Copy 10 of 65 copies
AD-E500 578

(2)

AD A118397

IDA PAPER P-1637

C³I DATA BASE AND NETWORKING ANALYSIS

T. C. Bartee
O. P. Buneman

April 1982

Prepared for
Office of the Assistant Secretary of Defense
(Communications, Command, Control and Intelligence)

DTIC FILE COPY

DTIC
ELECTE
AUG 19 1982
S D
E

INSTITUTE FOR DEFENSE ANALYSES
SCIENCE AND TECHNOLOGY DIVISION

82 08 11 109

The work reported in this document was conducted under Contract MDA 903 79 C 0320 for the Department of Defense. The publication of this IDA Paper does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that agency.

Approved for public release; distribution unlimited.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A117 397	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) C^3I Data Base and Networking Analysis	5. TYPE OF REPORT & PERIOD COVERED Final--Oct. 1980 - Nov. 1981
	6. PERFORMING ORG. REPORT NUMBER IDA PAPER P-1637
7. AUTHOR(s) T.C. Bartee, O.P. Buneman	8. CONTRACT OR GRANT NUMBER(s) MDA 903 79 C 0320
9. PERFORMING ORGANIZATION NAME AND ADDRESS Institute for Defense Analyses 1801 N. Beauregard Street Alexandria, Virginia 22311	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Task Order D-32
11. CONTROLLING OFFICE NAME AND ADDRESS Director, Information Systems OUSDRE (C^3I), The Pentagon Washington, D.C. 20301	12. REPORT DATE April 1982
	13. NUMBER OF PAGES 86
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

None

18. SUPPLEMENTARY NOTES

N/A

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

data bases, query languages, network languages, user interfaces, software standards, Ada, network protocols, network topology

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DoD networks support a large number of geographically dispersed data bases maintained under heterogeneous software environments. There is an urgent need to provide uniform and simple user access to these data bases, especially for Command and Control systems. This report studies the problems of implementing a network query language and the design of good user interfaces, especially for structurally complex data bases such as the Electronic Warfare Information System (EWIS). Software standards for future

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (continued)

data base development are also discussed, especially the problem of adapting Ada to interface to existing data base systems. Further development of data base interfaces is required, and it is suggested that a testbed be set up to support a selection of C^3I data bases and to support this development.

The geographic configurations and survivability requirements of DoD networks leads to overall configuration needs that are different from commercial networks. An analysis shows that distributed switches are preferable to a few large switches for a defense data network. Protocols and frontends are also discussed and a need for immediate action in these areas is established.

IDA PAPER P-1637

C^{3}I DATA BASE AND NETWORKING ANALYSIS

T. C. Bartee
O. P. Buneman

April 1982

DTIC COPY INSPECTED 2

Accession For
NTIS GRA&I X
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A

INSTITUTE FOR DEFENSE ANALYSES
SCIENCE AND TECHNOLOGY DIVISION
1801 N. Beauregard Street
Alexandria, Virginia 22311

Contract MDA 903 79 C 0320
Task Order D-32

ABSTRACT

DoD networks support a large number of geographically dispersed data bases maintained under heterogeneous software environments. There is an urgent need to provide uniform and simple user access to these data bases, especially for Command and Control systems. This report studies the problems of implementing a network query language and the design of good user interfaces, especially for structurally complex data bases such as the Electronic Warfare Information Systems (EWIS). Software standards for future data base development are also discussed, especially the problem of adapting Ada to interface to existing data base systems. Further development of data base interfaces is required, and it is suggested that a testbed be set up to support a selection of C^3I data bases and to support this development.

The geographic configurations and survivability requirements of DoD networks leads to overall configuration needs that are different to overall commercial networks. An analysis shows that distributed switches are preferable to a few large switches for a defense data network. Protocols and front ends are also discussed and a need for immediate action in these areas is established.

CONTENTS

	Abstract	iii
	Contents	v
	Abbreviations	vii
I.	INTRODUCTION	1
	A. Background Material	1
	B. Overview of Study Areas	2
	C. Recommendations	11
	D. Technical Content	11
II.	THE MULTI-LANGUAGE PROBLEM	13
	A. Introduction	13
	B. End Users	13
	C. Classification of Data Bases	16
	D. The EWIS Data Base	17
	E. Some EWIS Structures	20
	F. Limitations on the Codasyl Data Model	23
	1. Joint Keys	23
	2. Multiple Keys	23
	G. Network Architectural Structures	24
III.	QUERY LANGUAGES	29
	A. General Considerations	29
	B. Interactive Languages	29
	C. How Powerful Should a Query Language Be?	32
	D. The Syntax of a Query Language	36
	E. Query-by-Example	39
	F. Advanced Features of Query Languages	41
	1. Spelling Correction and "Escape Completion"	41
	2. Informative Failures	42
	3. Automatic Schema Navigation	43
	G. Protocol Development	43
IV.	IMPLEMENTATION STRATEGIES	47
	A. General Considerations	47
	B. Local Versus Remote Translation	50
	C. Implementation Techniques	52
	D. Front Ends	58
V.	PROGRAMMING LANGUAGE--DATA BASE INTERFACES	63
	A. General	63
	B. The Representation Problem	63

