts on the ARPANET: SRI-KL and SRI-KA at SRI International, and ISIB at the Information Sciences Institute of the University of Southern California (ISI). At SRI-KL, LADDER runs under the TOPS-20 operating system; at the other sites it runs under TENEX. LADDER is used to access the Blue File and FC data bases, which are described in detail elsewhere [11] [12]. They are currently stored on the Datacomputer DBMS [8] developed by Computer Corporation of America and the DBMS-20 system [9] of Digital Equipment Corporation.

2. Basic Capabilities of LADDER

To provide an understanding of the context in which we did the work described in this report, we give the following examples illustrating the basic question-answering capabilities of LADDER. Some of the more advanced features of the system will be reviewed later in Section III.

An attempt has been made to accept a wide range of English-language inputs that are relevant to the data base and to the task of naval command-control decision making. One simple, but very common, type of question is to ask what ships satisfy a given set of restrictions. The user can ask for ships of any particular class (e.g., Kitty Hawk), type (e.g., cargo freighter), or naval classification (e.g., SSBN). Examples of simple restriction-type questions are:

```
NAME THE LOS ANGELES CLASS SUBMARINES.
WHAT SHIPS ARE HEAVY CRUISERS?
LIST THE SHIPS OF TYPE DDG.
```

Additional restrictions can be specified by appending a country, kind of operation, or distinguishing feature. For example:

```
IS THE FOX AN AMERICAN CRUISER?
PRINT THE NUCLEAR POWERED NAVAL VESSELS.
WHAT IS THE FASTEST DUTCH MERCHANT SHIP?
```

Questions can ask for more complex restrictions, such as comparisons of characteristics, comparisons with other ships, specifications of position, indications of route, cargo, or casualty status. For example:

```
ARE ANY SUBMARINES MORE THAN 300 FEET LONG?
WHAT AMERICAN NAVAL SHIPS ARE FASTER THAN THE FASTEST DUTCH
     MERCHANT SHIP
ARE THERE ANY FOREIGN CARGO FREIGHTERS WITHIN 300 MILES OF
     CAPETOWN?
NAME THE U S TANKERS WHOSE CURRENT SPEED OF ADVANCE IS LESS
     THAN 10 KNOTS.
REPORT ALL SHIPS CARRYING COALS TO LONDON.
DESCRIBE THE CRUISERS THAT ARE NOT AT READINESS RATING C1.
```

Additional types of modifications can be produced by specifying attributes of the ships. For example:

```
SHOW ME THE DESTROYERS WHOSE RADAR IS INOPERATIVE!
DO ANY SHIPS WITHIN 400 MILES OF LUANDA HAVE A DOCTOR ABOARD
WHAT ARE THE OILERS WHOSE LAST REPORTED POSITION IS WITHIN 250
     MILES?
NAME THE NEAREST SHIP TO THE KENNEDY WITH AN OPERATIONAL AIR
     SEARCH RADAR.
```

Most of the questions typically asked of a data base are concerned with the current values of attributes that are explicitly

stored. LADDER provides many formats for specifying such questions. The simplest forms ask explicitly for the attributes. For example:

WHAT IS THE RADIO CALL SIGN OF THE FOX?
WHAT IS THE STANDARD DISPLACEMENT OF EACH OILER WITHIN 400
NAUTICAL MILES OF GIBRALTAR
PRINT THE CURRENT POSITION AND FUEL STATUS OF THE DESTROYERS
IN THE MED!

Many more formats permit asking about attributes of ships in subtler ways. For example:

HOW IS THE SOUTH CAROLINA POWERED?
WHERE WERE THE OILERS LAST REPORTED
WHERE WILL EACH DUTCH CARGO FREIGHTER GO
HOW FAST IS EACH SOVIET MERCHANT VESSEL IN THE NORTH ATLANTIC?
WHEN IS THE CALIFORNIA SCHEDULED TO ARRIVE ON STATION
TO WHAT TASK GROUP DOES EACH DDG BELONG?
WHAT CLASS DOES THE HOEL BELONG TO
WHY IS THE AMERICA AT READINESS RATING C5?
WHO COMMANDS THE STERETT?

C. OVERVIEW OF THE SECOND-GENERATION SYSTEM

1. Architecture of the First-Generation System

The first-generation LADDER system consists of three principal components. The first component, INLAND (Informal Natural Language Access to Navy Data), accepts questions in a restricted subset of English and produces a query or queries addressed to the data base as a whole. The queries to the data base refer to specific fields, but make no mention of how the information in the data base is broken down into files.

The next component, IDA (Intelligent Data Access) breaks down a query against the entire data base into a sequence of queries against various files. IDA translates each query into Datalanguage, the query language supported by the Datacomputer DBMS, and composes the answers of the subqueries into the final answer returned to the user.

The task of dispatching the Datalanguage queries to the appropriate Datacomputer is handled by FAM (File Access Manager). This component searches a locally stored model for the primary location of

the file (or files) to which a query refers, establishes connections over the ARPANET to the appropriate computers, logs in, opens the files, and transmits the Datalanguage query. If at any time, the remote computer crashes, the file becomes inaccessible, or the network connection fails, FAM can recover and, if a backup file is mentioned in its model of file locations, it can establish a connection to a backup site and retransmit the query.

2. Natural-Language Interface

The second-generation system includes major modifications to all the components described above. In the INLAND natural-language interface, evolutionary improvements have been made in the form of the new user-oriented features described in Section III. These improvements include a route finding package that avoids land masses, an interface to the Situation Display Graphics Subsystem (SDGS) [13] for graphical display of data base information, enhanced capabilities to let the user define his own question forms and use elliptical inputs, and additional feedback in the form of natural-language paraphrases of the data base queries that are issued in response to the user's question.

In addition, substantial progress has been made toward bringing up a new natural-language front end that is the product of our earlier long-range research. As described in Section VI, this new system will have two major advantages over the present LIFER-based system when it becomes operational. First, because it contains a much more comprehensive, general grammar of English, it will enable LADDER to handle a much wider range of language forms. The difficulty with the current system is that it depends on a grammar that is based on semantically meaningful categories in the domain of application, such as ships or ports. In natural language, however, grammatical patterns cut across these categories. The result is that the system might be made to accept

WHO IS THE KENNEDY COMMANDED BY?

but not

WHO IS THE KENNEDY OWNED BY?

even though the active forms of both these questions are acceptable. The reason is that, since the semantic categories associated with OWNED and COMMANDED differ, it is difficult to state a "passive rule" that is applicable in all cases.

The other major advantage of the new front end is that it will make the system more easily portable to other domains and data bases. The grammar used in the first-generation system is tailored specifically to the domain of naval command and control. This produces some gains in efficiency, but it means that switching to a different domain of application requires completely rewriting the grammar. Since the new system uses a general grammar of English, we expect the grammar to be substantially the same over a wide range of applications. In switching from one domain to another, new vocabulary will have to be introduced, but new grammatical rules should not.

Another factor enhancing the portability of the system is that the new front end uses a model of the domain of application, called a "conceptual schema," that is independent of the data base. In the current system, data base queries are produced directly by the natural-language front end. Thus, if the data base is significantly changed, the front end must also be modified, even if the domain of application and the concepts used remain the same. With a conceptual schema, the issue of what information the user is seeking is kept distinct from the question of how that information can be retrieved from the data base. If the organization of the data base is changed, but no new concepts are added, the only changes required to the front end will be in the mapping from the conceptual schema to the data base.

3. Data Base Access

IDA, the data access component of the first-generation LADDER system, is very limited in the sorts of queries that it accepts. Basically, IDA queries can only select a single set of tuples from the data base applying simple boolean restrictions, perform some simple computation on the set, and return the result of the computation or a

projection of the set. This rules out many useful queries that involve more complex combinations of several sets of tuples. For example, if the data base contained multiple position reports for each ship taken over a period of time, it would be impossible to obtain the most recent postion report for each ship with a single IDA query. Retrieving this information would require searching the set of postion reports for each ship to find the most recent one and forming the set of the most recent reports for all the ships. The IDA query language cannot represent such a request.

To overcome the limitations of IDA, the second-generation system incorporates SODA (SOphisticated Data Access), a completely redesigned data access component described in Section IV. SODA accepts requests for information expressed in a much more powerful query language than IDA. Examples of queries that can be expressed in SODA, but not in IDA, include:

GIVE THE MOST RECENT POSITION FOR EACH AMERICAN SHIP.
WHICH AMERICAN SHIPS ARE LESS THAN 100 MILES FROM WHICH SUBMARINES?
HOW MANY SHIPS ARE IN EACH SHIP CLASS?
WHAT SHIP CLASSES HAVE THE MOST SHIPS IN THEM?
WHICH AMERICAN SHIPS ARE MORE THAN 500 MILES FROM EVERY AMERICAN PORT?

To process a query, SODA plans what relations at what data base sites must be accessed to retrieve the answer, constructs the necessary programs in the languages of the DBMSs involved, and requests movement of data among the data base sites. The problems of distributed query processing are much more difficult for SODA than for IDA because of the increased complexity of the queries handled. These issues are discussed in detail in Section IV.E, and the solutions chosen for implementation in SODA are explained in Section IV.F.

4. Accessing Multiple DBMSs

Although the first-generation LADDER system is able to retrieve information from a distributed data base, each data base site must use the same DBMS--the Datacomputer. The second-generation system

has been extended to access data base sites running Digital Equipment Corporation's DBMS-20 system as well. This extension is discussed in Section V. One of the reasons for choosing DBMS-20 was that it uses not only a different query language from the Datacomputer, but also a different data model. While we use the Datacomputer as a relational data base, DBMS-20 is based on the CODASYL [14] network-structured data model, giving us maximum heterogeneity among the sites in our distributed data base. The resulting system is, to our knowledge, the only existing operational system to provide uniform access to a truly heterogeneous data base. An example of query execution using both types of DBMS is given in Section V.B, and a discussion of the relative merits of relational and CODASYL DBMSs for interactive query processing is included in Section V.C.

II EXPERIMENTS WITH LADDER

A. THE DATA AND OUR RESPONSE

During the period covered by this report we acquired the first data from sizable groups of LADDER users who were not computer-oriented. This data came from two sources: an experiment run at NOSC evaluating LADDER in a simulated command-control environment [15], and transcripts from 19 students taking a course in the command-control curriculum at the Naval Postgraduate School (NPS). The questions that LADDER was unable to handle have been analyzed and found to fit into three major categories.

The first category is requests for information not in the data base. This was particularly detrimental to the performance of the NPS students. One of their major training aids was a workbook produced by NOSC that was oriented toward the FC (Pacific Ocean) data base; however, they were accessing the Blue File (Atlantic and Mediterranean) data base. While we cannot do anything about inaccessible data, we have improved the error messages to print out "<unknown-word> is not in LADDER's vocabulary" when a word--<unknown-word>--is not in the lexicon. This should prevent users from wasting time trying alternative syntactic constructions involving the word. The second category concerns questions involving distance or direction. The coverage of this kind of question has been broadened considerably, as suggested by the list presented in Section II.D.

The third category of questions not handled by the system is what the NOSC report calls "define word" and "define phrase." These were surprisingly difficult for the users to understand and use. A major source of their difficulty seems to have been that the NOSC workbook did not correctly specify their use. At the time the evaluations took place, the "define word" command required both the new word and the

model to be single words (e.g., "DEFINE CAN LIKE DESTROYER" worked, whereas "DEFINE LA LIKE LOS ANGELES" did not). However, the workbook used for training both the NOSC and NPS users contained examples of the form that did not work. It is understandable that users had trouble with the feature and felt unhappy about LADDER, since it did not do what they were told it would. Less significantly, two of the "define phrase" examples in the workbook also would not work as shown. (At least 7 of the NOSC users' errors--over 6 percent--appear to have stemmed from these mistakes in the workbook.) Nevertheless, the form of the command used in the NOSC workbook appears to be more natural and easier to use than our original form. We have therefore extended LADDER to accept the definition of new phrases on a phrase-by-phrase basis rather than by embedding them within a complete sentence (for example, by typing DEFINE "* PECOS" LIKE "WITHIN 700 MILES OF PECOS"). In addition, the error messages printed when a DEFINE command does not work have been expanded, so that users may learn how to use the DEFINE capability more easily. Also, spelling correction now is performed on the model sentence or sentences.

A major criticism in the NOSC report was that, when a question could not be handled by LADDER, it took far too long for LADDER to fail and print the error message. By the time the NPS students took their turn, and prior to our having seen the NOSC report, we had installed a new feature that doubles the speed of parsing acceptable queries and much more than doubles the speed of rejecting unacceptable ones. We had no complaints from the NPS students about the amount of time LADDER took to fail.

B. THE VALUE OF EXPERIMENTATION

We at SRI and our colleagues at ACCAT, NAVELEX, and ARPA had been saying to one another for two years that incremental feedback from people who are much closer than us to the operational community is essential for the development of LADDER-like systems. Although we had all been saying this, we got little such feedback until May, 1978. It

was surprising how many easily-closed holes in the system were uncovered by these evaluations. It is also surprising that, to a large extent, the same holes were fallen into by user after user. This suggests that it might have been worthwhile to have had more feedback earlier on.

This will become even more important as the issue of installing a system like LADDER in an operational environment is faced. The experiences of the NOSC and NPS users show very strongly that that installation must be an evolutionary process. It cannot be done, for example, by spending a few days at CINCPACFLT, coming home to work for a year, and then showing up on their doorstep with the "completed" system. It should not take much time from the operators, but some short interaction every few months seems essential to the development process. This may sound like a truism regarding bringing new systems into operation, but it will be truer than ever in trying to develop a system based on LADDER whose claim is that it works on the user's own terms.

In summarizing the results of our response to the NOSC and NPS users' experiences, it appears that the new LADDER would now handle about 90 percent of the NOSC questions. Most of the rest (e.g. "What nation Pecos what owner," "What is distance of Pecos," "Who is own," "When arrival Knox at Pecos?") are questions that either do not have a well-defined meaning or are simply beyond our current real-time processing capabilities. Examples of inputs to LADDER that failed in the NOSC experiments and now appear to work are given in Section II.D. Thanks are due to Curt Blais and Hal Miller at NOSC and Gary Poock at NPS for their help in providing us with the wealth of data. A second class of command-control students are scheduled to try using LADDER in the summer of 1979. The experience of this group will provide an important indication of how rapidly a natural-language access system such as LADDER can be made to converge on an acceptably high coverage of the relevant questions that users wish to ask.

C. CONCLUSIONS

We have drawn several conclusions from examining the body of data gathered during these experiments. These are subjective evaluations that come from an admittedly technology-oriented perspective, but we nonetheless believe they are valid.

1. Importance of User Feedback

The most critical need we saw reflected in the data was for informative and timely feedback to the user. At every stage of the query process our users would have doubts about the system's performance. In response to their expressions of insecurity with respect to the computer system, we installed the following features:

(1) A paraphrase in English of each query to the data base.

(2) A character printed on the screen every three seconds while LADDER is waiting for response from the Datacomputer, to assure the user that the host machine and the LADDER software are still operational.

(3) Features enabling each user to check what extensions he has made to the basic language accepted by the system.

(4) Improved error messages to provide more information about why a query failed. In particular, a special message was provided to be printed when a user uses a word that is not in LADDER's vocabulary, since this often implies that he is asking about information not in the data base.

2. Importance of Flexibility

The experience of our users shows clearly that a good interface must not only accept grammatically correct natural-language inputs, but must attempt to determine the meaning of as wide a range as possible of incorrect inputs. This supports several distinguishing aspects of our approach to natural-language interfaces:

(1) Spelling Correction -- The unsatisfactory nature of the standard keyboard as a means of input for military decision makers is clear. The NOSC experiment was carried out with a Tektronix 4051 as a front end to LADDER. After the user typed in his query, he had an opportunity to check and edit it before it was sent to

LADDER. Nevertheless, 23 per cent of the queries contained a typing error. The spelling correction capability of LADDER appears to be its most attractive feature to new users.

(2) Ellipsis -- For maximum efficiency, users' queries should be as short as possible. The results of our initial experiments indicate that users are very creative in shortening their inputs. LADDER's ability to process elliptical inputs has been extended to accept more kinds of shortened queries, but further research in this area appears to be needed.

3. Apparent Adaptability of LADDER

Although a natural-language interface may function according to specifications, it cannot be viewed as a tool of potentially wide utility unless it can be easily changed as the specifications change. If we view the NOSC and NPS users' experiences as providing a modified set of specifications, LADDER appears to be sufficiently adaptable, at least with respect to a given data base. An important thrust of our next year's work will be to extend this adaptability to new data bases as well.

4. A Testbed System or a Demo System?

Many of the gaps in coverage of users' queries resulted from a difference between the LADDER system's normal use as a demostration vehicle and its experimental use in a testbed environment. Although LADDER can be extended to serve both functions, it is difficult to evaluate it as a purely testbed system when it has been "detuned" to function primarily as a demonstration vehicle.