PRECEDING PAGE BLANK-NOT FILMED

C. High-Level Languages With a Data Base Extension 65
D. Interfacing Existing DBMS to High-Level Languages 67

VI. PROGRESS REPORTS ON R&D SYSTEMS 73

A. General Findings 73
B. ADAPT 73
C. Adaplex and Multibase 77
D. Functional Query Language (FQL) 79
E. Mapping Relational to Codasyl Queries 80

References 83

FIGURES

1. AUTODIN II switches 4
2. General architecture of ARPANET 5
3a. End-to-end encryption using PLIs 7
3b. PLI layout 7
4. Multi-level secure switches used in a packet system 8
5. Line encryption when end-to-end encryption is used 9
6. Example of the multi-language problem, DCA systems 14
7. EWIS data base structure 18
8. A simplified fragment of the EWIS schema 20
9. A recursive structure in the EWIS data base 22
10. Multiple keys in the EWIS schema 23
11. WIN-AUTODIN II connections 26
12. Local and remote translation organizations 51
13. A query viewed as coroutines 56
14a. Conventional front end 60
14b. Terminal support NFE 60
15. A confluency in the EWIS schema 74
16. A simplified schema 75

ABBREVIATIONS

ADAPT	ARPA Data Access and Presentation Terminal
ARPANET	Defense Advanced Research Projects Agency Computer Network
ASD (C^3I)	Assistant Secretary of Defense (Communications, Command, Control and Intelligence)
AUTODIN	Automatic Digital Network
CCTC	Command and Control Technical Center
COINS	Community On-Line Intelligence System
CONUS	Continental United States
DBMS	Data Base Management System
DCA	Defense Communications Agency
DES	Data Encryption Standard
DoD	Department of Defense
DTI	Data Transmission, Inc.
EWIS	Electronic Warfare Information System
FQL	Functional Query Language
IDA	Institute for Defense Analyses
IDS	Integrated Data Store
IMP	Interface Message Processor
IP	Internetwork Protocol
IPLI	Internet Private Line Interface
NSA	National Security Agency
NFE	Network Front End
PLI	Private Line Interface
QBE	Query-by-Example
SIP	System Interface Protocol

TCP	Transmission Control Protocol
TIP	Terminal Interface Processor
WIN	WWMCCS Intercomputer Network
WWDMS	Worldwide Data Management System
WWMCCS	Worldwide Military Command and Control System

I. INTRODUCTION

A. BACKGROUND MATERIAL

In November 1980, IDA was requested by the Office of the Director, Information Systems, ASD(C^3I), to study data base management systems in a network environment and to continue studies of protocol and network strategies which had been undertaken previously. These studies were supplemented later by two further studies, one of error problems in the missile early warning system (Ref. 1) and another the formulation of alternative systems for AUTODIN II (Ref. 2). The work in these two areas was felt to be sufficiently critical that the completion of the data base management system and general networking strategy studies was delayed until this time period.

In the study reported on herein, emphasis is placed on the language problem that currently exists in DoD and that has arisen due to the separate development of a number of data base management systems with individual query languages and intermediate languages. The networking emphasis is on general strategies for DoD computer-to-computer networks and the effects of management decisions concerning how these networks are to be structured. More detailed analysis of a particular network problem, along with specific details, can be found in the AUTODIN II alternatives report (Ref. 2). The protocol work has concentrated on the DoD standard protocols and their status, along with the use of these protocols in several computer network front ends that have been developed.

At this time, DoD networks and computer systems are in a state of flux, and it is important that DoD management actively pursue cost-effective network strategies and data base management systems standardization policies in order for future command and control systems to fulfill their requirements satisfactorily.

B. OVERVIEW OF STUDY AREAS

A steady and significant decline in the cost of computer hardware has been evident for some years. A major effect of this decline has been the growth of the minicomputer and microcomputer markets, and only recently we have started to see an impact on traditional mainframe installations typical to organizations with large data-processing needs. For example, certain ranges of the newer "minicomputers" are comparable in overall power to established mainframes and are available at a fraction of the price. In fact, it is probably fair to say that minicomputers and larger mainframes can no longer be distinguished on the basis of size or speed, but only by how they are used and the size of the support staff required to run them.

Among all kinds of computer software, data base management systems are notoriously dependent on computer architecture and operating systems, and they are usually difficult to move to a new machine. (Only operating systems appear to be worse in this respect.) Only a few attempts have been made to write data base management systems in such a way that they can be easily transferred to new machines, and this has involved writing them in a high-level language and in a general way that exploits the variety of architecture available. As a result, very few of the data base management systems currently used within C^3I enjoy any degree of transportability.

Substantial hardware conversions will take place within C^3I in the next few years with the result that increasing reliance will be placed upon software standards and high-level programming languages for compatibility. Attempts to solve the multi-language problem will therefore have to be carefully coordinated with whatever new software standards are imposed for data bases. A section of this report will therefore deal with the adaptation of data base systems to new software interfaces and to new programming languages (Ada, Ref. 3, for example).

The major recommendation of the IDA report in Ref. 4 was that the adoption of a standard data model would simplify considerably the problem of building data base interfaces. This report gives further material in this area and describes recent research and development that is relevant to this aspect of the multi-language problem. However, a major part of this

report is concerned with related technical problems that arise both from the need to design friendly and efficient user interfaces and with the strategy required to cope with the proliferation of programming languages and data base management systems.

Another section of this report deals with "programming environments" for end users of data base systems. Although the users of a simple data base query system are not necessarily full-fledged programmers, many of the facilities available to programmers are equally useful to the person who wishes to construct simple queries. The ability to edit and save queries, debugging aids, cataloging and filing systems are all important to the user whose "programs" consist only of one or two lines of text.

This report also examines techniques for implementing data base query languages in a computer network, in which part of the process of interpreting a query may take place at a site remote from the computer that maintains the data base. Finally, research and development work that is relevant to the multi-language problem within C^3I networks is examined in the final sections.

With regard to computer network architecture, packet switching systems now in widespread use or development in DoD can be placed in two general categories.

1. The switches are used in a well-defined backbone and users connect to the switches through access lines. This type of system is best represented by AUTODIN II, the backbone of which is shown in Fig. 1.
2. The switches are distributed throughout the communications systems and are generally collocated with the users computer system (Fig. 2). ARPANET is the prototype for this general architecture, but COINS, the WWMCCS Intercomputer Network (WIN), PLATFORM, and several others fall into this category.

A further general distinction concerns the means for attaining security in a packet-switched communications system. In AUTODIN II the switches are multilevel secure and are manned by personnel with appropriate clearances (the highest level used in the system).

The alternative to multilevel secure switches is end-to-end encryption. In this case a device called a Private Line Interface (PLI) is placed between

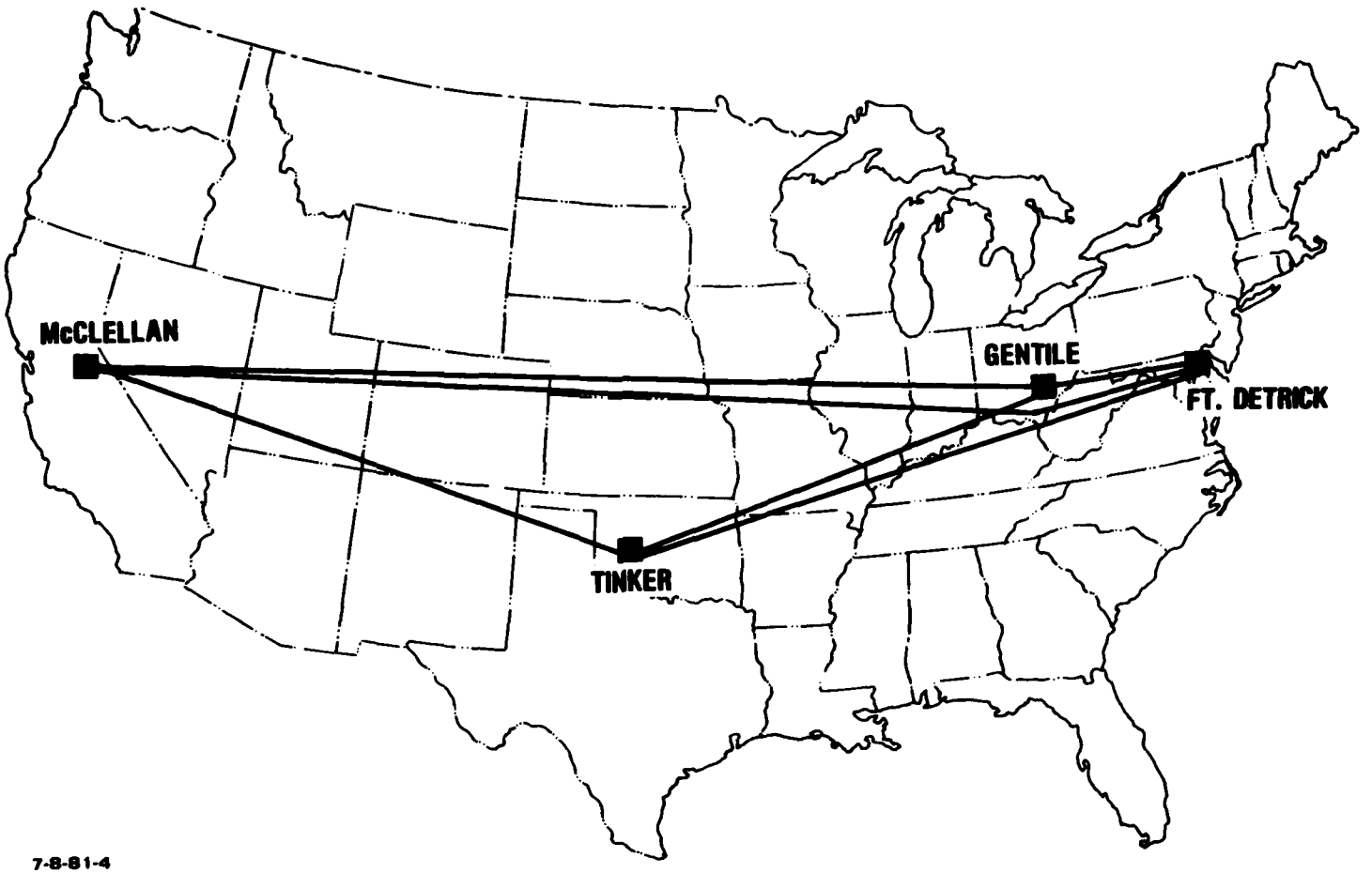


FIGURE 1. AUTODIN II switches

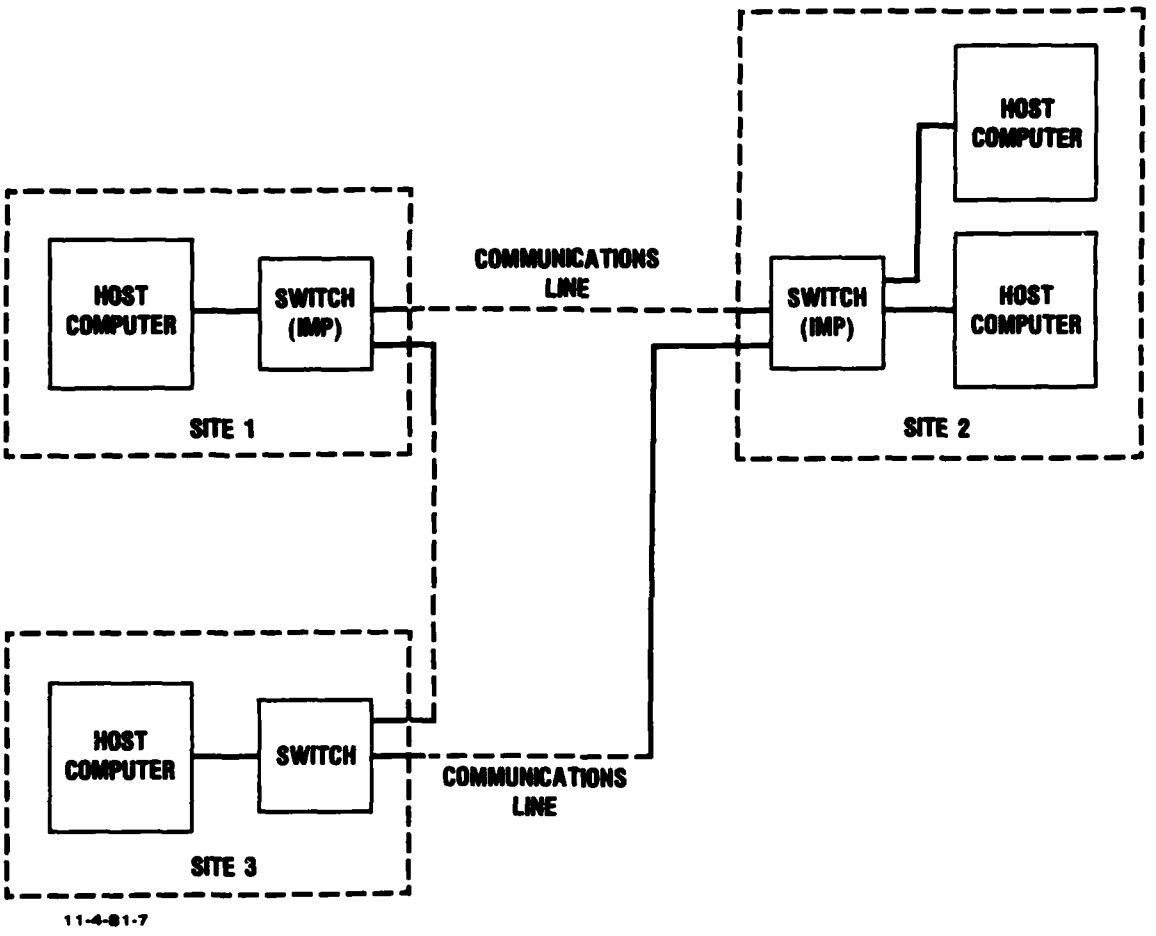


FIGURE 2. General architecture of ARPANET