D. QUESTIONS HANDLED BY THE NEW LADDER

MAP SELECTION AND DISPLAY COMMANDS

Select map 200 miles from Pecos
Select a map 200 miles from Pecos
Show all ships 200 miles from Pecos
Select a map of area within 700 miles of Pecos
Select a map of 1000 miles around 37.66n,174.5w
Select a map of 1000 miles from 37.66n,174.5w

List (show) all ships in area
Select map from 37-40n,174-30w

GENERAL/SPECIFIC INFORMATION QUERIES

Who is the owner
Who is OPCON of SAR-1
List port of departure and destination port of Pecos
Where is Pecos coming from
What is port of departure and port of destination of Pecos
Name of OPCON of SAR-1
What is Pecos port of registery
What is the home port of Pecos
What is ... of the listed ships
What is ... of the ships on the list

TIME COMPUTATION QUERIES

What is the time for Rathburne to reach Pecos
How long for Rathburne to reach Pecos

DISTANCE QUERIES

Display the distance of Pecos from here
What is distance of Pecos (presumed "from here")
What is distance from Pecos to here
What is distance between Pecos and all ships within 700 miles
What is distance from Pecos to all ships within 700 miles
What is the distance from Pecos to Connie, Biddle, R K Turner,
Halsey, Adelaide Star
What is distance between Pecos and San Francisco
What is distance from Pecos to San Francisco
What is the distance of ships within 700 miles of the Pecos to Pecos
What is distance to Pecos
What is distance of these ships from Pecos
What is distance from Pecos to all ships within 700 miles of Pecos
What is distance from Pecos to constellation
List distances of all ships within 700 miles of Pecos
What is the distance of Pecos from Honolulu
What is the distance to Pecos of each ship
within 700 miles of Pecos
How far is Biddle from Pecos
How far is Pecos from all ships within 700 miles

DEFINITION OF WORDS AND PHRASES

Define (Daships) to be like (ships within 700 miles of Pecos)
Define (Prthst of Pecos) to be like (ports of departure of Pecos)
Define (name SAR-1) to be like (name Rathburne and Knox)
Define (? and Rathburne) like (what is distance from Pecos
and Rathburne)

```
Define distance like what is the distance from
Define (what is the port of Departure $)
        like (what is the port of departure of the Pecos)
Define (  length of the Pecos) like (what is the length of
        the pecos)
Define last to be like 37-40n,174-30w
Define (what is the distance to last) like (what is the distance
        to 37-40n, 174-30w)
Define (what is distance from here to last) like (what is distance
        from here to 37-40n, 174-30w)
Define (what is distance from Honolulu to last) like (what is
        distance from Honolulu to 37-40n, 174-30w)
Define (range of Kennedy from Honolulu) like (what is the distance
        of Kennedy from Honolulu)
Define (* Pecos) like (within 700 miles of the Pecos)
Define SAR-1 to be like Rathburne and Knox
Define (w) like (what is the)
Define (range) like (what is the distance of)
```

III NEW USER FEATURES OF THE LADDER SYSTEM

A. QUESTIONS INVOLVING CALCULATIONS

In this section we review some of the advanced features of LADDER and explain how they have been extended during the period covered by this report. Some of these features enable the system to combine computation with data base retrieval. This is a major advantage of having a computer serve as the interface between a decision maker and a data base, since the computer can perform complex calculations on the data retrieved much faster and more reliably than a person. During the past year, we have implemented some examples of this kind of capability in LADDER, but have not attempted to provide for all the calculations a naval decision maker might need.

LADDER attempts to handle questions concerning distances between ships, which involve calculations dependent upon position information retrieved from the data base, and questions concerning steaming times, which require position information as well as current and maximum speed values from the data base.

This past year, we have also implemented in LADDER a route calculation routine for avoiding land masses. This routine uses a model of the sea areas of the world and the junction points that must be traversed between them. The particular model used by the current version of LADDER is very simple and hence may give inexact answers; the performance of the routines would improve with a more detailed model. Some questions the user can ask include:

HOW MANY MILES IS THE CONSTELLATION FROM HER NEXT PORT OF CALL
HOW FAR IS EACH AMERICAN DESTROYER FROM THE SOVIET CARRIERS
WHAT SHIPS CARRYING DOCTORS ARE WITHIN EIGHT HOURS' STEAMING TIME OF THE PECOS?
WHAT IS THE NORMAL TRANSIT TIME FOR THE KENNEDY FROM NORFOLK TO GIBRALTAR
DOES THE SARATOGA HAVE ENOUGH FUEL TO REACH BUENOS AIRES WITHOUT REFUELING?

HOW LONG WOULD IT TAKE FOR THE WAINWRIGHT TO GET TO NAPLES
WHAT IS THE BEST ROUTE FOR THE SUNFISH TO THE SKORY?

B. DISPLAYING DATA GRAPHICALLY

During the past year, we have implemented a very simple interface with the Situation Display Graphics Subsystem [13], developed by the Information Science Institute (ISI) of the University of Southern California. Four special commands are provided to cause information to be displayed in map format.

Before displaying any data base information the user must direct LADDER to display a map. This is done with the SELECT command, which consists of the word "select" followed by a region specification. For example:

SELECT A MAP OF THE NORTH ATLANTIC
SELECT THE AREA WITHIN 500 NAUTICAL MILES OF THE WORDEN.

After a map is displayed, ships or sets of ships may be added to it or removed from it using the SHOW and ERASE commands. The context of these commands is presumed to be the area on the display. For example, after typing "Select a map of the Mediterranean," the command, "Display all the carriers" will cause only carriers in the Mediterranean to be retrieved from the data base and displayed.

An additional command is provided to permit a graphic image to be saved on disk. This command takes the form

SAVE <name> ,

where <name> is any valid file name.

C. EXTENDING THE RANGE OF QUESTIONS

It is impossible to provide a natural-language interface system such as LADDER with an ability to accept all the questions that could conceivably be asked about a given data base. Furthermore, frequent users will want to develop their own shorthand questions for accessing the data they often use. To meet these needs, LADDER allows each user to extend the grammar dynamically, by example, to handle new types of

questions. The DEFINE command is used to extend the grammar by adding a synonym, a new phrase, a paraphrase of a single question, or a paraphrase of a sequence of questions, which we call a macroparaphrase.

To add a synonym of a word that is known to the system, the user may just type

DEFINE <new-synonym> LIKE <known-word>.

For example,

DEFINE CONNIE LIKE CONSTELLATION

will permit a question such as

WHO COMMANDS CONNIE?

to be handled.

To add a new phrase, the user may type

DEFINE "<new-phrase>" LIKE "<known-phrase>"

where <known-phrase> is any sequence of words that the system could accept in some sentence. Either <new-phrase> or <known-phrase> can be a single word. Examples of this feature include:

```
DEFINE "MEDSHIPS" LIKE "SHIPS WITH A DOCTOR ABOARD"
DEFINE "TIN CAN" LIKE "DESTROYER"
DEFINE "SHIPS OF INTEREST" LIKE "SHIPS WITH A DOCTOR ABOARD
     WITHIN 400 MILES OF PECOS."
```

These new phrases are handled by LADDER by substituting the known phrase for the new phrase whenever it occurs in a question, before the parsing of the sentence begins. LADDER will retype the user's question with the substitution when it is performed.

LADDER permits the user to add an entirely new question format by example. To do so, the user must provide LADDER with an example of how the extension is to be used in the context of a complete sentence. This is done by typing,

DEFINE "<new-sentence>" LIKE "<known-sentence>",

where <known-sentence> can already be handled. For example,

```
DEFINE "CARRIERSTAT MEDITERRANEAN" LIKE "WHAT IS THE CURRENT
     POSITION, FUEL STATE, AND READINESS STATUS OF ALL
     CARRIERS IN THE MEDITERRANEAN"
```

will cause the new pattern

CARRIERSTAT <MACRO.LOC>

to be added to the grammar. Subsequently, questions like

CARRIERSTAT NORTH ATLANTIC

will be accepted by LADDER.

During the past year, we have provided a novel facility for allowing a new question to substitute for a sequence of old questions, each of which is already understood by LADDER. The define command is still used, but the model (the part following "like") can be a sequence of questions. For example,

DEFINE "GIVE AN OVERVIEW OF JFK" LIKE "WHAT IS THE TYPE, LENGTH, BEAM, AND DISPLACEMENT OF JFK? WHAT WEAPONS DOES SHE CARRY? WHO COMMANDS HER? WHAT IS HIS LINEAL NUMBER?"

will add to the grammar the pattern:

GIVE AN OVERVIEW <OF> <SHIP>.

this will permit questions such as,

GIVE AN OVERVIEW ABOUT ALL THE US SUBMARINES

to be answered.

When the define command is processed by LADDER, each sentence in the model is parsed (spelling correction will be performed if necessary) but the data base is not queried to answer the questions. When a macroparaphrase is processed by LADDER, each question in the model is typed out before LADDER proceeds to answer it.

D. ELLIPTICAL QUESTIONS AND COMMANDS

LADDER accepts not only complete sentences, but also sentence fragments that can be interpreted in the context of the previous sentence. The syntactic term for this condition, in which words of the second sentence are left out but implied, is ellipsis.

When an input cannot be interpreted as a complete sentence, LADDER types out the message, "trying ellipsis:", and then checks to see if it is analogous to any contiguous string of words in the previous sentence. If it is, the input is substituted for that string and the resulting new sentence is printed out. LADDER then proceeds to carry out the resulting request. Examples of valid elliptical inputs in the context of the previous question, "WHAT IS THE LENGTH OF THE SANTA INEZ" include:

THE BEAM AND DRAFT
HOME PORT OF THE AMERICAN CARRIERS
PRINT THE NATIONALITY
KITTY HAWK

During the past year, we have extended the ellipsis capability to handle phrases such as:

WHAT ABOUT X

where x is a sentence fragment. Thus LADDER will now accept the sequence:

WHAT IS THE LENGTH OF FOX?
WHAT ABOUT DRAFT?

Elliptical fragments can also be added to the end of the previous sentence, as in the sequence:

WHAT ARE THE US CARRIERS?
IN THE MED?

E. MONITORING SYSTEM BEHAVIOR

1. Paraphrasing Data Base Queries

When the user types a question to LADDER, it is printed on the terminal. In addition LADDER now produces a paraphrase of the queries to the data base required to reply to the question. This paraphrase provides a means for the user to check that LADDER has interpreted his question properly. There may be more than one paraphrase produced if more than one data base query is required to answer a given question. For example, if the user asks,

WHAT SHIPS WITH A DOCTOR ABOARD ARE WITHIN 900 NAUTICAL MILES OF THE BRITISH BOMBARDIER?

LADDER will print out

For SHIP equal to BRITISH BOMBARDIER, give the POSITION and DATE.

and, subsequently,

For DOCTR equal to D and great circle distance to 46-33N, 21-29W less than or equal to 900, give the SHIP.

(46-33N, 21-29W is the position of the British Bombardier determined from the previous query.) These two paraphrases together constitute LADDER's interpretation of the user's question.

2. Timing

In the past, users of LADDER have pointed out that there may be long periods when nothing seems to be happening. To alleviate this sense of frustration, LADDER now types a dot whenever a command is sent to the Datacomputer, and counts the number of three-second intervals that elapse as it waits for the answer. This will inform users that LADDER is functioning properly and awaiting action from the data base.

IV HANDLING COMPLEX QUERIES IN A DISTRIBUTED DATA BASE

A. INTRODUCTION

As part of the continuing development of the LADDER system, we have substantially expanded the capabilities of the data base access component that serves as the interface between the natural-language front end of LADDER and the data base management systems on which the data is actually stored. SODA, the new data base access component, goes beyond its predecessor IDA [6], in that it accepts a wider range of queries and accesses multiple DBMSs. This section is concerned with the first of these areas, and discusses how the expressive power of the query language was increased, how these changes affected query processing in a distributed data base, as well as what are some limitations of and planned extensions to the current system.

To explain the new features of SODA, it will be useful to review briefly the capabilities of IDA. IDA is designed to access a relational data base. That is, it expects the data base to be organized as a set of relations (files), each of which contains a set of tuples (records) that are in turn composed of various fields. The IDA query language permits the user to view the entire data base as if it were a single relation, with IDA being responsible for planning which actual data base relations have to be accessed to answer the query. An IDA query is interpreted as a request to:

(1) Generate the set of all tuples satisfying a given description expressed as a Boolean combination of simple comparisons on the fields of the tuple.

(2) (Possibly) select the member of the set for which some attribute is largest or smallest, or count the members of the set.

(3) Return the values of certain attributes for each member (or for the selected member) of the set.

For instance, in the Blue File command and control data base [11] which we have been using, the English query, "What is the longest American ship?" could be expressed using IDA as:

((* MAX LGHN)(NAT EQ 'US')(? NAM))

The term (NAT EQ 'US') tells IDA that the tuples we are interested in are those for which the NAT field has the value 'US', i.e the tuples pertaining to American ships. The term (* MAX LGHN) tells IDA that we want to select from this set of tuples the tuple for which the field LGHN has the highest value, i.e. the tuple for the longest American ship. Finally, the (? NAM) field tells IDA that we want to return the value of the field NAM from this tuple, i.e. the name of the longest American ship.

IDA would interpret this query by finding the smallest set of relations in the data base that contains all the fields mentioned in the query and specifying to the DBMS what it believes to be the semantically meaningful links among those relations. IDA then generates a program in the DBMS access language that interprets the query with respect to these selected relations.

This approach limits the expressive power of the query language in a number of ways. First, only one set of objects can be talked about in each query. The only way in which two sets of objects can be referenced is if the set that the query is "about" is their intersection or union. Thus we can express the query "Which ships are American submarines?" (intersection), or "Which ships are American or are submarines?" (union), but there is no way to express "Which American ships are less than 100 miles from which submarines?"

Another restriction is that only one maximization, minimization, or count operator is allowed in each query, and it must be applied after all other operations. For example, we cannot express as a single query "How many ships are in each ship class?" since this requires forming a set of counts, rather than simply counting a set. Also, we cannot express "Which ship class has the most ships in it?" since this requires a count operator and a maximization operator in the same query.

Finally, only Boolean restrictions are allowed in specifying a set of objects. That is, all restrictions must be simple comparisons between fields or predefined functions of fields (such as distance functions computed on position fields), or combinations of simple comparisons using AND and OR. Thus IDA gives us no way to express a restriction involving a quantifier as in "Which American ships are more than 500 miles from every American port?"

B. EXPRESSING COMPLEX QUERIES IN SODA

The features of the SODA query language enable it to overcome all of the limitations of IDA discussed in the previous subsection. It allows queries that refer to more than one set of objects, it permits queries to specify the logical scoping of operations, and it allows quantifiers to be used in specifying restrictions on sets of objects. This subsection informally discusses a number of examples which illustrate these points. The details of the syntax of the SODA query language may be found in Appendix A.

In the examples, we will assume that we have the following subset of a simplified Navy command and control data base:

SHIP: (NAM, CLASS, TYPE, NAT, LGHN, POS)

SHIPCLASS: (CLASS, TYPE, LGHN, DRAFT, BEAM)

PORT: (PNAM, PNAT, PPOS)

In the SHIP relation, NAM is the name of the ship, CLASS is her class, TYPE is her type (e.g. 'SS' for submarine), NAT is her nationality, LGHN is her length, and POS is her current position. The SHIPCLASS relation gives information that is common to all ships of the same class. CLASS, TYPE, and LGHN are as in the SHIP relation, and DRAFT and BEAM are the corresponding dimensions of the ships in the class. In the port relation, PNAM is the name of the port, PNAT is the country in which the port is located, and PPOS is the geographical position of the port. We will also assume that the DBMS has the ability to compute the function GCDIST, which gives the great circle distance between two geographical locations.

SODA avoids the first limitation of IDA, the inability to refer to more than one set of objects per query, by using an IN expression to associate a variable with each of the sets we want to mention in the query. The query "Which American ships are less than 100 miles from which submarines?" (which could not be expressed in IDA) can be expressed in SODA as:

```
((IN S1 SHIP ((S1 NAT) EQ 'US'))
 (IN S2 SHIP ((S2 TYPE) EQ 'SS'))
 ((GCDIST ((S1 POS) (S2 POS))) LT 100)
 (? (S1 NAM))
 (? (S2 NAM)))
```

In this SODA query the expression (IN S1 SHIP ((S1 NAT) EQ 'US')) sets the variable S1 to range over tuples in the SHIP relation for which the NAT field has the value 'US', i.e tuples for American ships. Similarly, the expression (IN S2 SHIP ((S2 TYPE) EQ 'SS')) causes S2 to range over tuples for submarines. Then for each pair of ships in the Cartesian product of these two sets, the additional restriction ((GCDIST ((S1 PTP) (S2 PTP))) LT 100) is applied. That is, we check whether the great circle distance between the two ships is less than 100 miles. For each pair of ships that satisfies all these restrictions, we return the names of the ships. This is indicated by the selectors (? (S1 NAM)) and (? (S2 NAM)).

We can illustrate SODA's ability to express the relative scoping of operations with the query, "How many ships are in each ship class?" This could be expressed in SODA as:

```
((IN C SHIPCLASS)
 (COUNT CNT1
        (IN S SHIP ((S CLASS) EQ (C CLASS))))
 (? (C CLASS))
 (? CNT1))
```

The form of a counting operation is a list where first element is the symbol COUNT, the second element is a count variable, and the rest of the list is a subquery which defines the set of tuples to be counted. The effect of a count operation is to set the value of the count

variable to the number of tuples in the indicated set. In this example, since the set to be counted is defined in terms of the field (C CLASS), and since this occurrence of C is bound outside of the COUNT expression and ranges over all tuples in the SHIPCLASS relation, the query is interpreted to mean that the count operation is to be performed once for every tuple in the SHIPCLASS relation. Thus, the interpretation of the entire query: for each tuple in the SHIPCLASS relation, count the number of tuples in the SHIP relation which have the corresponding value for the CLASS field and return the name of the class and the count.

An example of of a COUNT and a MAX in the same query is provided by the SODA representation of the query, "What ship classes have the most ships in them?":

```
((MAX CNT1
      (IN C SHIPCLASS)
      (COUNT CNT1
             (IN S SHIP ((S CLASS) EQ (C CLASS)))))
 (? (C CLASS))
 (? CNT1))
```

This query simply embeds the body of the preceding query inside a maximizing operation over the count variable. The basic form of a maximizing operation is a list where the first element is the symbol MAX, the second element is the term to be maximized, and the rest of the list is a subquery that defines the set of tuples to be maximized over. In this case the term to be maximized is CNT1 in the set consisting of the tuples in the SHIPCLASS relation augmented by the corresponding values of CNT1, the number of ships in each ship class. The effect of the MAX operation is to set the occurrences of the variables bound by the MAX (in this case C and CNT1)) to range over the values for which the maximized quantity has the greatest value. So in this example, the MAX operation sets the variable C to range over the tuples in the SHIPCLASS relation for the ship classes with the most ships in them and sets CNT1 to the corresponding number of ships. The rest of the query simply returns the name of those ship classes and the number of ships they contain.

Finally, SODA includes several types of quantifiers that can be used to express complex restrictions on the sets of objects referenced by queries. As an illustration of the use of a quantified restriction, recall that there was no way in IDA to express the query "Which American ships are more than 500 miles from every American port?" In SODA this could be expressed by:

```
((IN S SHIP ((S NAT) EQ 'US'))
 (ALL (IN P PORT ((P PNAT) EQ 'US'))
      ((GCDIST ((S POS) (P PPOS))) GT 500))
 (? (S NAM)))
```

The first line of this query restricts the system's attention to American ships via a simple restriction on the SHIP relation. The second expression in the query further restricts this set but involves a universal quantifier. The simplest form of a universally quantified restriction is a list consisting of the symbol ALL, an IN expression, and any number of restrictions. An ALL restriction is satisfied if all the tuples in the set specified by the IN expression satisfy all the restrictions in the list. If there is more than one binder expression in the list, then the join of the sets they specify must satisfy all the restrictions in the list.

In the current example, all the values of P that satisfy

```
(IN P PORT ((P PNAT) EQ 'US'))
```

must also satisfy

```
((GCDIST ((S POS) (P PPOS))) GT 500)
```

for the ALL restriction to be satisfied. Informally, this means that all American ports must be more than 500 miles from the ship in question, for that ship to meet this restriction. Finally, the NAM field from every tuple that meets these restrictions is returned to the user.

A SOME restriction has the same syntactic form as an ALL restriction, the difference in interpretation being that the restriction

is satisfied if some tuple in the set specified by the binder expressions satisfies the other restrictions within the SOME expression. Thus, to change the previous query to "Which American ships are more than 500 miles from some American port?" we only have to replace the ALL by a SOME:

```
((IN S SHIP ((S NAT) EQ 'US'))
 (SOME (IN P PORT ((P PNAT) EQ 'US'))
       ((GCDIST ((S POS) (P PPOS))) GT 500))
 (? (S NAM)))
```

Notice that in these examples, there are some restrictions placed inside the IN expression itself and some restrictions placed after the IN expression. In a SOME restriction this distinction is of little consequence, since placing a restriction one place or the other does not change the interpretation of the query. If we place a restriction inside an IN expression, we are using it to define the set that is being quantified over. This is equivalent, however, to quantifying over a less restricted set, but being more restrictive as to the additional conditions that one of the members of the set has to satisfy, which is the interpretation of placing a restriction outside the IN expression. Thus, we could have expressed the previous query by either of the following expressions:

```
((IN S SHIP ((S NAT) EQ 'US'))
 (SOME (IN P PORT)
       ((P PNAT) EQ 'US')
       ((GCDIST ((S POS) (P PPOS))) GT 500))
 (? (S NAM)))
```

or

```
((IN S SHIP ((S NAT) EQ 'US'))
 (SOME (IN P PORT ((P PNAT) EQ 'US')
                  ((GCDIST ((S POS) (P PPOS))) GT 500)))
 (? (S NAM)))
```

In an ALL restriction, however, this distinction is crucial. If we move a restriction from inside an IN expression to outside, the interpretation is changed completely, since instead of the restriction partially defining what set is being quantified over, it partially

defines the condition that all the members of the set must meet. In this respect the syntax of the SODA query language is designed to mirror the syntax of English, so that the process of translating from English to SODA will be simplified. The idea is that the restrictions derived from noun phrase modifiers like "American" in "all American ships" would be placed inside an IN expression, but restrictions that come from predicate modifiers would be placed outside the IN expression. If this rule is followed, then the resulting SODA queries will exactly capture the difference between "Are all American ships submarines?" and "Are all ships American submarines?" Conversely, the SODA queries for "Are some American ships submarines?" and "Are some ships American submarines?" will be logically equivalent, as are the English questions.

As a final point on this topic, it should be noted that, although the two questions with "some" must have the same answer, they do differ slightly in what they suggest about the assumptions of the person asking the question. "Are some American ships submarines?" suggests that he believes that there are American ships, whereas "Are some ships American submarines?" suggests only that he believes that there are ships. As Kaplan [16] has pointed out, it can be very important to inform the user of a data base system when the assumptions behind his queries are wrong, so that he can properly interpret the answers he gets from the system. The distinction in SODA between restrictions inside an IN expression and those outside could be used to differentiate the restrictions whose satisfiability the user is assuming, from those whose satisfiablity he is questioning.

C. EXPANDING VIRTUAL RELATIONS AND ORDERING ACESSES TO RELATIONS

In the previous subsection, we assumed that SODA always used the relations specified by IN-expressions to retrieve the requested fields. For instance, if the variable S is introduced in the expression (IN S SHIP), then any subsequent reference such as (S NAM) would be interpreted as indicating the NAM field in the SHIP relation.

In fact SODA is more flexible than this. The relations specified in the initial SODA query are interpreted as virtual relations that may refer to fields stored on several actual data base relations. In SODA, there is one virtual relation for each type of object that we want to talk about (i.e. allow as the value of a variable), and for each type of object there is a schema that indicates the semantically meaningful ways of linking fields in different relations. (In the current implementation, the same schema is used for all virtual relations. This is an artifact of the particular data base being used, and would not be possible in general.) For instance, the schema for the virtual SHIP relation would specify that when talking about a ship, if we mention a field in the data base SHIP relation (e.g. NAM) and another field in the SHIPCLASS relation (e.g. DRAFT), then the way to link them is to join the two relations via the CLASS field.

SODA uses this information to transform the references to virtual relations in the initial query into references to actual data base relations. It does this by scanning the query for all the fields mentioned in connection with each variable introduced by an IN expression. It then uses the schema for the virtual relation that the variable ranges over to find the smallest set of data base relations that include all the fields and to specify the links between these relations. SODA then replaces the original IN expression that mentions the virtual relation with a series of IN expressions that mention the selected data base relations and specify the joins between them. The references to the fields in the virtual relation are replaced by the corresponding references to fields in the data base relations. For example, if we wanted to retrieve the name and draft of all the ships in the data base, the initial query would be:

```
((IN S SHIP)
 (? (S NAM))
 (? (S DRAFT)))
```

Since the NAM field occurs only in the data base SHIP relation and since the DRAFT field occurs only in the SHIPCLASS relation, both relations must be accessed. SODA therefore transforms this query into:

```
((IN S SHIP)
 (IN C SHIPCLASS ((C CLASS) EQ (S CLASS)))
 (? (S NAM))
 (? (C DRAFT)))
```

In this expanded query, SHIP and SHIPCLASS are interpreted as being actual data base relations, whereas in the initial query SHIP was interpreted as a virtual relation.

In expanding references to virtual relations, SODA must choose which relation to use to retrieve a particular field if that field is available from more than one relation. In our sample data base the type and length of a ship can be retrieved either from the SHIP relation or the SHIPCLASS relation. To solve this problem, SODA uses heuristic techniques developed for IDA to attempt to minimize the number of relations accessed. For more information on how this is done, see [6].

Another problem for SODA is to choose the order in which to access the relations mentioned in a query. We could interpret a SODA query as specifying a particular procedure by making a fixed processing strategy (such as strictly sequential processing) part of the definition of the language. The user would then be responsible for determining the order in which relations are accessed by choosing the order in which they are mentioned. Since SODA's main purpose is to be the target language for a natural-language processor that makes no attempt to order the queries it generates for efficiency, we use a few simple heuristics to reorder the initial query. First, restrictions are applied as soon as all the relations that they mention have been accessed, since this cuts down the amount of data that must be processed in the rest of the query. Next, any maximization, minimization, or counting expressions that can be processed are taken, since these expand the amount of data only slightly. After these expressions, IN expressions which can immediately be restricted are preferred over IN expressions which cannot. These heuristics are all intended to help choose the most restricted parts of the query first in hopes of minimizing the size of intermediate results.

D. PROBLEMS IN DISTRIBUTED PROCESSING OF COMPLEX QUERIES

If all the relations mentioned in an expanded reordered SODA query are stored at one data base site, then all that remains to be done is to translate the query into the query language of the DBMS at that site and execute the query. If, however, the data is distributed over two or more sites, some strategy must be devised for combining information from several locations.

What type of strategy is used will depend on assumptions about the relative efficiency of various operations. Since our data base is distributed over several sites on the ARPANET, a relatively low-speed communications channel, we have assumed that query processing will be most efficient if as much work as possible is done at a single site, and the amount of data transmitted between sites is kept to a minimum. (If transferring data between sites were fast compared to query processing at one site, the best strategy might be to spread the data over as many sites as possible to take advantage of concurrent processing.)

Given these assumptions, there seem to be two simple approaches that might be followed. One approach is to move all the relevant data to a single data base site and execute the query in one access to that site. We will call this the centralized approach. An efficient implementation of this approach would involve doing any local processing that would reduce the amount of data transmitted, such as taking projections, restrictions, or joins of relations, before sending the data to the primary site.

The other approach, which we will call the incremental approach, is to decompose the query into a series of simpler queries, each of which can be executed at a single data base site. Then each query is executed in turn at the corresponding site, and the results are transferred to the site where the next query is to be processed and combined with the information there. An efficient implementation of this approach would attempt to order the execution of the queries so as to minimize the total amount of data transmitted.

These two approaches do not exhaust the range of possibilities, of course. In fact, from a slightly more general point of view, they can be seen to be the two extremes of a spectrum. Since the final answer to a query will be generated at only one of the data base sites, we can view the problem of distributed query processing as how to organize the data base sites as a "data-flow tree," with information being transmitted up the branches towards the root, where the final answer is generated. From this point of view, the centralized approach limits its attention to the maximally branching, minimally deep trees, and the incremental approach limits its attention to the minimally branching, maximally deep trees. The most efficient organization may well be found in one of the intermediate possibilities, but we only consider these two approaches, as they are the easiest to implement.

If used intelligently, the incremental approach is often much more efficient than the centralized approach. The reason for this is not hard to see. Using the incremental approach, if we begin processing with a partial query that is highly restricted, that restriction will be "inherited" by all the subsequent partial queries that are processed, since at every stage we combine everything we have done so far before transferring the data to the next site. In the centralized approach, however, any processing that is done before transferring data is done independently of processing at other sites, so there is no way to take advantage of restrictions that may have been computed elsewhere.

For instance, in our sample data base, suppose that the PORT relation and the SHIPCLASS relation are stored at site 1 and the SHIP relation is stored at site 2. If we wanted to know the name and draft of all the ships currently in American ports, we would have to access all three relations and, therefore, both data base sites. The expanded SODA query for this request would be:

```
((IN P PORT ((P PNAT) EQ 'US'))
 (IN S SHIP ((S POS) EQ (P PPOS)))
 (IN C SHIPCLASS ((C CLASS) EQ (S CLASS)))
 (? (S NAM))
 (? (C DRAFT)))
```

The most natural way of processing this query using the incremental approach would be to retrieve the locations of American ports from site 1, transfer this information to site 2 to find the names and classes of the ships at these locations, and then transfer that information back to site 1 to find the drafts of the ships and return the answers. Presumably, the amount of data transferred during this process would be significantly smaller than the amount that would be transferred either by moving the required fields of the SHIP relation to site 1 or by moving the required fields of the SHIPCLASS and PORT relations to site 2, as would be required by the centralized approach.

Examples such as this suggest that the incremental approach is to be generally preferred to the centralized approach. However, in complex, quantified queries, which are the major concern of our work on SODA, the incremental approach may be impossible to apply. This fact seems not to have been generally recognized in the literature on distributed query processing (e.g., [17]), where joining is typically the only method considered for combining data from two or more relations.

The problem of distributed query processing is considerably simplified by considering only joins for two reasons: First, joins specified over more than two relations can always be decomposed into a series of binary joins. Thus, if some of the relations to be joined are at one site and some are at another site, the relations at the same site can be processed first, and the intermediate results can be combined later. In the previous example, the query specified a join over the PORT, SHIP, and SHIPCLASS relations. In processing, this was decomposed into a join over the PORT and SHIP relations, and a join between the result of this operation and the SHIPCLASS relation.

The second simplification that joining permits is that, since the join operation is associative, it doesn't matter logically how the decomposition is done. Therefore, the decomposition can be chosen to suit the way the data is distributed. In our example we first joined the PORT relation to the SHIP relation and then joined the result of

that operation to the SHIPCLASS relation. If the distribution of the data or the expected sizes of intermediate results had been different, however, it might have been more efficient to join the SHIP and SHIPCLASS relations first, and then add in the PORT relation.

In complex, quantified queries, on the other hand, the possible ways of decomposing queries are much more restricted. It is often impossible to break up queries to match the distribution of the relations, and in some cases, queries over several relations cannot be decomposed at all.

This point can be illustrated by changing our previous example slightly. Consider the same query, finding the name and draft of all ships in American ports, but with the PORT and SHIP relations at site 1 and the SHIPCLASS relation at site 2. In this case, it is probably most efficient to find the ships that are in American ports by joining the PORT relation and SHIP relation at site 1 and transfer the result to site 2 to join it with the SHIPCLASS relation to form the final answer.

Now let us alter the query so that it includes a universal quantifier, but still refers to the same relations in the same order, e.g., "Which American ports contain only ships which have draft greater than 50 feet?":

```
((IN P PORT ((P PNAT) EQ 'US'))
 (ALL (IN S SHIP ((S POS) EQ (P PPOS)))
      (IN C SHIPCLASS ((C CLASS) EQ (S CLASS)))
      ((C DRAFT) GT 50))
 (? (P PDEP)))
```

The logical structure of this SODA query can be indicated by paraphrasing it back into English as follows:

```
For each tuple P in the PORT relation
         with (P PNAT) equal to 'US'
     such that, for all tuples S in the SHIP relation
                   with (S POS) equal to (P PPOS)
          and all tuples C in the SHIPCLASS relation
                   with (C CLASS) equal to (S CLASS),
          (C DRAFT) is greater than 50,
return (P PDEP).
```

Since the PORT and SHIP relations are both stored at site 1, and since there is a link specified between them, ((S POS) EQ (P PPOS)), we would like to decompose the query by first operating on these two relations and transferring the intermediate result to site 2 for processing with the SHIPCLASS relation. Unfortunately, the universal quantifier ALL does not permit any such decomposition. There is no way to combine the data referred to outside of the ALL expression with only part of the data referred to inside. We would need a distribution principle analogous to A*(B+C) = (A*B)+(A*C) to distribute the universal quantifier over the relations mentioned inside of the ALL expression, but no such principle exists.

Any query decomposition that is performed must respect the scope of quantifiers. We can independently process a portion of a query that lies entirely within the scope of a quantifier or entirely outside the scope of a quantifier, but we cannot independently process a portion of a query that splits the scope of a quantifier.

By nesting quantifiers more deeply, it is possible to construct queries over several relations that cannot be decomposed at all. Suppose we wanted to know the ship classes for which every American port contains some ship in that ship class. This could be represented in SODA as:

```
((IN C SHIPCLASS)
 (ALL  (IN P PORT ((P PNAT) EQ 'US'))
       (SOME (IN S SHIP ((S POS) EQ (P PPOS))
                        ((S CLASS) EQ (C CLASS)))))
 (? (C CLASS)))
```

The English paraphrase of this SODA query would be:

```
For each tuple C in the SHIPCLASS relation
    such that, for all tuples P in the PORT relation
            with (P PNAT) equal to 'US',
        there is some tuple S in the SHIP relation
                    with (S POS) equal to (P PPOS)
                    and (S CLASS) equal to (C CLASS),
return (C CLASS).
```

This query cannot be decomposed. We cannot combine the data from the SHIPCLASS relation with the data from either the SHIP relation alone or the PORT relation alone, because this would cut across the scope of a quantifier. For the same reason, the SHIP relation and PORT relation cannot be combined without processing the whole SOME restriction. But this cannot be done independently of the SHIPCLASS relation, because the SOME restriction refers to the data from the SHIPCLASS relation via the term (C CLASS). Answering this query, therefore, requires simultaneous access to three relations.

Even though in queries such as these we cannot always combine relations locally before transferring data, we still can use projections and restrictions to cut down the amount of data that must be transferred. It turns out that in some cases we can add logically redundant restrictions that have this effect, although this is not done in the current implementation. Recall the previous query, "Which American ports contain only ships which have draft greater than 50 feet?" We could add a redundant restriction without changing the answer to the query and get "Which American ports contain only ships which are in some American port and have draft greater than 50 feet?" The SODA representation of the modified query would be:

```
((IN P PORT ((P PNAT) EQ 'US'))
 (ALL (IN S SHIP ((S POS) EQ (P PPOS))
                 (SOME (IN P1 PORT)
                       ((P1 PNAT) EQ 'US')
                       ((P1 PPOS) EQ (S POS))))
      (IN C SHIPCLASS ((C CLASS) EQ (S CLASS)))
      ((C DRAFT) GT 50))
 (? (P PDEP)))
```

We still cannot independently combine the data generated by the expression (IN P PORT...) with the data generated by (IN S SHIP...), but if the PORT and SHIP relations are at the same site, we can compute the restrictions on (IN S SHIP...), including the restriction to ships in American ports using (IN P1 PORT...). So, although this restriction is logically unnecessary, it permits us to transfer much less data than would be required without it.

E. DISTRIBUTED QUERY PROCESSING IN SODA

As the previous subsection indicated, complex queries do not always permit decomposition into sequences of simpler queries that match the distribution pattern of the data base. As a result, we have chosen to base the initial implementation of SODA on the centralized approach to distributed query processing. In doing so, we have traded the efficiency of the incremental approach in handling simpler queries for the generality of an approach that handles the more complicated queries which are our primary interest. A more sophisticated implementation could employ a mixed strategy, using the incremental approach when it is applicable and falling back on the centralized approach when it is not. Also, we have not implemented the type of query transformation discussed in the preceding subsection, since further research is needed to determine what the scope and limits of such techniques are.

In processing a query, SODA must first decide which data base site to use as the primary site for executing the query. A set of reasonable candidates is selected by starting with a list of all the sites that contain at least one of the relations mentioned in the query. Then redundant sites are eliminated until no site remaining in the list has the property that some other site in the list contains all the relations mentioned in the query that are contained by that site. Processing of the query is then simulated, trying each of the remaining sites as the primary site. The choice of primary site that appears to result in the least amount of data being transferred is selected to be the primary site for actually processing the query. This measure is currently crudely estimated by choosing the site that results in the fewest unrestricted queries requesting data from a secondary site. If this leaves more than one possibility, then one of those that results in the fewest restricted queries is chosen. A query is considered to be restricted if there is a restriction on any of the relations mentioned in the query.