the host and packet switches as shown in Fig. 3a.* Figure 3b shows the layout of the present PLI. The PLI encrypts the data in each packet, leaving the header in the packet in the clear. The switches can then route the packets but the actual data is encrypted (by NSA-supplied encryption devices in present PLIs; Date Encryption Standard (DES) may be used in some future system). As a result, when packets pass through the switches operators cannot examine the data in encrypted form nor can misrouted packets divulge classified data. Nevertheless, with one exception, these systems are run "system high" with all switches at Government (military) sites and are manned by the same cleared personnel who maintain the computer. The KG encryption devices are keyed by using the same keys for a user community and different keys for other communities; in this way the communications system is partitioned into "communities of interest."

The encryption configuration for access lines varies according to whether multilevel secure switches or end-to-end encryption are used. Figure 4 shows that an NSA encryption device is placed at each end of the access line for multilevel secure switches, and Fig. 5 shows that a PLI and two encryption devices are required for an access line if end-to-end encryption is used. (If the host is collocated with the switch the encryption devices are not required, only the PLI.) In each case the encryption device prevents monitoring traffic flow, injecting false packets, etc., along the line.

It is possible to mix the above end-to-end and multilevel secure strategies and system types but so far DoD systems have been "pure" in this regard.

There are natural advantages and disadvantages for both types of systems but the particular implementation also affects system characteristics profoundly. For example, the AUTODIN II multilevel secure switches are relatively expensive (manning costs are high) and this leads to few switches

*The devices now in use are called PLIs. Under development are Internet PLIs (IPLIs) and other devices. We will use the term PLI with the understanding any of these could be used.

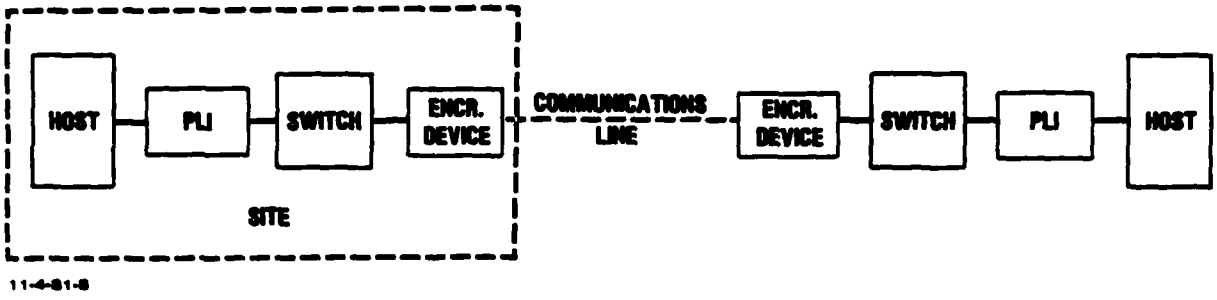


FIGURE 3a. End-to-end encryption using PLIs

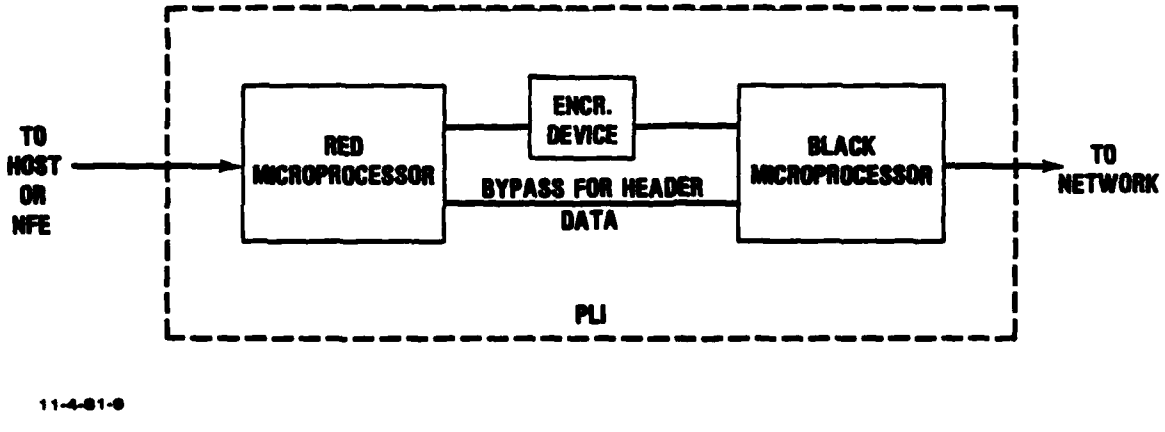


FIGURE 3b. PLI layout

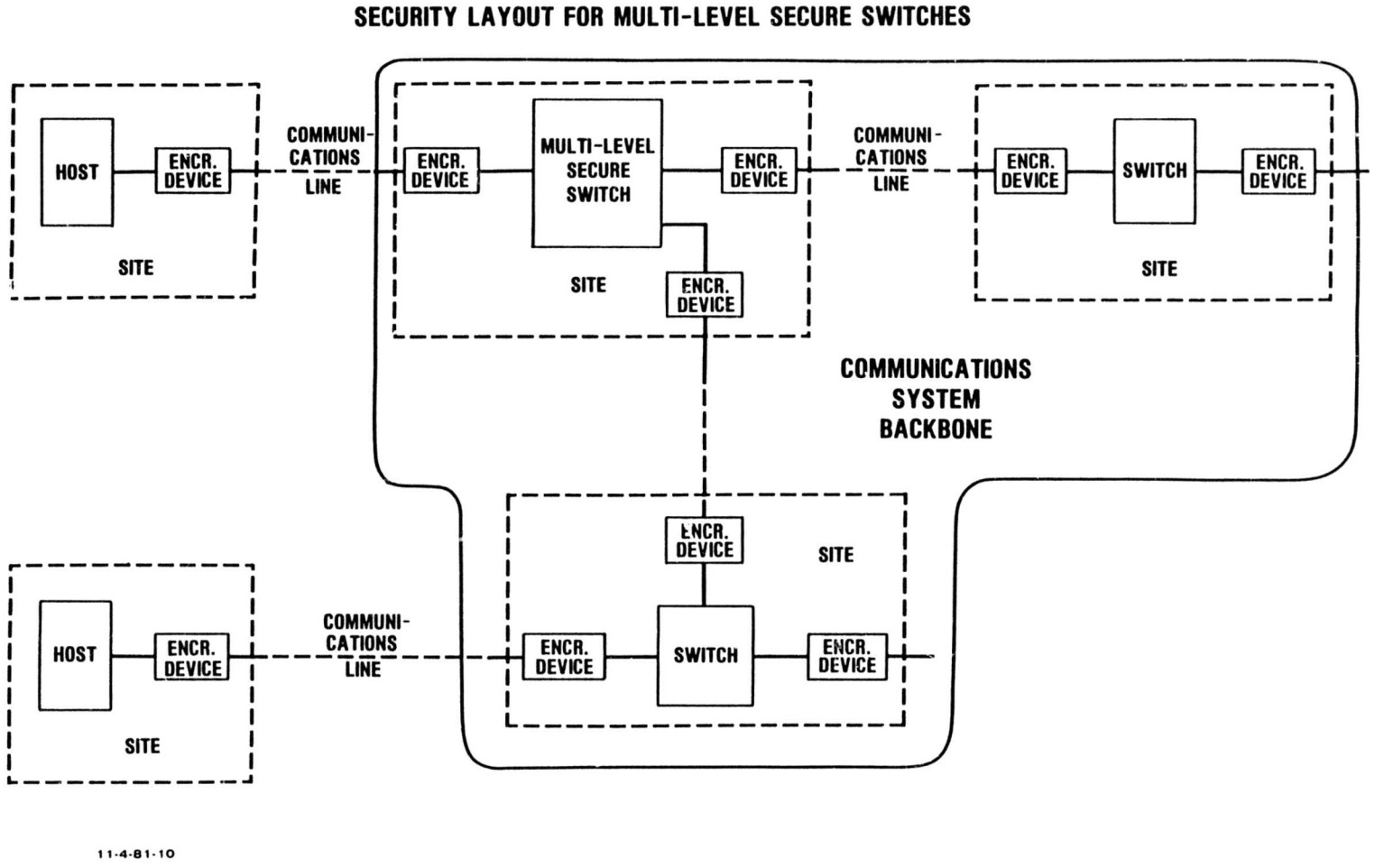


FIGURE 4. Multi-level secure switches used in a packet system