Once the primary data base site has been chosen, SODA reformulates the query for execution at that site. The query is examined, one

expression at a time. Expressions which refer only to data that is already at the primary site are left unchanged. If an expression refers to data that is not stored at the primary site, then this data is transferred to a temporary relation at the primary site, and the query is reformulated to refer to this relation. To take an example from Section IV.D, recall that the SODA query for retrieving the name and draft of all ships in American ports is:

```
((IN P PORT ((P PNAT) EQ 'US'))
 (IN S SHIP ((S POS) EQ (P PPOS)))
 (IN C SHIPCLASS ((C CLASS) EQ (S CLASS)))
 (? (S NAM))
 (? (C DRAFT)))
```

If the PORT relation is stored at site 1 and the SHIP and SHIPCLASS relations are stored at site 2, site 2 will be chosen as the primary site for execution of the query, as this results in only a single restricted query being executed at a secondary site. Since the PORT relation is not stored at site 2, SODA first obtains the information needed from the PORT relation by dispatching the query:

```
((IN P PORT ((P PNAT) EQ 'US'))
 (? (P PPOS)))
```

to site 1 and stores the result in a temporary relation at site 2, say in field FIELD1 of relation TEMP1. The transferred data is constrained as much as possible by applying the restriction ((P PNAT) EQ 'US') before the transfer, and only the fields required by the rest of the query are moved, in this case, just the PPOS field. The main query is now reformulated as:

```
((IN T TEMP1)
 (IN S SHIP ((S POS) EQ (T FIELD1)))
 (IN C SHIPCLASS ((C CLASS) EQ (S CLASS)))
 (? (S NAM))
 (? (C DRAFT)))
```

Since the query now refers only to relations stored at site 2, it can be executed in a single access to that site.

The process just described is complicated somewhat by a set of issues involving redundantly stored data and error recovery. One of the principal advantages of a distributed data base is that the system can be made more reliable by storing data redundantly at several data base sites. If this is done, then the system can tolerate failure of one or more data base sites and still be able to answer all the queries covered by the data base (although not always with the most recent information).

In SODA, therefore, we take into account the possibility that a given relation may be stored at more than one data base site, and we use this fact to try to recover from data base failures. Because of our centralized approach to query processing, we distinguish between failure of the primary site and failure of one of the secondary sites. Since all intermediate results are stored at the primary site, a failure there requires complete replanning and re-execution of the query. If a secondary site fails, however, SODA backs up only as far as the beginning of the portion of the query involved in the current access to that site and begins replanning from that point. This preserves any intermediate results that have actually been extracted from secondary sites and thus avoids unnecessary recomputation.

A more difficult question for SODA is at what site to access a particular relation, when more than one possibility is available. Solutions to this problem are also constrained by our use of a centralized approach for query processing, since, in general, we want to access relations at the primary site if possible. There are exceptions to this rule, however. In particular, if we must transfer information from a secondary site, it may be more efficient to go ahead and combine that information with data from another relation at the secondary site, even though the other relation may also be stored at the primary site.

SODA uses a number of simple heuristics to decide whether to access a relation at the primary site or a secondary site. Roughly, SODA will prefer a secondary site if the relation is joined to another relation which must be accessed at the secondary site, and if performing the join appears likely to cut down the amount of data retrieved from that

relation. A join is assumed to cut down the amount of data retrieved if it puts more restrictions on the data.

For instance, if we wanted to know about American ships with a draft of more than 50 feet, we would have to access the NAT field in the SHIP relation and the DRAFT field in the SHIPCLASS relation. Suppose the SHIP relation is stored only at a secondary site and the SHIPCLASS relation is stored both at that site and the primary site. In this case, we would access the SHIPCLASS relation at the secondary site because it would further restrict the set of ships for which data must be transferred to the primary site. If, on the other hand, we simply wanted to retrieve the drafts of American ships, we would access the SHIPCLASS relation at the primary site, since this would not further restrict the data being transferred.

These heuristics are rather crude, since they do not take into account the relative sizes of relations, how constraining a particular restriction is, or the functionality of joins between relations (e.g., many-to-one, one-to-many). There is clearly a trade-off, however, between time spent in access planning and time spent in query execution, and it is not clear how much more effort could be put into access planning that would justify itself in more efficient query execution.

Compared to the SDD1 distributed DBMS [17] [18] [19], the techniques used in SODA have both advantages and disadvantages. SDD1 takes what is essentially a centralized approach to query processing, but not as completely as SODA. The main difference is that for purposes of assembling all the relevant data at a single site, SDD1 treats the query as if it contained only joins and restrictions. If a query specifies a more complex way of combining relations than joining, SDD1 will find a join that "covers" that portion of the query, in the sense that the data it retreives includes as a subset all data required to answer the query. However, it does not perform a precise logical analysis, as SODA does, to retrieve exactly the required data. Because SDD1 takes this simpler view, it is able to use more sophisticated heuristics for combining partial results from several secondary sites

before transferring them to the primary site. However, since the partial results are only approximate, the entire query must be re-executed at the primary site.

One clear advantage that SDD1 has over SODA is that SDD1 maintains statistical information about the size of relations and the distribution of values of fields. This enables SDD1 to predict more accurately than SODA the size of intermediate results, and hence do a better job of query optimization. It should be noted, however, that SODA is designed to permit use of such information without any changes to the basic structure of the system.

One final difference between SDD1 and SODA is that, although SDD1 permits arbitrarily redundant data bases, a particular query is answered only with respect to a single nonredundant mapping of the data base. Because SODA can decide at processing time where to access a redundantly stored relation, it is possible to answer some queries more efficiently and to recover from the failure of a secondary data base site without completely reprocessing a query.

F. LIMITATIONS AND POSSIBLE EXTENSIONS OF SODA

Like any real system which addresses a complex problem, SODA offers only partial solutions to the issues it raises. There are several areas where significant improvements or extensions could be made. One of these areas is the expressive power of the query language. Although SODA is a richer language than IDA and many other data base query languages, there are still useful queries that it cannot express.

One of the constructs that SODA lacks is some kind of conditional expression. For example, it might be desirable not to store the current position of ships that belong to task forces individually in the SHIP relation, but rather to have this information derived by looking up the location of the task force to which a ship belongs in a TASKFORCE relation. This would make it possible to update the location of the entire task force at once, rather than ship by ship. If we do this, however, retrieving the position of a ship becomes a conditional

procedure, depending on whether the ship belongs to a task force. To retrieve the position of a particular ship, such as the Fox, we would have to be able to express in SODA the following query, which we currently cannot handle:

```
For the tuple S in the SHIP relation
        with (S NAM) equal to 'FOX',
    if (S TFNAM) has the undefined value
        then return (S POS),
    otherwise, for the tuple T in the TASKFORCE relation
            with (T TFNAM) equal to (S TFNAM),
        return (T TFPOS).
```

Another class of queries that cannot be expressed in SODA is queries that involve following chains of indefinite length through the data base. For instance, in the personnel data bases that are commonly used to illustrate concepts of data base access, a classic problem is to answer the query, "Which employees earn more than their managers?" Many of the simpler query languages that have been proposed, including IDA, cannot represent such a question, although for SODA this would be no problem.

However, if we want to define the relationship "superior of" to be the transitive closure of "manager of" (i.e., the manager of the manager, etc.), we are in trouble. There does not seem to be any non-procedural query language, including SODA, that could express queries such as, "Which employees earn more than all/some of their superiors?" The problem is that expressing this type of query asks a question about all chains through the data base of a certain kind, whereas existing query languages only allow asking about all tuples of a certain kind.

Another general area where SODA could be improved is query optimization and access planning. The heuristics used to pick the order in which relations are accessed are quite crude, taking into account only which references to relations are restricted. As we pointed out in discussing the heuristics for distributed query processing, it would also be useful to consider the relative sizes of relations, how constraining a restriction is, and the functionality of joins between relations.

It will never be possible to guarantee that a query will be processed in the optimum way, however. First of all, to do so would require knowing the size of all of the possible intermediate results that might be generated in processing the query, and in general the only way to get this information is to execute the query. Second, even if we had good enough estimates for all of the relevant factors, choosing the most efficient way to process a query would still be a combinatorial search, which might take longer to perform than executing the query with the few simple heuristics we currently have. So any technique for query optimization must be empirically tested to see whether the savings it produces are worth the cost of applying it.

Finally, some of the improvements planned for SODA concern pragmatic problems in dealing with interactive users. One of these problems is that if there is no information in the data base satisfying a complex query, the system simply returns a null result with no further explanation. Often it would be much more helpful to the user if the system would provide some indication of why it failed to find an answer. For instance, if we ask the system to compute the distance between the Fox and the Kennedy and get no answer, it might indicate that the Fox is not listed in the data base, or the position of the Fox is not given in the data base, or the Kennedy is not listed in the data base, or the position of the Kennedy is not given in the data base, or any combination of the above. We are currently investigating how this information might be obtained from the data base and supplied to the user in a form he can understand. For more discussion of problems of this kind, see Kaplan [16].

The other pragmatic problem we are looking at is how to save and make use of previously retrieved information to avoid recomputing it. For example, if we ask the LADDER system in English, "Which American ships are in the Atlantic?" followed by the question "What are their fuel states?" the pronoun "their" will be correctly resolved to the phrase "American ships in the Atlantic" by the natural-language front end, but this set of ships will be recomputed by the data base access

component. Although the natural-language processor realizes that the two queries are related, SODA does not. We are examining various issues that arise in dealing with this problem, including what information to save, how long to save it, and where it should be stored.

V ACCESSING A CODASYL DATA BASE SYSTEM: DBMS-20

A. INTRODUCTION

One of the significant improvements made to the LADDER system during the period covered by this report was to extend its capabilities to include querying more than one DBMS. The system can now access DBMS-20, a DBGT data base system [9] provided by Digital Equipment Corporation (DEC) for the DecSystem-20 operating system, which supports a subset of the CODASYL data model [14]. This section of the report discusses the problems encountered in developing a translator to produce queries for DBMS-20 and in interfacing the DBMS-20 system to LADDER.

The data access component of LADDER, called SODA, consists of two principal parts. These are (1) a planning module, which decides how an input query is to be evaluated given the distribution of required data among various relations stored on multiple data base management systems (DBMSs) at multiple sites on the ARPANET, and (2) a set of translation modules, each of which translates SODA queries into the query language supported by a particular DBMS. Initially, the only SODA translator produced queries in Datalanguage, the query language supported by the Datacomputer DBMS, which, like SODA, is fundamentally a relational system.*

DBMS-20, on the other hand, is a "network" DBMS. Simply stated, a network data base allows the value of a field to be an explicit pointer to another record. (This is done by use of the SET feature in DBMSs based on the CODASYL specification.) For example, to link a SHIP record

---

* In using the Datacomputer, SODA assumes that it is a relational DBMS. In general, however, the Datacomputer can be viewed as a hierarchical DBMS: it allows the use of "repeating groups" (i.e., groups of fields which may be repeated an indefinite number of times inside a record), a feature is explicitely forbidden in the relational model of data. In the Blue File, this feature of the Datacomputer is not in use, so the Blue File is for all practical purposes a relational data base.

to the corresponding SHIPCLASS record in the Blue File on the Datacomputer, one must read the CLASS field in the SHIP record, and find the record in the SHIPCLASS file whose CLASS field has the same value. On the other hand, in a CODASYL version of the Blue File, a SET could be implemented in such a way that a pointer would directly link each SHIP record to the corresponding SHIPCLASS record. Obviously, the query program which is generated for the CODASYL version of a Blue File would be very different from one generated for the relational Blue File. The ability of SODA to access a DBTG network data base opens up new possibilities for speeding up processing of some queries, through use of the SET feature.

B. COMPILING SODA QUERIES

DBMS-20 accepts query programs written in Interactive Query Language (IQL), a COBOL-based language, which it compiles and executes. Our development effort consisted of making incremental changes to the IQL language to support needed features not originally supplied, modifying the compiler to support the language changes, providing an interface with the rest of the LADDER system, and implementing a translator to compile SODA expressions into IQL programs. In the latter effort, we closely mimicked the structure of the translator which generates Datalanguage for the Datacomputer DBMS.

A SODA query is a list that may contain a number of different constructs. For each construct there is a prototypical code structure in IQL generated, and one or two modules of code which produce that structure. The following subsections review these modules.

1. Relation-Searching Constructs

A fundamental operation in answering queries is stepping through some or all of the tuples of a relation to perform some other operation on them. In the SODA expressions seen by the SODA translators, there are six constructs which involve this operation: IN, MAX1, MIN1, COUNT, SOME and NONE (MAX, MIN, and ALL expressions are

reformulated in terms of the others). The last two of these return a value of True or False, and will be discussed in a later section on restrictions.

IN is the most straightforward of the constructs. An IN expression specifies a relation whose tuples are to be examined and a tuple variable to associate with this relation. Restrictions may also be specified in the IN expression to indicate that processing is only to be performed on certain tuples in the relation. All SODA constructs following the IN construct are considered to be within the scope of the loop it implicitly defines. A COUNT expression defines a variable whose value is to be the number of tuples that satisfy the subquery in the COUNT expression. This variable can be referenced later by other parts of the program, or returned as an answer to LADDER. A MAX1 expression contains a specification of a quantity to be maximized and a subquery defining the tuples over which the maximization is to be performed. The effect of the MAX1 expression is to find a set of values for the variables in the scope of the expression that produces the highest value for the quantity being maximized. As with COUNT, these variables are available for reference by subsequent parts of the program. MIN1 does the same thing for minimization.

We currently support two ways of searching the tuples of a relation on DBMS-20 data bases: sequential search and calc-key (hashed) search. We also plan to implement a search which makes use of DBTG SETs. In all cases, the query as expressed in SODA looks the same--the determination of the access method to be used is performed by the SODA/IQL translator.

Example 1 shows the SODA formulation of the query "Who commands the Grayling, Baton Rouge, Pogy, Sturgeon, and Los Angeles?", and the IQL code generated from it. The program's structure is essentially a loop which steps sequentially through each tuple of the UNITS relation, and in the cases in which a unit name is matched, prints out the unit-commander's rank and name. (Note the use of the generated variable XOR13; restrictions will be discussed in more detail below).

Example 1:

"Who commands Grayling, Baton Rouge, Pogy, Sturgeon, and Los Angeles?"

```
((IN S SHIP (OR ((S NAM) EQ 'GRAYLING')
                ((S NAM) EQ 'BATON% ROUGE')
                ((S NAM) EQ 'POGY')
                ((S NAM) EQ 'STURGEON')
                ((S NAM) EQ 'LOS% ANGELES')))
 (? (S RANK))
 (? (S CONAM)))
```

```
   OPEN ACCAT $
   FIND FIRST UNITS RECORD OF BLUEAREA AREA $
10 IF ERROR-STATUS = 307 GO TO 11 $
   COMPUTE XOR13 = 1 $
   IF UNITS-ANAME EQ 'GRAYLING' GO TO 14 $
   IF UNITS-ANAME EQ 'BATON ROUGE' GO TO 14 $
   IF UNITS-ANAME EQ 'POGY' GO TO 14 $
   IF UNITS-ANAME EQ 'STURGEON' GO TO 14 $
   IF UNITS-ANAME EQ 'LOS ANGELES' GO TO 14 $
   COMPUTE XOR13 = 0 $
14 IF XOR13 = 0 GO TO 12 $
   PRINT UNITS-RANK UNITS-CONAM $
12 FIND NEXT UNITS RECORD OF BLUEAREA AREA $
   GO TO 10 $
11 * $
  GO TO XT $
**END**
```

Example 2 is a slightly more elaborate query, "Where is the Constellation?", which requires searching two relations: SHIP (a relation which contains ship names), and then TRACKHIST (which contains the most recent position of each ship). The structure of the program is two nested loops, the outer one performs a sequential search through SHIP; the inner one (beginning with the PUSHLP just after statement 14 and ending with the POPLP just before statement 12) performs a hash-lookup in TRACKHIST. The SODA/IQL translator will generate a hashed search if the corresponding SODA IN-construct contains a single restriction specifying a particular value of the attribute on which the relation is hashed (the CALC-KEY field); otherwise it generates a sequential search. Since there may be more than one tuple having a given value for the CALC-KEY field, it is necessary to generate a loop, using the FIND NEXT DUPLICATE statement.

In this query, the two relations being searched are distinct; but in general, they could be the same relation, which would lead to the following problem with the DBMS-20 implementation of CODASYL: only one "current position" pointer and one data buffer is maintained for each relation, so that the position and current values of the fields associated with the outer loop would be destroyed by performing the inner one. We have therefore implemented the PUSHLP statement, which saves the current value of the position pointer for a relation, and the POPLP statement, which restores the position pointer and rereads the current tuple of the relation into the corresponding data buffer.

Example 2:

"Where is the Constellation?"

```
((IN S SHIP ((S NAM) EQ 'CONSTELLATION'))
 (IN T TRACKHIST ((T UICVCN) EQ (S UICVCN)))
 (? (T PTP))
 (? (T PTD)))

   OPEN ACCAT $
   FIND FIRST SHIP RECORD OF BLUEAREA AREA $
10 IF ERROR-STATUS = 307 GO TO 11 $
   COMPUTE XAND13 = 0 $
   IF SHIP-NAM NE 'CONSTELLATION' GO TO 14 $
   COMPUTE XAND13 = 1 $
14 IF XAND13 = 0 GO TO 12 $
   PUSHLP $
   SET TRACKHIST-UICVCN TO SHIP-UICVCN $
   FIND TRACKHIST RECORD $
15 IF ERROR-STATUS = 326 GO TO 16 $
   PRINT TRACKHIST-PTP TRACKHIST-PTD $
17 FIND DUPLICATE TRACKHIST RECORD $
   GO TO 15 $
16 * $
   POPLP $
12 FIND NEXT SHIP RECORD OF BLUEAREA AREA $
   GO TO 10 $
11 * $
  GO TO XT $
**END**
```

2. Restrictions

Restrictions may be simple comparsions using "equal," "not equal," "greater than," "less than," etc., or they may be complex logical combinations of these built out of AND, OR, SOME, and NONE. AND and OR are straightforward; SOME and NONE are looping constructs like IN, COUNT, MAX1, and MIN1, that operate on a set of tuples defined by the body of the SOME or NONE expression. Simply stated, a SOME expression returns True if the corresponding set of tuples is nonempty and False, otherwise; a NONE expression does the reverse.