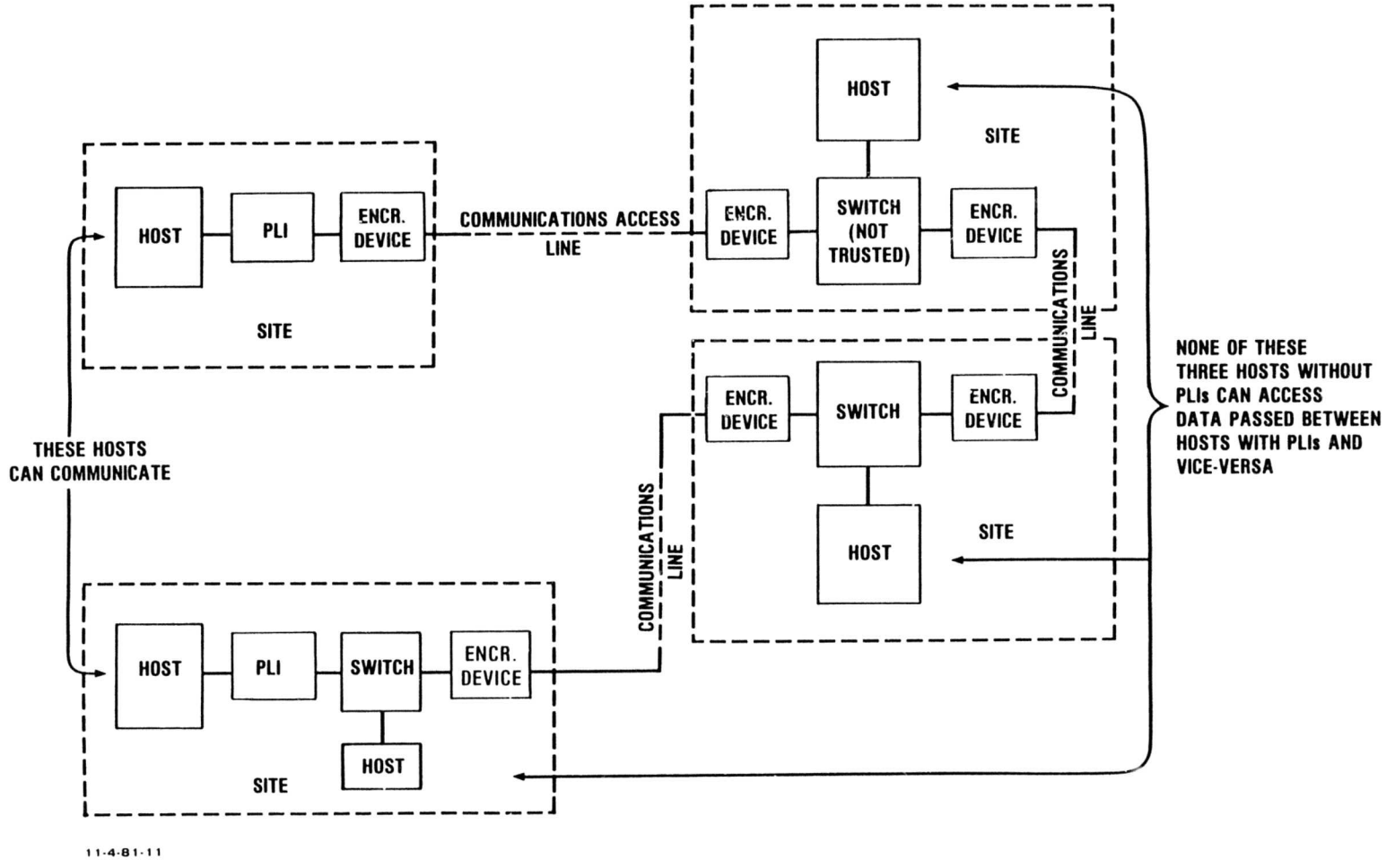


FIGURE 5. Line encryption when end-to-end encryption is used

in the backbone (four for the present design). This leads to relatively long and expensive access lines for many users and affects survivability.