In all cases, the value of a restriction is placed in a temporary variable whose value is then interrogated by the code corresponding to the SODA context in which the restriction occurred. This is slightly awkward but necessary, because the restriction-code generator doesn't know why the test is being made--it could be simply to determine whether a tuple is eligible for further processing, or it could be to decide if the answer to the entire query is "YES" or "NO." Examples 3 and 4 illustrate these cases, respectively.

Example 3:

"What ship classes contain some American ships?"

```
((IN C SHIPCLASCHAR)
 (SOME (IN D SHIPCLASDIR ((D SHIPCLAS) EQ (C SHIPCLAS)))
       (IN S SHIP ((S UICVCN) EQ (D UICVCN))
                  ((S NAT) EQ 'US'))
 (? (C SHIPCLAS)))

   OPEN ACCAT $
   FIND FIRST SHIPCLASCHAR RECORD OF BLUEAREA AREA $
10 IF ERROR-STATUS = 307 GO TO 11 $
   COMPUTE XSOME14 = 0 $
   PUSHLP $
   FIND FIRST SHIPCLASDIR RECORD OF BLUEAREA AREA $
15 IF ERROR-STATUS = 307 GO TO 13 $
   COMPUTE XAND17 = 0 $
   IF SHIPCLASDIR-SHIPCLAS NE SHIPCLASCHAR-SHIPCLAS GO TO 18 $
   COMPUTE XAND17 = 1 $
18 IF XAND17 = 0 GO TO 16 $
   PUSHLP $
   SET SHIP-UICVCN TO SHIPCLASDIR-UICVCN $
   FIND SHIP RECORD $
19 IF ERROR-STATUS = 326 GO TO 20 $
```

```
      COMPUTE XAND22 = 0 $
      IF SHIP-NAT NE 'US' GO TO 23 $
      COMPUTE XAND22 = 1 $
   23 IF XAND22 = 0 GO TO 21 $
      COMPUTE XSOME14 = 1 $
   21 FIND DUPLICATE SHIP RECORD $
      GO TO 19 $
   20 * $
      POPLP $
   16 FIND NEXT SHIPCLASDIR RECORD OF BLUEAREA AREA $
      GO TO 15 $
   13 * $
      POPLP $
      IF XSOME14 = 0 GO TO 12 $
      PRINT SHIPCLASCHAR-SHIPCLAS $
   12 FIND NEXT SHIPCLASCHAR RECORD OF BLUEAREA AREA $
      GO TO 10 $
   11 * $
     GO TO XT $
   **END**
```

Example 4:

"Is there a ship named Sturgeon?"

```
((SOME (IN S SHIP ((S NAM) EQ 'STURGEON'))))

     OPEN ACCAT $
     COMPUTE XSOME11 = 0 $
     FIND FIRST SHIP RECORD OF BLUEAREA AREA $
  12 IF ERROR-STATUS = 307 GO TO 10 $
     COMPUTE XAND14 = 0 $
     IF SHIP-NAM NE 'STURGEON' GO TO 15 $
     COMPUTE XAND14 = 1 $
  15 IF XAND14 = 0 GO TO 13 $
     COMPUTE XSOME11 = 1 $
  13 FIND NEXT SHIP RECORD OF BLUEAREA AREA $
     GO TO 12 $
  10 * $
     PRINT XSOME11 $
    GO TO XT $
  **END**
```

3. Terms

Terms are the SODA expressions that actually refer to data and may be computed, compared, and tested. Their values may be either integer numbers or character strings. The processing of a constant or a

count variables is straightforward; the handling of a reference to a field of a tuple requires a little explanation. The simplest case is a tuple-variable/field pair that refers to the current value of the field in the current tuple of the relation associated with the variable; that value can be indicated in IQL simply by joining the relation name to the field name with a hyphen, e.g., SHIP-NAM. If there are nested loops, however, and a tuple-variable/field pair of one loop occurs inside an inferior loop, the "current" tuple of the associated relation may have been reset by the inner loop. Because of this possibility, whenever a new loop is started, the values of all the fields of the current tuple of the containing loop that are referred to inside the new loop are stored in specially created temporary variables; references are then made to those variables rather than to the fields themselves.

A similar use of temporary variables occurs in the evaluation of a MAX1 or MIN1 expression. As described in Appendix A, a MAX1/MIN1 expression picks out one value for each of the tuple variables in the scope of the MAX1/MIN1. This set of values will be one that produces the maximum or minimum value for the expression being maximized or minimized over. The simplest way to find the appropriate set of values for these variables is to loop through the relations that contain the candidate tuples, comparing each one to a temporary variable which represents the maximum or minimum value seen so far, and updating that variable and other temporary variables for fields of the associated tuples needed later in the query. This is illustrated by example 5, which finds the ships with the maximum length. The IQL program for this query scans once through the ship relation to find the maximum length for any ship, letting XX10 be any indicator of whether any ships were found (this would be more meaningful if we were looking for, say, Dutch aircraft carriers, rather than just ships) and letting XX11 be the current maximum length. The ship relation is then scanned a second time to list all the ships having that length.

Example 5:

"What are the longest ships?"

```
((MAX1 (S1 LGHN)
       (IN S1 SHIP))
 (IN S2 SHIP ((S2 LGHN) EQ (S1 LGHN)))
 (? (S2 NAM)))

   OPEN ACCAT $
   COMPUTE XX11 = 0 $
   COMPUTE XY10 = 0 $
   FIND FIRST SHIPCLASCHAR RECORD OF BLUEAREA AREA $
13 IF ERROR-STATUS = 307 GO TO 14 $
   COMPUTE XZ12 = SHIPCLASCHAR-LGHN $
   IF XZ12 GT 9999 OR XX11 GE XZ12 GO TO 15 $
   COMPUTE XX11 = XZ12 $
   COMPUTE XY10 = 1 $
   COMPUTE XNUM21 = SHIPCLASCHAR-LGHN $
15 FIND NEXT SHIPCLASCHAR RECORD OF BLUEAREA AREA $
   GO TO 13 $
14 * $
   IF XY10 = 0 GO TO XT $
   FIND FIRST SHIP RECORD OF BLUEAREA AREA $
16 IF ERROR-STATUS = 307 GO TO 17 $
   COMPUTE XAND19 = 0 $
   IF SHIP-LGHN NE XNUM21 GO TO 20 $
   COMPUTE XAND19 = 1 $
20 IF XAND19 = 0 GO TO 18 $
   PRINT SHIP-NAM $
18 FIND NEXT SHIP RECORD OF BLUEAREA AREA $
   GO TO 16 $
17 * $
  GO TO XT $
**END**
```

4. Special Functions

GCDIST, RLDIST, COURSE and BEARING are four special functions for computing distances and relative positions on spherical surfaces, which we have added to DBMS-20 to illustrate the integration of complex calculation with data base retrieval. These functions could have been implemented more easily at the LISP level, but this would have made it much less efficient to answer questions in which the calculation is not the top level operation, such as "What is the closest ship to Luanda?" To handle these functions, we have extended the IQL system to recognize special commands which cause the appropriate assembly code to be

executed. The input arguments to these functions, (two positions, each expressed as a quadruple, e.g. (75 'N' 23 'E')), and their answer values are communicated via nine temporary variables, in a call-by-reference manner. Example 6 shows the use of GCDIST to answer the query "How far is the Fox from the Kennedy?"

Example 6:

"How far is the Fox from the Kennedy?"

```
((IN S1 SHIP ((S1 NAM) EQ 'FOX'))
 (IN T1 TRACKHIST ((T1 UICVCN) EQ (S1 UICVCN)))
 (IN S2 SHIP ((S2 NAM) EQ 'KENNEDY% JF'))
 (IN T2 TRACKHIST ((T2 UICVCN) EQ (S2 UICVCN)))
 (? (GCDIST ((S1 PTPX) ,
             (S1 PTPNS) ,
             (S1 PTPY) ,
             (S1 PTPEW) ,
             (S2 PTPX) ,
             (S2 PTPNS) ,
             (S2 PTPY) ,
             (S2 PTPEW)))))
```

```
   OPEN ACCAT $
   FIND FIRST SHIP RECORD OF BLUEAREA AREA $
10 IF ERROR-STATUS = 307 GO TO 11 $
   COMPUTE XAND13 = 0 $
   IF SHIP-NAM NE 'FOX' GO TO 14 $
   COMPUTE XAND13 = 1 $
14 IF XAND13 = 0 GO TO 12 $
   PUSHLP $
   SET TRACKHIST-UICVCN TO SHIP-UICVCN $
   FIND TRACKHIST RECORD $
15 IF ERROR-STATUS = 326 GO TO 16 $
   PUSHLP $
   COMPUTE XSTR41 = TRACKHIST-PTPEW $
   COMPUTE XNUM39 = TRACKHIST-PTPY $
   COMPUTE XSTR37 = TRACKHIST-PTPNS $
   COMPUTE XNUM35 = TRACKHIST-PTPX $
   COMPUTE XSTR18 = TRACKHIST-UICVCN $
   SET TRACKHIST-UICVCN TO XSTR18 $
   FIND TRACKHIST RECORD $
19 IF ERROR-STATUS = 326 GO TO 20 $
   PUSHLP $
   FIND FIRST SHIP RECORD OF BLUEAREA AREA $
22 IF ERROR-STATUS = 307 GO TO 23 $
   COMPUTE XAND25 = 0 $
   IF SHIP-NAM NE 'KENNEDY JF' GO TO 26 $
   COMPUTE XAND25 = 1 $
```

```
26 IF XAND25 = 0 GO TO 24 $
   PUSHLP $
   SET TRACKHIST-UICVCN TO SHIP-UICVCN $
   FIND TRACKHIST RECORD $
27 IF ERROR-STATUS = 326 GO TO 28 $
   PUSHLP $
   COMPUTE XSTR49 = TRACKHIST-PTPEW $
   COMPUTE XNUM47 = TRACKHIST-PTPY $
   COMPUTE XSTR45 = TRACKHIST-PTPNS $
   COMPUTE XNUM43 = TRACKHIST-PTPX $
   COMPUTE XSTR30 = TRACKHIST-UICVCN $
   SET TRACKHIST-UICVCN TO XSTR30 $
   FIND TRACKHIST RECORD $
31 IF ERROR-STATUS = 326 GO TO 32 $
   COMPUTE XNUM36 = XNUM35 $
   COMPUTE XSTR38 = XSTR37 $
   COMPUTE XNUM40 = XNUM39 $
   COMPUTE XSTR42 = XSTR41 $
   COMPUTE XNUM44 = XNUM43 $
   COMPUTE XSTR46 = XSTR45 $
   COMPUTE XNUM48 = XNUM47 $
   COMPUTE XSTR50 = XSTR49 $
   GCDIST XNUM36 XSTR38 XNUM40 XSTR42
          XNUM44 XSTR46 XNUM48 XSTR50 XARG34 $
   PRINT XARG34 $
33 FIND DUPLICATE TRACKHIST RECORD $
   GO TO 31 $
32 * $
   POPLP $
29 FIND DUPLICATE TRACKHIST RECORD $
   GO TO 27 $
28 * $
   POPLP $
24 FIND NEXT SHIP RECORD OF BLUEAREA AREA $
   GO TO 22 $
23 * $
   POPLP $
21 FIND DUPLICATE TRACKHIST RECORD $
   GO TO 19 $
20 * $
   POPLP $
17 FIND DUPLICATE TRACKHIST RECORD $
   GO TO 15 $
16 * $
   POPLP $
12 FIND NEXT SHIP RECORD OF BLUEAREA AREA $
   GO TO 10 $
11 * $
  GO TO XT $
**END**
```

5. Distributed Query Processing using DBMS-20

When it is necessary to access multiple data base systems to answer a single query, intermediate results must be transferred between systems. For uniformity, these answers are always expressed as relations, which we have implemented as temporary files on both the Datacomputer and on DBMS-20. We have provided a uniform interface for the creation, access, and transfer of these temporary files, so that we have completely general capabilities to answer queries that require simultaneous access to data distributed over multiple data bases of either type.

In the case of DBMS-20, the SODA/IQL translator has to keep track of whether a relation to be searched is in the permanent data base (in which case it is susceptible to calc-key searching) or whether it is a temporary file (in which case only sequential searching may be performed). If the latter, it will generate sequential search using FIND NEXT commands, where the relation-name is one of six predefined names of the form SEQ-FILE1, SEQ-FILE2, etc. It also generates a command which tells IQL to bind the name SEQ-FILEn to the actual temporary file, and to open and close it when appropriate.

Example 7 shows the SODA, Datalanguage, and IQL queries that are generated to answer the query "Where are the American ships?", assuming the SHIP relation (which maps the ship name field into the joining field UICVCN) is stored on a Datacomputer and the TRACKHIST relation (which maps UICVCN into last recorded position and date of recording) is stored on DBMS-20.

Example 7:

"Where are the American ships?"

First, the names and UICVCNs of all American ships are put into a temporary file on the Datacomputer by issuing the following SODA query with a special flag set to indicate that the information is to be put into a temporary file:

```
((IN S SHIP ((S NAT) EQ 'US'))
 (? (S NAM))
 (? (S UICVCN)))
```

The SODA query is compiled into the following Datalanguage program which stores the requested information in the temporary file THAASFILE1. The name of the file is then returned to SODA. Temporary files are always generated with 50 fields per tuple, the fields being strings of up to 100 characters. (They are named STRING1, STRING2, etc., and typically most of their values are null.)

```
OPEN %TOP.ACCAT.SAGALOWICZ.TEMPFILE READ;
CREATE THAASFILE1 FILE LIKE TEMPFILE ;
MODE THAASFILE1 WRITE ;
FOR THAASFILE1 , XX1 IN SHIP WITH (XX1.NAT EQ 'US')
BEGIN STRING1 = XX1.NAM STRING2 = XX1.UICVCN END ;
```

SODA requests that the temporary file be transfered to DBMS-20, and obtains the final answer by executing the following SODA query on DBMS-20:

```
((IN S THAASFILE1)
 (IN T TRACKHIST ((T UICVCN) EQ (S STRING2)))
 (? (S STRING1))
 (? (T PTP))
 (? (T PTD)))
```

In the IQL translation of this query the **BINDFILE** statement declares to IQL that the temporary file THAASFILE1 is to be the referent of the dummy relation-name SEQ-FILE2 in subsequent statements.

```
    OPEN ACCAT SEQ-FILE2 $
**BINDFILE** 2 THAASFILE1
    FIND SEQ-FILE2-STRING1 = 'NEXT' FROM BEGINNING $
10  IF ERROR-STATUS = 307 GO TO 11 $
    PUSHLP $
    SET TRACKHIST-UICVCN TO SEQ-FILE2-STRING2 $
    FIND TRACKHIST RECORD $
13  IF ERROR-STATUS = 326 GO TO 14 $
    PRINT SEQ-FILE2-STRING1 TRACKHIST-PTP TRACKHIST-PTD $
15  FIND DUPLICATE TRACKHIST RECORD $
    GO TO 13 $
14  * $
    POPLP $
12  FIND SEQ-FILE2-STRING1 = NEXT $
    GO TO 10 $
11  * $
   GO TO XT $
**END**
```

C. COMPARING RELATIONAL TO CODASYL DBMS'S FOR INTERACTIVE QUERYING

LADDER is now able to generate queries to both the Datacomputer and DBMS-20. Although we have had limited experience with DBMS-20, a number of remarks can already be made concerning the respective advantages of these two types of DBMS. In our system, the Datacomputer is treated as a relational DBMS (although, it can in general be used as a hierarchical DBMS). DBMS-20 is a CODASYL, or network, DBMS. Simplifying a great deal, one may say that the main difference between those two types of DBMS is the way files--or relations--are linked: in a relational DBMS, the link is simply by field values, while in a CODASYL DBMS, the link may also be by explicit pointers. Comparing the relative advantages and disadvantages of these two data models is very subjective, and any general statement will have exceptions in specific situations, but a number of observations stand out in the context of interactive data retrieval using an INLAND-SODA type of front end.

The first remark is that accessing a network-type DBMS is generally much more complex from a programming point of view. The user must be aware of the pointers and must indicate to the system when to traverse them. Our experiments with Datalanguage and IQL indicate that the same user query requires a DBMS query program about two times bigger for DBMS-20 than for the Datacomputer. However, this difference is totally invisible to a user of LADDER or SODA, because these systems provide a uniform interface to either DBMS.

A second remark is that for short interactive queries, there seems to be little advantage in using pointers rather than links by field values. In the particular case of the Blue File, it is our feeling that for short queries, the Datacomputer is generally faster than DBMS-20. On the other hand, for complex queries, and large amounts of data, the links by pointers might be more efficient than links by field values. In the typical use of a natural-language system, we do not expect complex queries to occur very often. They are difficult to express and comprehend in natural language, and the user will generally not want to wait too long to see an answer; he would probably rephrase the query so that it could be handled faster.

Finally, a relational data base provides more uniform access to the information it contains, because it is as easy to follow an interrelation link in one direction as the other. On the other hand, CODASYL pointers are unidirectional, and generally a pointer is kept for only one direction. This difference is very important for query efficiency: queries which are apparently very similar when expressed in natural language (or in a relational language) will behave very differently on a network-type data base. This can also be true on a relational data base, if special indexes or other strong bindings are maintained on some fields but not on others, but it can easily be avoided. In our experiments with the Blue File, we have established the data structure in such a way that the symmetry is guaranteed: if a field is indexed in a relation, and this field is used for an interrelation join, then the corresponding field is also indexed in the other relation. This type of symmetry appears very difficult to obtain in a network data base. In the CODASYL model the asymmetry may be diminished but it appears difficult to eliminate totally.

In summary, a great many discussions have occurred in the past ten years about the respective advantages and disadvantages of the various data models. Although we do not in any sense have the final answer, it appears that when a natural-language front end protects the user from the idiosyncracies of whatever DBMS is used, the differences between the various DBMSs are not terribly significant. Of course in any large system, the problem of choosing a DBMS will remain important, since many data base transactions will bypass the natural-language or interactive front end and access the DBMS directly.

D. CONCLUSION

We have demonstrated that LADDER can be interfaced to multiple data base systems, using diverse query languages and access techniques. Moreover, not only can LADDER use either of the two systems to answer queries, but it can use these heterogeneous systems cooperatively to answer individual queries in cases where neither system by itself has

sufficient information to produce the answer. A beneficial side effect of this work is that LADDER is now compatible with a widely used commercial standard: the CODASYL model. Modifications to LADDER to support this new capability were mostly limited to a specific module of the system; the only modifications outside it were to provide an I/O interface to DBMS-20 to transfer the queries, responses, and temporary files.

VI TOWARD A GENERAL NATURAL-LANGUAGE INTERFACE

A. OVERVIEW

This section of the report describes the progress to date on the new natural-language interface for data base access. It includes an overview of the interface, a more detailed account of several of its major portions, and some results from initial test runs.

The natural-language interface is composed of four major components:

(1) A bottom-up parser (DIAMOND) that uses a linguistically-oriented, general grammar of English.

(2) A set of translators that generate calls to a set of primitive semantic functions from the phrase structures produced by the parser.

(3) A conceptual schema that encodes a model of the application domain in a network structure, and a set of functions that interpret the semantic primitives within the schema.

(4) A query generator that produces a SODA data base query corresponding to the conceptual schema interpretation.

A question is parsed by the DIAMOND parser using the linguistic grammar; the result is one or more (if the question is syntactically ambiguous) phrase structure trees that represent the possible syntactic analyses of the question. Each phrase structure tree is translated into a set of primitive semantic function calls, which construct an interpretation of the phrase structure tree in the appropriate domain. Phrase structure trees may be rejected because they have no interpretation in the domain, or they may be split if they have an ambiguous interpretation. Finally, the interpreted phrase structure tree is converted into a query on the data base.

The system described above was designed to provide flexibility and portability in the natural-language interface. The following sections describe the parsing system DIAMOND, the semantic primitive translators, and the conceptual schema, as well as issues of portability pertaining to them.

B. THE DIAMOND PARSER

1. Stages of Interpretation

The basic framework of the new interface is embodied in a "language definition system" called DIAMOND. DIAMOND may be viewed in broad outline as both a sophisticated programming language and an associated execution system. As a programming language, DIAMOND allows us to describe formally how sentences may be interpreted. The interpretation process is performed in multiple stages under the control of the DIAMOND executive and in accordance with the specifications of the language definition. DIAMOND's flexibility allows us to experiment with different numbers of stages and with alternative sequences for applying various types of knowledge. Currently, we are working with three basic interpretation stages:

The first stage interprets the input "bottom-up" (i.e., words -> phrases -> larger phrases -> sentences). Phrases are constructed in isolation, without reference to the context in which they might be embedded. This stage is used for relatively simple tests that do not depend on surrounding context. It produces a complex data structure (a generalization of a syntax tree) that reflects the decomposition of the input sentence into its component phrases and associates a number of attributes with each phrase. This structure is expanded and refined during subsequent stages of interpretation.

In the second stage of interpretation, each phrase is refined in the context of the preliminary interpretation of the entire sentence. Phrases processed during this stage are generally subsets of those initially considered and thus are more likely (but not guaranteed) to belong to a correct interpretation of the sentence. Consequently, for

the sake of computational efficiency, many of the more extensive processing tasks are delayed until this stage. For example, in this stage we establish a relationship between phrases identified during syntactic analysis and descriptions of (sets of propositions about) objects in the domain model.

In the third stage, the interpretation is further refined within the context of the entire sentence, building on the results of the previous two stages. Major tasks of this stage include delimiting the scopes of quantifiers and associating references to objects with particular entities in the domain model, taking into account the overall dialogue and task context.

The separation of processing into stages has allowed us to examine more easily the question of when during interpretation certain types of knowledge can or should be used. For example, the identification of the referent of a pronoun requires knowledge of the role that the pronoun plays in the sentence, as well as knowledge about the discourse context [20]. Thus it cannot be performed until after the sentence has been interpreted with respect to concepts in the domain. By contrast, simple structural tests (e.g., number agreement) between phrase constituents can be made during the first stage.

2. Grammar Rules

The language definition encoded in DIAMOND consists of (1) a lexicon in which words are separated into categories with associated features and (2) phrase structure rules augmented with procedures to be evaluated during successive stages of processing. Figure 1 presents a noun phrase rule in the language definition (simplified for illustrative purposes). It is discussed in some detail below to show how its different constituents identify linguistic structure, relate linguistic form to domain concepts, and relate the sentence as a whole to its context.

The example noun phrase rule has four constituents. The first part, the phrase structure, indicates the phrase to be formed. Phrases

```
NP = {DET/QUANT} (ADJ) NOUN (PP);                                    |
                                                                     |
   CONSTRUCTOR                                                       |
     (PROGN (@FROM NOUN NUMBER)                                      | Stage 1
            (@FROM DET DEF)                                          |
            (COND ((@ ADJ)(OR (AGREE TYPE ADJ NOUN)                  |
                              (F.REJECT 'NO-AGREEMENT)))))           |
                                                                     |
   TRANSLATOR                                                        |
     (@SET SEMANTICS (COMBINE (@ SEMANTICS ADJ)                      | Stage 2
                              (@ SEMANTICS NOUN)))                   |
                                                                     |
   INTEGRATOR                                                        |
     (@SET D.IDENT (RESOLVE (@ SEMANTICS)))                          | Stage 3
```

Figure 1 Sample Noun Phrase Rule

are generally standard linguistic units, such as noun phrases and verb phrases. The rule specification allows for optional and alternative elements. Thus, as in this example, one rule for interpreting a noun phrase may allow many alternatives. In this example, the DET (determiner) and QUANT (quantifier) constituents are alternatives, indicated by the braces, and the ADJ and PP constituents are optional, indicated by the parentheses. The NOUN constituent is required.

The second part of the rule, the CONSTRUCTOR, is a procedure to be evaluated when a phrase is formed using this rule. Generally, a constructor assigns attributes to the phrase and tests for consistency among the attributes of its constituents. For example, (@FROM NOUN NUMBER) copies the value of the NUMBER attribute from the NOUN constituent to the NP phrase being built. Procedures can rate an interpretation of a phrase, based on an assessment of its likelihood, and reject unlikely ones. The F.REJECT statement at the end of the CONSTRUCTOR will reject a proposed phrase if the ADJ and NOUN are not of the same type (i.e. the type of adjective in the phrase cannot modify the type of noun present). Unlikely phrases need not be rejected but can be marked as "less-than-good" (e.g., FAIR or POOR); they can be used if no better interpretation is found.

Much of what is commonly called "syntactic" information (information about words and phrases and how they combine independently of their relationship to the domain) is encoded in the phrase structure and constructor parts of the language definition. We have encoded and tested a general, linguistically-motivated, set of rules for syntactic analysis that covers a wide range of English constructions. Sample sentences covered by this syntax include:

WHAT IS THE NORMAL STEAMING TIME FROM NORFOLK TO GIBRALTAR FOR THE KENNEDY?
WHICH U.S. NAVY FF'S HAVE CASREPS INVOLVING SONAR SYSTEMS?
ASSUMING A MINIMUM ALLOWABLE FUEL STATE OF 30 PERCENT, HOW FAR COULD THE GROZNY AND VARYAG STEAM AT MAXIMUM SPEED?
WHAT SHIPS IN THE SOUTH ATLANTIC HAVE A DOCTOR ONBOARD?
TO WHAT CLASS DOES THE BATON ROUGE BELONG?

The third part of the rule, applied during the second stage of processing, is the TRANSLATOR procedure. A translator for a phrase is evaluated after the phrase has been combined with others to form a syntactic analysis for an entire sentence. Thus, unlike the constructor, the translator has available information about how the phrase fits into the sentence. In Figure 1, the operation is simply one of combining the semantics for the ADJ with those for the NOUN.

The fourth rule component, the INTEGRATOR procedure, is applied during the third stage of processing. An integrator for a rule specifies how to relate the concepts mentioned in a phrase to specific domain entities. During this stage, for example, pronoun referents are resolved against identified entities in the domain.

Within the translator and integrator procedures, we have formalized our knowledge of the relationship between syntactic structures and meanings by the use of a common set of semantic functions called semantic primitives, described in the next section. Translator and integrator functions which construct semantic primitive calls from the syntax have been implemented for major portions of the current linguistic grammar.

3. Summary

As we have seen, DIAMOND currently separates the three tasks of construction, translation, and integration into well-defined successive processing steps. While this partitioning provides a good framework with which to understand the interpretation process, it gives rise to problems of efficiency. Chief among these problems is the fact that domain information does not enter the parsing process until all the syntactic constructions for an sentence have been produced. For long sentences, the number of syntactic structures produced can be large, with a consequent increase in the processing time required at successive stages. One of the problems for future research involves finding reasonable strategies for combining the three processing stages, so that domain information can reduce the ambiguity of the construction process.

C. SEMANTIC PRIMITIVES

The semantic primitives provide a flexible, uniform interface between linguistic analysis of a sentence and domain semantics. Sentences with similar meaning but differing phrase structures are mapped into similar sets of primitive semantic-function calls. For example, consider the simple passive transformation which relates the two sentences:

(1) WHO COMMANDS THE KENNEDY?

(2) BY WHOM IS THE KENNEDY COMMANDED?

Even though the surface syntactic structure varies, the propositional structure of (1) and (2) is the same: a predicate, COMMANDS, with two arguments, THE KENNEDY and WHO. Both sentences would invoke the same semantic primitive to construct a representation of the propositional content. The translation from surface structure to semantic primitives thus encodes linguistic knowledge about the way surface structure relates to meaning.

It is important to note that the translation into semantic primitives is domain-independent, that is, it was not designed with a

particular subject area in mind. Thus we hope that it will provide a uniform way of analyzing the syntactic structure of sentences, relieving the natural-language interface builder of a large part of his task.

Currently there are four categories of semantic primitives:

(1) Propositional primitives. These primitives are derived from the propositional content of a sentence. They include calls to create sets (corresponding mainly to noun phrases) and propositions relating these sets (verb phrases, adjectival modifiers, etc.).

(2) Request primitives. These add various types of request information from the sentence. They include counting ("how much," "how many"), yes/no, and set specification ("what ships are ...").

(3) Modality primitives. These deal mainly with the tense and mood aspects of verbs.

(4) Quantification primitives. These deal with the quantificational structure of a sentence, including explicit ("each," "every," "all," etc.) and implicit quantification (e.g., superlatives).

To illustrate the way in which the various types of semantic primitives are invoked, consider the query:

WHEN COULD ALL THE SUBS IN THE MED REACH GIBRALTAR?

The propositional structure of this sentence is straightforward: a predicate, REACH, which connects three arguments, GIBRALTAR, SUBS IN THE MED, and WHEN, a time argument. In addition, the argument SUBS IN THE MED itself has a propositional structure, based on the implicit containment predicate IN, which relates SUBS and MED. Thus two calls to the propositional semantic primitives are invoked; first to build up the phrase SUBS IN THE MED, and then to attach arguments to the predicate REACH.

The request primitives are invoked for the argument WHEN to indicate that the value of this argument is being requested by the query. The quantifier primitives are invoked to attach the quantifier ALL to the phrase SUBS IN THE MED. Finally, the modality primitives are invoked to indicate that the modal auxiliary COULD is attached to he predicate REACH.

At this point, the translation process has extracted all the information it can from the sentence, and converted it into a set of calls on the semantic primitives. Obviously, there is more information in the sentence which cannot be encoded by these four types of semantic primitives. Here we rely on the inherent flexibility of the primitives; they can be augmented as our linguistic analysis grows more demanding. For example, if there is some theory of stative vs. active interpretation of verb structure that we wish to incorporate, we simply define a new class of primitive semantic functions, along with the programs that produce calls to these functions from the phrase structure of a sentence.

The semantic primitives also provide a flexible approach to the problems facing an interface builder, since he need use only the primitives required by the problem at hand. For example, in designing the interface for a query system, the propositional and request primitives would be most important. Primitives which dealt with more subtle aspects of the sentence, such as quantification or the modality of verbs, could be added later.

D. CONCEPTUAL SCHEMA

In contrast to the current LADDER system, in which a question is converted directly into a data base query, the new system uses a conceptual schema as the target representation for translation. A conceptual schema is a representation of the domain as it appears to the user; in its simplest form, a conceptual schema must encode the user's knowledge of the objects in the domain and the relationships which hold between them. This is different from the data base schema, which represents the system's view of the way data is stored in the data base. As discussed in the next section, the relation between the conceptual schema and the data base schema can be quite complex. In our system, the conceptual schema encodes knowledge about ships, officers, ports, cargoes, and other domain objects, and the relationships between them.

There are several reasons why a conceptual schema makes the task of interpreting the question easier and provides some capabilities that could not be accommodated by direct data base translation alone. One reason is that it allows for conceptual completeness. A data base typically will have information only about some portion of a given domain. The user may not be aware that some information is missing. Consider the queries:

WHAT SOVIET SHIPS HAVE ASW CAPABILITY?
WHAT U.S. SHIPS HAVE ASW CAPABILITY?
WHAT SHIPS HAVE ASW CAPABILITY?

The user would expect a competent interface to understand all three queries, if it understood any one of them. Yet an interpreter which depends on a data base schema for its model of the domain may very well interpret one of the queries correctly, while either failing to interpret the others, or giving wrong answers to them. For example, suppose one of the data base files contained information on the weapons capabilities of U.S. ships only. Then the second query could be interpreted correctly, but none of the others. An interpretation process which depended on the data base schema wouldn't know whether only U.S. ships could have weapons capability, or whether the particular data base just didn't have the information for other nations' ships. In order to distinguish the case of a particular data base not having a certain attribute for an object from the case where the domain does not permit the object to have the attribute, a better model of the domain than the data base schema is required. The conceptual schema is such a model; it provides a measure of semantic completeness in the interpretation process, independent of a particular data base.

A second advantage of having a conceptual schema comes from the ability to have inference processes use the schema. We are currently using inferences about the domain in a rudimentary way to help in such tasks as disambiguation; a short example occurs at the end of this section. More advanced research tasks which we would like to pursue include sophisticated semantic processing of the parse tree (e.g., noun-

noun phrases*), query interpretation, and dialogue issues, such as using knowledge of the user's goals to choose an apropriate response. All of these tasks require the ability to reason about the domain of discourse. In order to do such reasoning in a general way, a data-independent representation of objects in the domain and their interrelationships is essential.

Finally, a conceptual schema enhances portability to different data bases within the same domain. A conceptual schema provides a measure of independence of the domain from a particular data base. Thus, in switching the interface between data bases which deal with the same domain, we can expect the conceptual schema to remain mostly unchanged. The interface builder must indicate how conceptual schema predicates and sets are mapped into a particular data base schema. In our system a code generator uses this mapping to derive a SODA query on the data base from a conceptual schema representation.

Our implementation of a conceptual schema is based on a network representation of objects in the domain and their relationships. This approach is based on techniques for knowledge representation developed at SRI over several years [21], although it may also be viewed as being "frame-like" in the sense of a number of recent AI languages [22] [23]. There are two basic parts to the representation: an object set taxonomy which gives subset and superset relations among various sets in the domain; and delineations, which specify the types of arguments which relations can have. In addition, there are some metastructures that control inferencing processes that operate during the interpretation of a query, and which are thus not strictly a part of the domain representation.

Although exact details of the representation are too complicated to go into here, we will attempt to give an idea of how it is used to actually construct an interpretation of a query. Consider the question:

WHO COMMANDS THE LAFAYETTES?

---

*For example "oil tankers" are tankers that carry oil, but "U.S. destroyers" are destroyers owned by the U.S.

The lexical entries for WHO and LAFAYETTE have pointers to sets in the domain representation. Figure 2 shows the relevant portion of the taxonomy. WHO refers to the class of LEGAL-PERSONS; subsets of LEGAL-PERSONS, labeled by s-arcs, are COUNTRIES and OFFICERS (note that one could ask, WHO OWNS THE KENNEDY, where WHO refers to a country). LAFAYETTE refers to the individual node LAFAYETTE, which can be an element (e-arc) of either NAVAL-SHIP-CLASSES or NAVAL-SHIPS, i.e., the Lafayette class as opposed to the ship Lafayette. At this point, then, there is ambiguity in the references for both nouns. An important part of the representation is that these ambiguities can be easily accommodated by the taxonomy structure.

The representation also shows what relations can exist between objects in the domain. The delineations for the COMMANDINGS and NAVAL-CLASS-MEMBER relations are shown in Figure 3. The arcs emanating from these nodes are labeled with the argument type, and point to the set from which the argument must be drawn, e.g., the "commander" argument of COMMANDINGS must come from the set OFFICERS.

The verb COMMANDS in the query refers to the COMMANDINGS relation. Syntactic information from the sentence tells us that WHO, as subject, must be the "commander" argument. Thus we have identified WHO as belonging to the OFFICERS subset of LEGAL-PERSONS, rather than COUNTRIES.