The ARPANET distributed switch technology uses inexpensive switches and PLIs. This gives survivability equal to about that of the computer systems being interconnected since the switches are in the computer room with the computers, provided enough lines are used to interconnect the switches. The PLIs represent a cost not found in the multilevel secure system, however, and terminal access to the network (instead of directly to a "home" computer for each terminal) can involve PLI usage and attendant costs.

Section II gives more details on the advantages and disadvantages for these types of systems along with more technical details.

The development of front ends for DoD use is currently progressing at several places. A large PDP 11/70 front end for WWMCCS is under development by DCA at DTI*. This front end can be used on AUTODIN II after the SIP protocol has been tested and also for ARPANET-type systems (it is now being tested at the WIN facility and the ARPANET 1832 host-to-imp protocol is now in operation). The remaining front end developments are centered on microprocessor usage and these are discussed in Section III.

The front ends now being developed for DoD use the TCP/IP DoD standard protocols. The microprocessor-based front ends offload only TCP/IP along with the necessary SIP or ARPANET "network access protocol." The DCA-sponsored WWMCCS front end offloads more protocols including a virtual terminal (TELNET link) protocol, and a part of a file transfer and mail service protocol. The ability of front ends to support terminals varies and since there is no standard DoD virtual terminal protocol there is some variation here (the Recommendations discuss this).

---

*Data Transmission, Inc.

C. RECOMMENDATIONS

1. In spite of an increased awareness in DoD of the multi-language problem there appears to be a lack of serious development work aimed at solving it. Of the various projects we have reviewed in a previous report (Ref. 4), one is progressing well, another appears to have been diverted from its original purpose, and one seems to have collapsed entirely. Further applied research is needed with a specific set of data bases for these systems.

2. Standards for interfacing existing data base management systems with the emerging "strongly-typed" languages such as PASCAL and Ada are urgently required. Technical suggestions are presented in this report as to how to develop these standards.

3. A test-bed needs to be established on which user experiments with query languages and with translators can be run. An agency should be tasked to develop and implement meaningful data base studies for C^3I systems.

4. DoD standards work in the area of protocols needs to be expanded. The TCP and IP protocol standards are progressing satisfactorily, and all systems should conform to these standards. Virtual terminal, file transfer, and mail standards urgently need to be established for DoD. Further, a DoD standard host/front-end protocol is needed for front-end developments.

5. A microcomputer-based front end for medium and small DoD users needs to be developed. This front end should use the DoD standard protocols. The front end should also be usable in terminal support. Work needs to be done on front ends for local networks and gateways from local networks to global networks.

D. TECHNICAL CONTENT

The following sections of this report will survey progress in several projects designed to solve the multi-language problem. A number of criteria are discussed that may be used to evaluate query languages for power and usability. In order to obtain some realistic assessment of what is needed in the way of query languages and data models, the schema of a recently developed military information system and the range of related software is examined. This section also examines network architectures for DoD usage.

The next section reviews data base query languages from the point of view of the users. Until recently, a query language was usually developed as an addition to an existing data base management system. Recently, a few data base management systems have been developed around a query language and these are enjoying considerable commercial success. IBM's Query-By-Example is a notable case in point. Information on front ends and their status conclude this section.

The problem of implementing data base query languages is considered in the next section. At present, two strategies are employed. One involves rather complicated translation (or compilation) techniques, which usually limit the generality of the query language. Another involves the direct interpretation of a query and results in excessive I/O. A third option is described that overcomes these two defects and promises to simplify considerably the process of building query language interpreters.

The following section concentrates on building interfaces between data bases and programming languages. This problem has been sorely neglected: while extensive research efforts have been made, both in programming languages and data bases, in the representation and manipulation of data little has been done to achieve natural interfaces between these; and programming languages continue to be designed without consideration of a data base interface, even though the majority of applications of that language will involve a data base. Some efforts are now being made to remedy the latter situation: for example, a data base extension for Ada is being designed. However, there will also be a need to integrate new programming languages with existing data bases and this is a subject that should also be given some attention. Protocol status in DoD and recommendations for future work are also in this section.

The final section will review progress in a number of projects related to these topics. Of particular interest is some recent work that provides a relational view of hierarchical and network data bases and, as a result, provides the power of relational query languages against such data bases. Some data base extensions to existing programming languages are also discussed.

II. THE MULTI-LANGUAGE PROBLEM

A. INTRODUCTION

Figure 6 shows the operating systems, programming languages, data base (or file) management systems, and data bases available through CCTC (Ref. 5). The range of programming languages and general-purpose data base interfaces (this term included query languages and any other general system that provides access to some class of data bases) should be immediately evident. This example clearly shows proliferation of data base interfaces, which is a main concern of this report. In order to obtain information from one of the limited data bases, a user must be technically competent with at least one of these interfaces. The user wishing to gain access to several such data bases must understand several such interfaces and worse, usually has to understand the operating systems under which they run. Unfortunately, no standards have yet been defined for such interfaces either internally or for the user. As a result, the end user is seldom able to gain access to more than one or two data bases and this can only be achieved after a considerable investment of time in technical training.

B. END USERS

The term "end user" is frequently used to describe someone who interacts with a computer system in some way or another. Unfortunately, this term is too broad and covers an enormous range of skills and, unless it is further qualified, may be used to describe almost anyone in an organization that uses some kind of data processing. It is worth listing the ways in which a person may be considered an end user.