Similarly, we find that THE LAFAYETTES must be the "commanded" argument of COMMANDINGS, i.e., it belongs to NAVAL-SHIPS. However, there are additional clues from syntax which force a different interpretation. THE LAFAYETTES is plural, and hence demands a reference in the domain which is not necessarily a single object. Since the LAFAYETTE node interpreted as a NAVAL-SHIPS is a single element, this can't be the correct interpretation. In an alternate interpretation, the node LAFAYETTE is an element of NAVAL-SHIP-CLASSES; by the NAVAL-CLASS-MEMBER relation, a single class can stand for a set of ships, and so is an acceptable interpretation of the plural syntax. Thus the syntactic clue of plurality is used to disambiguate the semantic

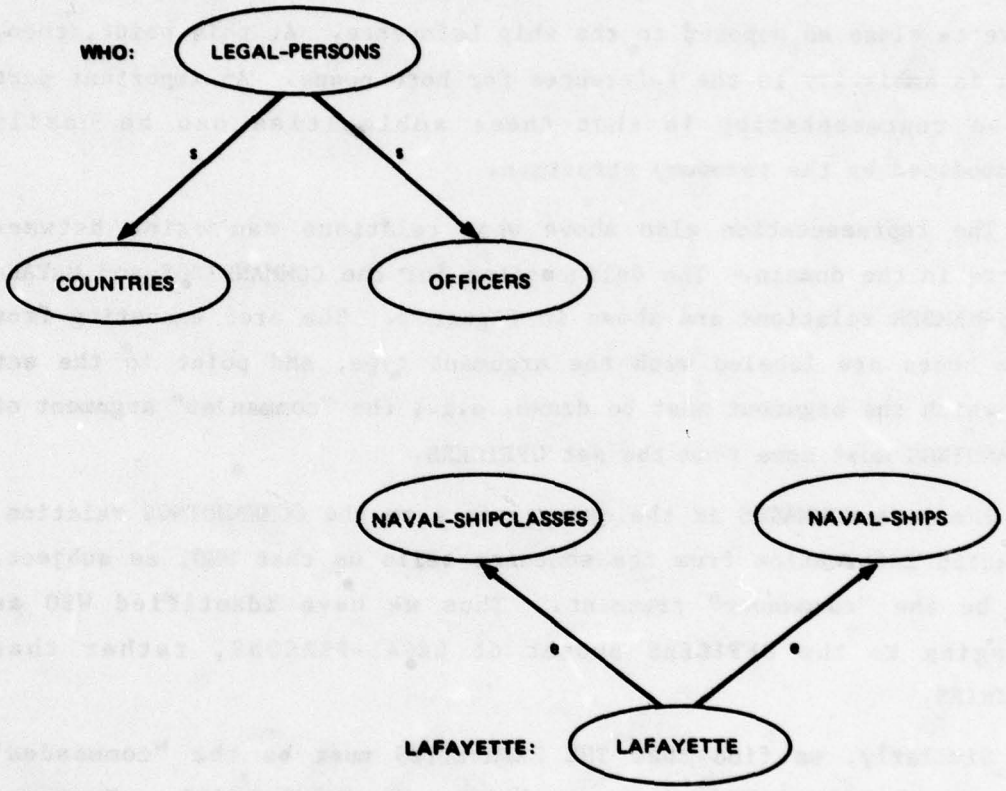


FIGURE 2 CONCEPTUAL SCHEMA TAXONOMY

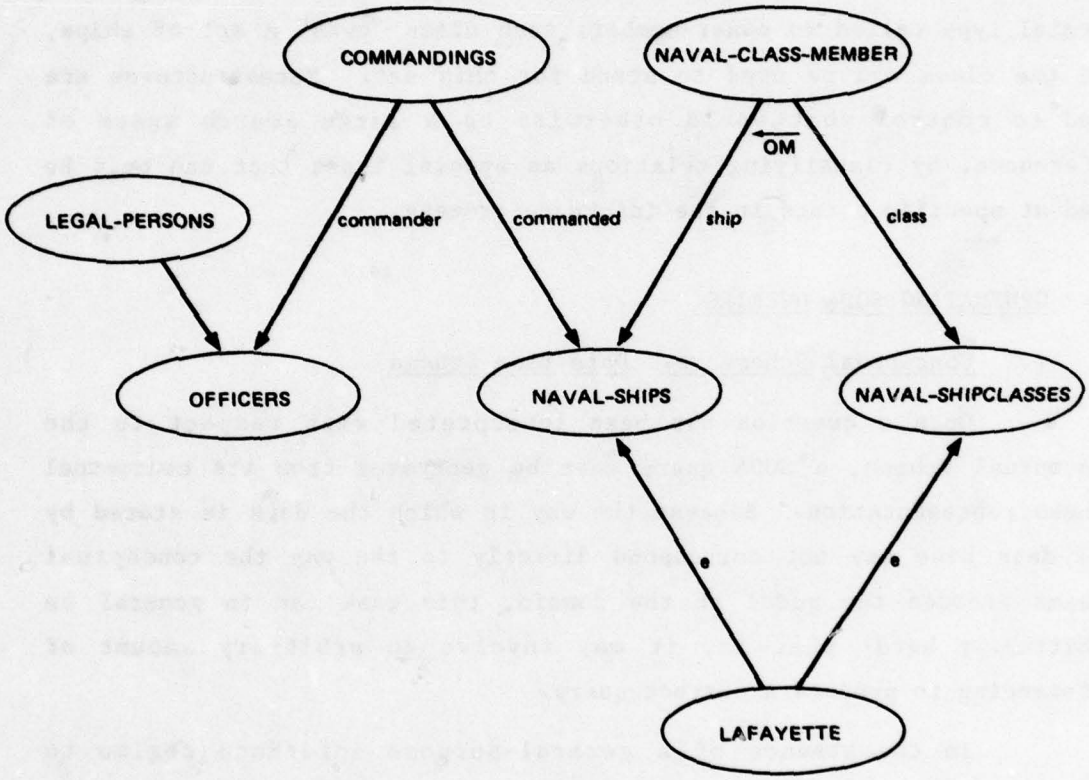


FIGURE 3 CONCEPTUAL SCHEMA DELINEATIONS

interpretation of the reference to the node LAFAYETTE. Note that this type of inference is impossible in the current LADDER system, where many syntactic clues, among them plurality of noun phrases, are simply disregarded by the grammar. To LADDER, WHO COMMANDS LAFAYETTE and WHO COMMANDS THE LAFAYETTES look exactly alike.

The OM symbol on the NAVAL-CLASS-MEMBER relation is part of the metastructure of the representation. It signals that this relation is a special type called an owner-member: each class "owns" a set of ships, and the class can be used to stand for this set. Metastructures are used to control what would otherwise be a large search space of inferences, by classifying relations as special types that can only be used at specific points in the inference process.

E. GENERATING SODA QUERIES

1. Conceptual Schema vs. Data Base Schema

Once a question has been interpreted with respect to the conceptual schema, a SODA query must be generated from its conceptual schema representation. Because the way in which the data is stored by the data base may not correspond directly to the way the conceptual schema encodes the model of the domain, this task can in general be arbitrarily hard; that is, it may involve an arbitrary amount of inferencing to produce a correct query.

In the absence of a general-purpose inference engine to produce SODA queries, we have concentrated on several techniques which promise to cover the most common types of attachments between a data base schema and a conceptual schema. The rest of this section describes these techniques, and gives examples of how they can be used to attach our conceptual schema to the schema of the Blue File [11] data base.

2. Simple Predicates

In the best case, a predicate in the conceptual schema will correspond to a relation in the data base schema. This is true for the

predicates given in the previous section, namely, COMMANDINGS and NAVAL-CLASS-MEMBER. The commander of a unit is found in the UNIT relation, and the class of a ship is found in the SHIP relation. The actual generation from the conceptual schema involves setting up SODA tuple variables for each predicate in the query, and SODA terms for each set. Where two predicates share a common argument, an equijoin is produced in the query code. Thus in Figure 4 we have the conceptual schema representation of WHO COMMANDS THE LAFAYETTES, and the SODA query generated, with the origin of the various subexpressions of the SODA query indicated.

This example was rather straightforward; various subtleties of the data base design can cause complications. For example, an object in the conceptual domain, like SHIPS, is not always referred to in a consistent manner by the data base. Some files may use the name of the ship, while others use the UIC or VCN of the ship instead. The generation process must take these quirks into account.

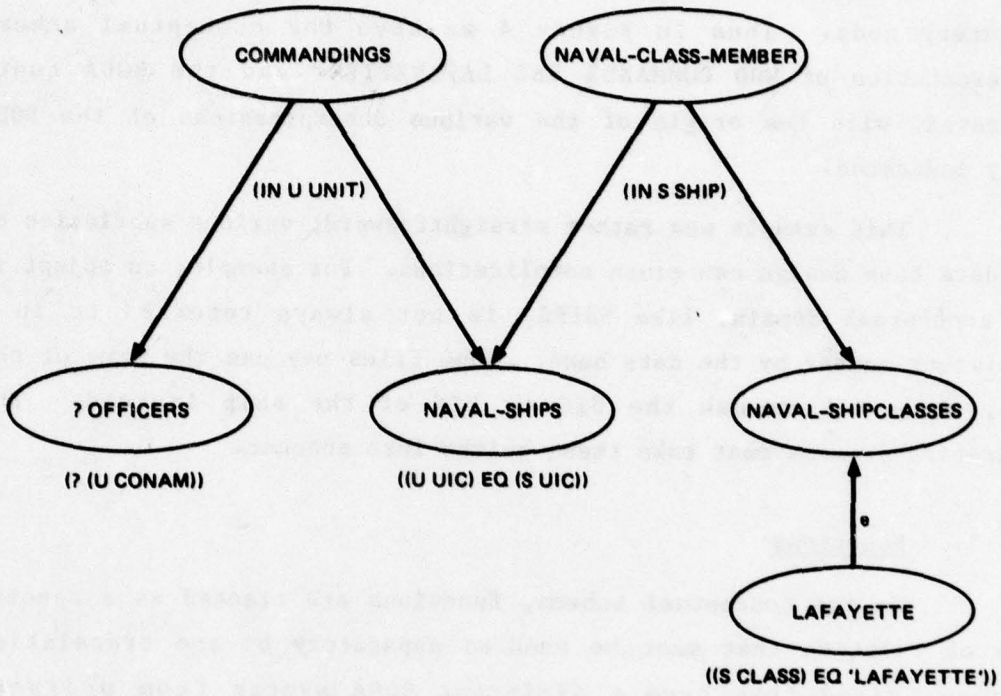
3. Functions

In the conceptual schema, functions are treated as a special type of relation that must be handled separately by the translation process, since they have a different SODA syntax from ordinary relations. Also, care must be used to ensure that their arguments are defined before they are invoked. For example, consider the question:

WHAT POSITIONS ARE 300 MILES FROM THE LAFAYETTE?

Such a query could not be answered by the data base because the distance function needs to have both of its position arguments defined. For the above question, the result of the distance function is specified (300 miles), as well as one of its position arguments (the Lafayette); but the function cannot be inverted to generate a list of positions that are 300 miles from the Lafayette. The generation process must recognize this condition and give an appropriate error message.

A second peculiarity of functions is that they may not be executable by the DBMS that contains the data base. In this case, the



SODA QUERY: ((IN U UNIT)
(? (U CONAM))
(IN S SHIP)
((U UIC) EQ (S UIC))
((S CLASS) EQ 'LAFAYETTE'))

FIGURE 4 GENERATION OF A SODA QUERY FROM THE CONCEPTUAL SCHEMA

generation process must send off SODA queries to retrieve the arguments of the function, evaluate the function, and then put the result into another SODA query.

4. Implicit Sets

Not all sets that the conceptual schema knows about correspond to relations in the data base. For instance, there is a SHIP relation where each tuple gives information about a ship, but there is no TANKER relation. Such conceptual schema sets as TANKERS usually have an implicit definition in the data base, in the sense that we could form a more or less complicated SODA restriction over some relation to extract the set. If we are lucky, there will be a single field whose value will tell us whether a tuple is a member of a restricted set or not, e.g., the TYPE field gives us TANKERS. It is easy to see, however, that there may be cases where the data base does not have the necessary information, or the query language is not powerful enough, given the form of the data base, to extract it. A set of this sort which can't be extracted from the Blue File, although the information is there, is: SHIPS THAT LEFT PORT AT NIGHT.

5. Existence

At times a data base field will indicate the existence of an object or property, rather than its exact value. A typical example of this is the MED field in the SHIP relation. If the value of this field is D, then there is a doctor on board the ship; but it doesn't give the identity of the doctor. The presence of these existence fields causes problems for generation of data base queries, because it means some questions that ask for the existence of an object will be answerable, while others that ask for the name of the object will not. Consider the questions:

IS THERE A DOCTOR ON THE BIDDLE?
WHAT DOCTOR IS ON THE BIDDLE?
HOW MANY DOCTORS ARE ON THE BIDDLE?

Only the first of these questions can be answered, given the above definition of the MED field.

It should be obvious from the above examples that the process of generating a correct SODA query from the conceptual representation can be a complicated one. To take this even one step further, we could imagine trying to come up with a reasonable query when there was no correct way to answer the question directly, e.g., to respond "At least one" if asked "How many doctors are on the Biddle?". It would then seem impossible to generate SODA queries without considering general dialogue issues such as the goals and presuppositions of the user.

F. PORTABILITY

As we stated at the beginning of this section, one of the major goals of our work on the new natural-language interface has been to enhance the portability of the LADDER system. By incorporating both a level of domain-independent semantic primitives and the domain-dependent conceptual schema, we have attempted to facilitate portability both to new domains and, within a domain, to new data bases.

In transporting the natural-language interface to a new domain, there are two tasks facing the interface builder: to write the semantic primitive functions in the new domain, and to add an appropriate domain-dependent lexicon. The semantic primitive formalism provides reasonable guidelines for both tasks. While these tasks are by no means trivial, most of the work involved in syntactic analysis, as well as some domain-independent semantic analysis, has been incorporated into the phrase structure translators. Thus the interface builder is relieved of a large part of the linguistic analysis work necessary to build the interface.

Once the semantics of a particular domain has been specified in a conceptual schema, transportability between data bases dealing with that domain is straightforward. The conceptual schema need only be designed once for a given domain.

Both types of portability remain to be tested in the future, to see how transportable the interface is for new real-world domains and data bases. It is expected that defining a new domain will still be a

significant problem, while transferring between data bases within a domain will be fairly easy.

G. STATE OF IMPLEMENTATION AND PRELIMINARY RESULTS

A subset of the translators from phrase structures to primitive semantic function calls has been implemented. These are mostly the propositional and request primitive calls. This subset is sufficiently complete to cover the linguistic construction in approximately 90 percent of our test query set of 249 sentences. Significant gaps yet remaining include translators for infinitive phrases, imperatives, and quantifiers.

The conceptual schema for the navy command-control domain currently has about 110 sets and subsets pertaining to objects (ships, planes, etc.), and about 100 relations between them (speed, employment, etc.).

The lexicon currently contains 12 adjectives, 12 verbs, 34 prepositions, and 141 nouns with their complete semantic (domain dependent) attributes. This is sufficient to cover about half the test queries. As we attempt to interpret more test queries, both vocabulary and the conceptual schema will be incrementally increased.

The translators, conceptual schema, and lexicon are currently adequate to interpret 112 of the 210 test queries (54 percent) that can be answered from the Blue File data base. ("Interpret" means that the interface is capable of deriving a conceptual schema representation for these queries.) For those 112 interpretable sentences, the grammar produced 311 phrase structure trees, an average of 2.8 per sentence. From these phrase structure trees, 142 interpretations were produced, an average of 1.3 interpretations per sentence. Work on the program to generate SODA queries from conceptual schema interpretations has not yet been completed.

VII PUBLICATIONS AND PRESENTATIONS

A. PUBLICATIONS

Grosz, B. J., "Focusing in Dialogue," in TINLAP-2 Theoretical Issues in Natural Language Processing-2, D. L. Waltz (ed.) pp. 96-103, University of Illinois, Urbana, Illinois (July 1978).

Hendrix, G. G. et al., "Developing a Natural Language Interface to Complex Data," ACM Transactions on Database Systems, Vol.3, No. 2, pp. 105-147 (June 1978).

Hobbs, J. R., "From English Descriptions of Algorithms into Programs," Proceedings, Annual Conference, Association for Computing Machinery, pp. 323-329, Seattle, Washington (October 1977).

Hobbs, J. R., (with S. Rosenschein) "Making Computational Sense of Montague's Intensional Logic," Artificial Intelligence, Vol. 9, pp. 287-306 (December 1977).

Hobbs, J. R., "Resolving Pronoun References," Lingua, Vol. 44, pp. 311-338 (1978).

Hobbs, J. R., "Coherence in English Discourse," (extended abstract) Information Abstracts, The International Congress on Computational Linguistics, Bergen, Norway (August 1978).

Hobbs, J. R., "Coherence and Coreference," SRI Artificial Intelligence Center Technical Note 168, SRI International, Menlo Park, California (August 1978). Also in Cognitive Science, Vol. 3, No. 1, pp. 67-90 (January-March 1979).

Hobbs, J. R., and Robinson, J. J., "Why Ask?" SRI Artificial Intelligence Center Technical Note 169, SRI International, Menlo Park, California (October 1978). Also to appear in R. Freedle (ed.), Discourse Processes: A Multidisciplinary Journal.

Robinson, J. J., "Purposeful Questions and Pointed Answers," (extended abstract) Information Abstracts, The International Congress on Computational Linguistics, Bergen, Norway (August 1978).

Sacerdoti, E. D., "A LADDER User's Guide," SRI Artificial Intelligence Center Technical Note 163, SRI International, Menlo Park, California (October 1978).

B. PRESENTATIONS

17 October 1977 — Jerry R. Hobbs, "From English Descriptions of Algorithms into Programs," Annual Conference, Association for Computing Machinery.

18 October 1977 — Barbara J. Grosz (with Ann E. Robinson), "Long Range Issues in Natural-Language Research," Seminar on Artifical Intelligence, Computer Science Department, University of California, Berkeley, California.

23 November 1977 — Jerry R. Hobbs, "From English Descriptions of Algorithms into Programs, Information Sciences Institute, University of Southern California, Marina del Rey, California.

18 January 1978 — Daniel Sagalowicz, "LADDER," Computer Science Seminar, University of Pennsylvania, Philadelphia, Pennsylvania.

19 January 1978 — Earl D. Sacerdoti, "Managing Errorful Data," Workshop on Semantic Error Detection, Naval Research Laboratory.

19 January 1978 — Daniel Sagalowicz, "Semantic Update Checking," Workshop on Semantic Error Detection, Naval Research Laboratory.

23 January 1978 — Earl D. Sacerdoti, "Language Access to Distributed Data with Error Recovery," General Motors Research Laboratories, Warren, Michigan.

25 January 1978 — Jerry R. Hobbs, "Coherence in English Discourse," Department of Computer Science, University of California, Berkeley, California.

16 February 1978 — Earl D. Sacerdoti, "Representations for Problem Solving," Stanford University Computer Science Department.

13 March 1978 — Jerry R. Hobbs, "Coherence and Coreference," Department of Computer Science, University of California, Berkeley, California.

12 April 1978 — Robert C. Moore, "Logical Properties of PLANNER-Like Languages-I," Seminar on Representation of Knowledge, Computer Science Department, Stanford University, Stanford, California.

19 April 1978 — Robert C. Moore, "Logical Properties of PLANNER-Like Languages-II," Seminar on Representation of Knowledge, Computer Science Department, Stanford University, Stanford, California.

20 April 1978 — Earl D. Sacerdoti, "Natural Language Access to Data Bases," Computer Science Dept., Stanford University.

16 May 1978 Jerry R. Hobbs, "Coherence and Coreference," Department of Linguistics, Stanford University, Stanford, California.

22 May 1978 Gary G. Hendrix, "The Spectrum of Natural Language Research at SRI International," Workshop on Applied Natural-Language Systems, Office of Naval Research, Washington, D.C.

24 May 1978 Barabara J. Grosz, "Focusing and Description in Natural Language Dialogues," Workshop on Computational Aspects of Linguistic Processing and Discourse Setting, University of Pennsylvania, Philadelphia, Pennsylvania.

27 May 1978 Daniel Sagalowicz, "LADDER," Data Base Seminar, Computer Science Department, Stanford University, Stanford, California.

31 May 1978 Earl D. Sacerdoti, "LADDER," Naval Postgraduate School, Monterey, California.

31 May 1978 Daniel Sagalowicz, "Natural Language Access to Databases," Panelist at SIGMOD Conference, Austin, Texas.

20 June 1978 Robert C. Moore, "Reasoning about Knowledge and Action," Artificial Intelligence Seminar, Computer Science Department, University of California, Berkeley, California.

22 June 1978 Robert C. Moore, "Introduction to Modal Logic," Modal Logic Seminar, Computer Science Department, Stanford University, Stanford, California.

19 July 1978 Earl D. Sacerdoti, "Applications of Artificial Intelligence to Data Base Management," Data Base Management Course, University of California Extension, Santa Cruz, California.

25 July 1978 Earl D. Sacerdoti, "Natural Language Access to Data Bases," Command and Control Information Systems Workshop, Naval Postgraduate School, Monterey, California.

26 July 1978 Barbara J. Grosz, "Focusing in Dialogue," Theoretical Issues in Natural Language Processing-2, University of Illinois, Urbana, Illinois.

29 July 1978 Jane J. Robinson and Jerry R. Hobbs, "Why Ask?" (presented by Jane J. Robinson) Linguistic Society of America, Summer Meeting, Urbana, Illinois.

15 August 1978 Jane J. Robinson, "Purposeful Questions and Pointed Answers," 7th International Congress on Computational Linguistics, Bergen, Norway.

15 August 1978 Earl D. Sacerdoti, "Interactive Data Base Query Systems," Hewlett-Packard Laboratories, Santa Clara, California.

17 August 1978 Jerry R. Hobbs, "Coherence in English Discourse," (presented by Jane J. Robinson) 7th International Congress on Computational Linguistics, Bergen, Norway.

Appendix A

FORMAL DEFINITION OF THE SODA QUERY LANGUAGE

Appendix A

FORMAL DEFINITION OF THE SODA QUERY LANGUAGE

The syntax of the SODA query language is most easily described by a context-free grammar, plus some constraints on the occurrence of variables. The grammar for SODA is extremely simple, having only the following five rules:

```
query -> ([binder | restriction | (? term)]+)

subquery -> [binder | restriction]* binder [binder | restriction]*

binder -> (IN tuplevar relation [restriction]* ) |
          (MAX term subquery) |
          (MIN term subquery) |
          (MAX1 term subquery) |
          (MIN1 term subquery) |
          (COUNT countvar subquery)

restriction -> (ALL subquery) |
               (SOME subquery) |
               (NONE subquery) |
               (AND [restriction]+) |
               (OR [restriction]+) |
               (NOT restriction) |
               (term comparison term)

term -> (function ([term]+)) |
        (tuplevar field) |
        countvar |
        constant
```

In this grammar, nonterminal symbols are written in lower case, and terminal symbols are written in upper case. To make the grammar more concise, we have allowed the right side of a rule to be written as a regular expression. The notation "...|..." indicates an alternative, the notation "[...]*" indicates a sequence of any length greater than or equal to zero, and the notation "[...]+" indicates a sequence of any length greater than zero. Note that the parentheses appearing in the

grammar are part of the SODA language which is being defined, while the square brackets are part of the notation in which the grammar is written.

SODA queries are composed of three principal types of expressions: binders, which bind variables to refer to data extracted from the data base, restrictions, which restrict the data, and question-mark expressions (selectors), which request retrieval of parts of the data. A SODA query is any nonempty, parenthesized sequence of these expressions that satisfies the constraints on the occurrence of variables which will be discussed shortly. If the sequence of expressions includes one or more binders, then the query implicitly defines a set of tuples, and the selectors in the sequence specify a projection of that set which is to be returned as the answer to the query. If there are no selectors in the query, then it is interpreted as a yes/no question, asking whether the set defined by the query is nonempty. If the query is simply a sequence of restrictions, then it is interpreted as a yes/no question asking whether all of the expressions in the sequence are true.

The structure of the language is recursive, with MAX, MIN, MAX1, MIN1, COUNT, ALL, SOME, and NONE being operations on certain sequences of expressions which would themselves be well-formed queries. We will call such sequences subqueries, and they must meet the following conditions: first, since it only makes sense to return information from the top level, no selectors are allowed in the sequence. Furthermore, we insist that there be at least one binder in the sequence, since there must be some data from the data base to maximize, minimize, count, or quantify over.

The binders include IN expressions, COUNT expressions, and MAX, MAX1, MIN, and MIN1 expressions. An IN expression sets a variable to range over the set of tuples in a relation (or a restricted subset, if any restrictions are specified). A COUNT expression sets a variable to the number of tuples in the set defined by a subquery. MAX and MIN expressions pick out all the tuples in a set for which some term has the

largest or smallest value. MAX1 and MIN1 expressions do the same, except that they pick out a single tuple from this set. MAX1 and MIN1 can be executed more efficiently than MAX and MIN, so they are to be preferred when applicable, such as when it is known on semantic grounds that there can be only one tuple in the set of interest that has the maximum or minimum value (e.g., there can be only one most recent position report for a ship).

Restrictions include simple Boolean restrictions with AND, OR, and NOT, plus universally quantified restrictions using ALL and existentially quantified restrictions using SOME. [(NONE...) is an abbreviation for (NOT (SOME...))]. The details of how these constructs are interpreted are explained in the discussion of the examples in Section IV.B.

The grammar does not specify what the relations, fields, functions, comparisons, constants or variables are. Constants include numbers and any character strings enclosed in single quotes, such as 'US'. Any other alphanumeric character string can be used as a variable. There are two types of variables: tuplevars, which range over the tuples of a given relation, and countvars, which are used to refer to the result of a counting operation. There need not be any difference in form between tuplevars and countvars, but no symbol can be used as both within the same query.

The other categories not specified by the grammar are all implementation-dependent. Fields and relations obviously depend on the particular data base being accessed. The functions and comparisons depend on the capabilities of the underlying DBMSs in which the queries are actually executed. In the current implementation of SODA, there are four navigation functions (e.g., GCDIST, for computing the great circle distance between two geographical locations), and the comparison operators are EQ, NE, LE, GE, LT, and GT, representing "equal," "not equal," "less than or equal," "greater than or equal," "less than," and "greater than or equal," respectively.

To explain the constraints on the occurrence of variables, we need to define several notions. A variable which is the second element of an IN or a COUNT expression is said to be introduced by that expression. The smallest COUNT, SOME, NONE, or ALL expression that includes the expression that introduces the variable is said to be the scope of the variable. If the expression that introduces the variable is not inside any COUNT, SOME, NONE, or ALL expression then the scope of the variable is the entire query. An occurrence of a variable is bound by the expression which introduces it if the occurrence is contained by every MAX or MIN expression that contains the expression which introduces the variable; otherwise, the occurence is bound by the largest MAX or MIN expression which does not contain the occurrence but does contain the expression which introduces the variable. We can now state the contraints on the occurrence of variables as follows:

(1) No variable may occur in the query unless it is introduced by some expression in the query.

(2) No variable may be introduced by more than one expression.

(3) No variable may occur outside its scope.

(4) The relation "X contains an occurrence of a variable bound by Y" must not form any circular chains of MAX and MIN expressions.

The first rule ensures that the range and scope of every variable is defined. The second rule simply means that the same variable can't be used in two different ways in the same query. This is actually slightly stronger than it needs to be, since two variables which have nonintersecting scopes could be the same without creating logical confusion, but queries are simpler to process and easier to understand if this is not done.

The third rule prevents using a variable in a context where the reference doesn't make sense semantically. It is easiest to think of a variable as referring to a particular tuple in a set of tuples. Inside a COUNT, SOME, NONE, or ALL expression, the variable refers to each tuple in the set in turn, as in "for all tuples in the SHIP relation

such that the tuple..." Outside the expression, however, there is no way to determine which tuple is being referred to. This contrasts with MAX (and MIN) expressions, where, although the variable refers to each tuple in turn inside the expression, there is a definite referent for the variable outside the expression, namely, the tuples for which the term being maximized has the greatest value.

The final rule forces the definitions of sets that are being maximized or minimized over to be noncircular. One MAX or MIN operation can refer to the result of another, but only if the second is well defined without referring to the first.

To put the SODA query language in perspective, we can compare it to Codd's original language based on the relational calculus, DSL ALPHA [24] [25]. One major difference between the two languages is that SODA is only a data retrieval language, whereas ALPHA also permits updataing the data base. In their power to express queries, the two languages are fairly close. ALPHA has the ability to request retrieval in a specified order or to set a limit on the number of tuples to be used in computing the answer. These features were left out of SODA because they do not have a natural interpretation in purely set-theoretic terms. On the other hand, SODA has more powerful counting, maximizing, and minimizing operators. In SODA, these can operate on sets of tuples defined by arbitrary subqueries; in ALPHA, they are much more restricted.

There are important differences in the syntax of the languages as well. SODA is, in fact, a data sublanguage of LISP, and thus it shares LISP's highly parenthesized syntax. While in some ways this makes SODA queries more difficult for people to read, it greatly facilitates the generation of SODA queries by other programs, a primary requirement for use in the LADDER system.

Finally, the syntax of SODA has been designed to facilitate translation of natural-language questions into formal queries. This has resulted in departures from typical relational-calculus syntax, particularly in quantifier expressions. In most "English-like" formal languages, English words are simply tacked onto a semantic structure

that bears little relation to English. In SODA, the correspondence of the symbols of the language to English words is of minor importance and is basically just a mnemonic device. What is important is that the semantics of several of the constructs of the language have been designed to correspond to the semantics of certain types of English phrases. For an illustration of this point see the discussion of quantifier expressions at the end of Section IV.B.

Appendix B

AN EXPERIMENTAL FRENCH-LANGUAGE LADDER

Appendix B

AN EXPERIMENTAL FRENCH-LANGUAGE LADDER

To help assess the transportability of the LADDER system, and to emphasize the potential use of LADDER for allowing different groups to access a common data base in their own terms, we have created a French-language version of LADDER. This task was performed by a native French speaker who had a rudimentary knowledge of INTERLISP and no knowledge of LADDER. In one man-month he was able to convert the full range of data base queries accepted by the English LADDER to French. The resulting system, called FLADDER, is available at SRI for use via the ARPANET. The full range of LADDER's user-oriented features, including spelling correction, synonym and paraphrase definition, and feedback to the user of its interpretation of his questions (in French) are included in this system.

Examples of inputs that FLADDER can deal with are:

Quand le Fox doit-il appareiller?
Quand le sonar du Kennedy sera-t-il repare
Combien de temps faut-il a l'Aspro pour rejoindre le Knox?
Quel est le destroyer le plus proche de Naples
A quelle classe le bateau Sovietique le Minsk appartient-il
A quel convoi le batiment Americain Biddle est-il affecte?
Quelles sont les marchandises transportees par le Taru?
Quel est le temps de croisiere normal du Dale de Gibraltar a Norfolk
Quel est le prochain port d'escale du Adams
Pourquoi l'America n'est pas a l'etat de preparation C1
Quel est le batiment le plus rapide
Quels sont les navires au nord de l'equateur
Qui naviguent a moins de 12 noeuds
Quels sont les bateaux naviguant sous pavillon Britannique
Existe-t-il des bateaux qui font partie de ce convoi
Dis moi quels sont la longueur et le tirant d'eau du Knox!
Quel est la portee maximum de l'engin Polaris
Est ce qu'il existe des navires dont la vitesse est superieure
a celle de l'Aspro?
Quel est le nom de l'officier commandant le Sunfish
Quelles sont les unites embarquees sur le Kitty Hawk
Quels navires sont a moins de 1 jour de route suivant une orthodromie

de ma position actuelle?
Calcule moi le chemin le plus court entre Le Havre et Naples
Quelle est la route suivant la loxodromie de Oslo a Luanda
A quelle distance de New York se trouve le Saratoga
A combien de milles de Capetown est le Kennedy
Quel est le temps de traversee entre New York et Le Havre pour l'Arctic?
Quelle est la duree de croisiere du Pecos depuis Rotterdam
jusqu'a Baltimore?
Quand le Rathburne arrive-t-il a Newport?
Ou va le Gridley?
Vers ou se dirige le Robison
Ou est actuellement le Hoel?

REFERENCES

1. G. G. Hendrix, et al., "Developing a Natural Language Interface to Complex Data," ACM Transactions on Database Systems, Vol.3, No. 2, pp. 105-147, (June 1978).

2. E. D. Sacerdoti, "Language Access to Distributed Data with Error Recovery," Proceedings of the Fifth International Joint Conference on Artificial Intelligence, pp. 196-202, Cambridge, Massachusetts (August 1977).

3. E. D. Sacerdoti, ed., "Mechanical Intelligence: Research and Applications," Final Technical Report, Covering the Period 12 April 1976 through 9 October 1977, SRI International, Menlo Park, California (December 1977).

4. G. G. Hendrix, "The LIFER Manual: A Guide to Building Practical Natural Language Interfaces," SRI Artificial Intelligence Center Technical Note 138, Stanford Research Institute, Menlo Park, California (February 1977).

5. G. G. Hendrix, "Human Engineering for Applied Natural Language Processing," Proc. 5th International Joint Conference on Artificial Intelligence, Cambridge, Massachussetts (August 1977).

6. D. Sagalowicz, "IDA: An Intelligent Data Access Program," Proceedings of the Third International Conference on Very Large Data Bases, Tokyo, Japan (October 1977).

7. P. Morris and D. Sagalowicz, "Managing Network Access to a Distributed Data Base," Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California (May 1977).

8. Computer Corporation of America, "Datacomputer Version 1 User Manual," Cambridge, Massachussetts (August 1975).

9. DEC System 20, DBMS Programmer's Procedures, Manual No. DEC-20-APPMB-A-D.

10. W. Teitelman, "INTERLISP Reference Manual," Xerox Palo Alto Research Center, Palo Alto, California (December 1975).

11. Naval Electronics Laboratory Center, "The Relational Model for the Blue File Data Base (Revised)," Project Scientist: Garrison Brown; San Diego, California (November 1976).

12. Naval Electronics Laboratory Center, "A Relational Model for an At Sea Commander's Tactical Data Base," Project Scientist: Garrison Brown, San Diego, California (October 1976).

13. R. Bisbey II and D. Hollingworth, "Situation Display: A Command and Control Graphics Application," Research Report, Information Sciences Institute, Marina Del Rey, California (to appear).

14. CODASYL Data Base Task Group, April 1971 Report (ACM, New York, 1971).

15. H. G. Miller, R. L. Hershman, and R. T. Kelly, "Performance of a Natural Language Query System in a Simulated Command Control Environment," task report, Code 832, Naval Ocean Systems Center, San Diego, CA (May 1978).

16. S. J. Kaplan, "Cooperative Responses from a Natural Language Data Base Query System: Preliminary Report," Technical Report, Deptartment of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania (November 1977).

17. E. Wong, "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases," Technical Report CCA-77-03, Computer Corporation of America, Cambridge, Massachusetts (March 1977).

18. "A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 1," Technical Report CCA-77-06, Computer Corporation of America, Cambridge, Massachusetts (July 1977).

19. "A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 2," Technical Report CCA-78-03, Computer Corporation of America, Cambridge, Massachusetts (January 1978).

20. C. L. Sinder, "A Computational Model of Co-reference Comprehension in English," Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts (May 1979).

21. G. G. Hendrix, "Encoding Knowledge in Partitioned Networks," in Associative Networks - The Representation and Use of Knowledge in Computers, N. V. Findler (ed.) pp. 51-92, Academic Press, New York (1979).

22. D. G. Bobrow and T. Winograd, "An overview of KRL, A Knowledge Representation Language," Cognitive Science, Vol.1, No. 1, pp. 3-46 (January 1977).

23. M. Stefik, "An Examination of a Frame-Structured Representation System," Stanford Heuristic Programming Project Memo HPP-78-13, Computer Science Department, Stanford University, Stanford, California (September 1978).

24. E. F. Codd, "A Data Base Sublanguage Founded on the Relational Calculus," Proceedings of the 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, available from ACM.

25. C. J. Date, An Introduction to Data Base Systems, pp. 63-82 (Addison-Wesley Publishing Company, Reading, Massachusetts, 1975).