1. The recipient of regular computer-generated reports. Nearly everyone connected with some form of automatic data processing

CCTC ADP Software

Operating Systems:
Honeywell (3), IBM(1+), DEC(3), Univac(1)

Programming Languages:
COBOL, FORTRAN, JOVIAL, BASIC, WWDMS, assemblers(4+)

Data Base/File management systems
NIPS, WWDMS
(Honeywell FMS, IBM ISAM etc., RSX-11, Unix)

General Data Base Interfaces:
DPS/DDPS, GDR, GIPSY, GOLDS, GRTS, SOULS, GPASP
(COBOL, FORTRAN, JOVIAL, BASIC, WWDMS, NIPS)

"Data Bases"
FFIF, CAWSS, EWIS, GUARD, JCPS(3+), JAD, NICKA, NDMS,
NUCAP, NUCWA, UNITPEP(3+), USBRC, AIREP, COSTA, DASMIS,
DCF, DCPAT, EVAC, FOILS, SPF, HITS, JISR, JSPS, NATO,
NDBS, SDAS, STOCK

Notes:

1. 100+ applications programs (data retrieval, simulation, etc.)
2. Does not include numerous programs for moving data within networks, message-passing protocols, electronic mail, etc.

FIGURE 6. Example of the multi-language problem, DCA systems

uses such forms. Frequently these reports are adequate to the needs of the reader. However, it is often the case that hours are wasted in performing simple analyses (e.g., adding up a column of figures with a pocket calculator) that could easily have been performed by the program that generated the report.[1]

2. People whose only access is through reports but who may ask for an "ad hoc" report on some data. Generally, this is done by requesting applications programmers to write a special program that generates the appropriate data, but there are systems that provide the user with a range of reports through a form request--a sort of non-interactive query language. Timeliness of a report is often a problem; a figure of weeks or even months (if a new program is involved) is not uncommon.
3. Users of an interactive data base query language. Such users are in the minority at present. Their ability to extract the data they want is limited by the power of the query language and their skill at using it. The problem is that there is no standard query language, and such users may have to learn different languages for different data bases. This, of course, is a main topic of this report.
4. Applications programmers. Although there would appear to be no limit to the ability of an applications programmer to obtain the desired data, this is frequently not the case. An applications programmer, because of the amount of technical programming detail that must be mastered, is frequently "locked into" one system: a computer, programming language or operating system. Generally, applications programmers extract data for other members of an organization. A lack of understanding between the applications

[1]This problem of inappropriate reports may be more serious than is apparent from an initial appraisal. One of the authors (Buneman) has been connected with a survey of a large commercial organization with substantial data processing needs that makes extensive use of reports. A detailed questionnaire indicated that their analysts could spend up to 50% of their time in manually transferring data from one computer program to another.

programmer and the person who uses his results is a major cause of frustration and delays.

Naturally, these distinctions are not clearcut. In an organization for which there is little need for ad hoc reports or queries users will fall into the first or last of these categories. However, the distinction between an applications programmer and the user of a query system may not be sharp and it is certainly the case that a good query language can be used with enormous benefit by an applications programmer. We believe, therefore, that the development of a good general-purpose data base query system will be of general benefit, both because it will provide more analysts with the rapid access to data that they require, and because it will improve the speed with which many applications can be written.

C. CLASSIFICATION OF DATA BASES

A data base may consist of anything from a single file to a highly complex data structure maintained in secondary storage by some data base management system, but this has no bearing on how the data base may be used. Again, a crude distinction is appropriate:

1. Transaction systems. Many organizations have to cope with a large flow of transactions. The efficient processing of these transactions may pose storage problems that can be solved by the structure created by a data base management system. A bank, for example, may employ a data base management system simply to aid in processing its transactions (checks, etc.). However, the data structure created will be largely designed for efficiency rather than a natural representation of the data. In fact, the whole "data base" of a bank is usually a mixture of direct-access files, tape libraries, and microfiche records. The system is not necessarily well suited to arbitrary queries (e.g., "What was my balance at the end of last February?"). In general, this is true for transaction systems: they are characterized by frequency updates, relatively simple use of the data base management system (if one is used at all) and poor representation of "real world" data.

2. Information Systems. By contrast, the purpose of an information system is to achieve a representation of a part of the "real world". It is a system which stores some related set of facts for more or less general use, and a measure of its success is its ability to provide the answers to questions that were not anticipated at the time the data base was designed. Such systems have relatively low frequency of update but maintain rich data structures that should model complex "real world" relationships.

Again, this distinction is not sharp, but it is the case that most data bases within C^3I, and certainly those that will be accessible to any community of users, fall into the second category. They form the basis for information systems. To illustrate this, we shall review an example that has recently been developed for DCA.

D. THE EWIS DATA BASE

The Electronic Warfare Information System (EWIS) data base has been implemented under a DCA contract to provide the Joint Chiefs of Staff and the National Command Authority with general information about, and the current status of, various electronic warfare devices and surveillance systems. Our purpose in studying it here is that it is an integrated data base maintained under the WWDMS Integrated Data Store, a Codasyl-like system, and is representative of a growing number of relatively complex information bases maintained in this way.

EWIS is an excellent example of such a data base for two reasons:

1. It fully exploits Codasyl (IDS) structures. Any general-purpose query system, either an intermediate query language or an "end user" interface, must be capable of representing these structures in some way. An end user must have a precise understanding of the semantic relationships maintained by the data base even if he does this through some other data model. Figure 7 shows the schema for the EWIS data base.

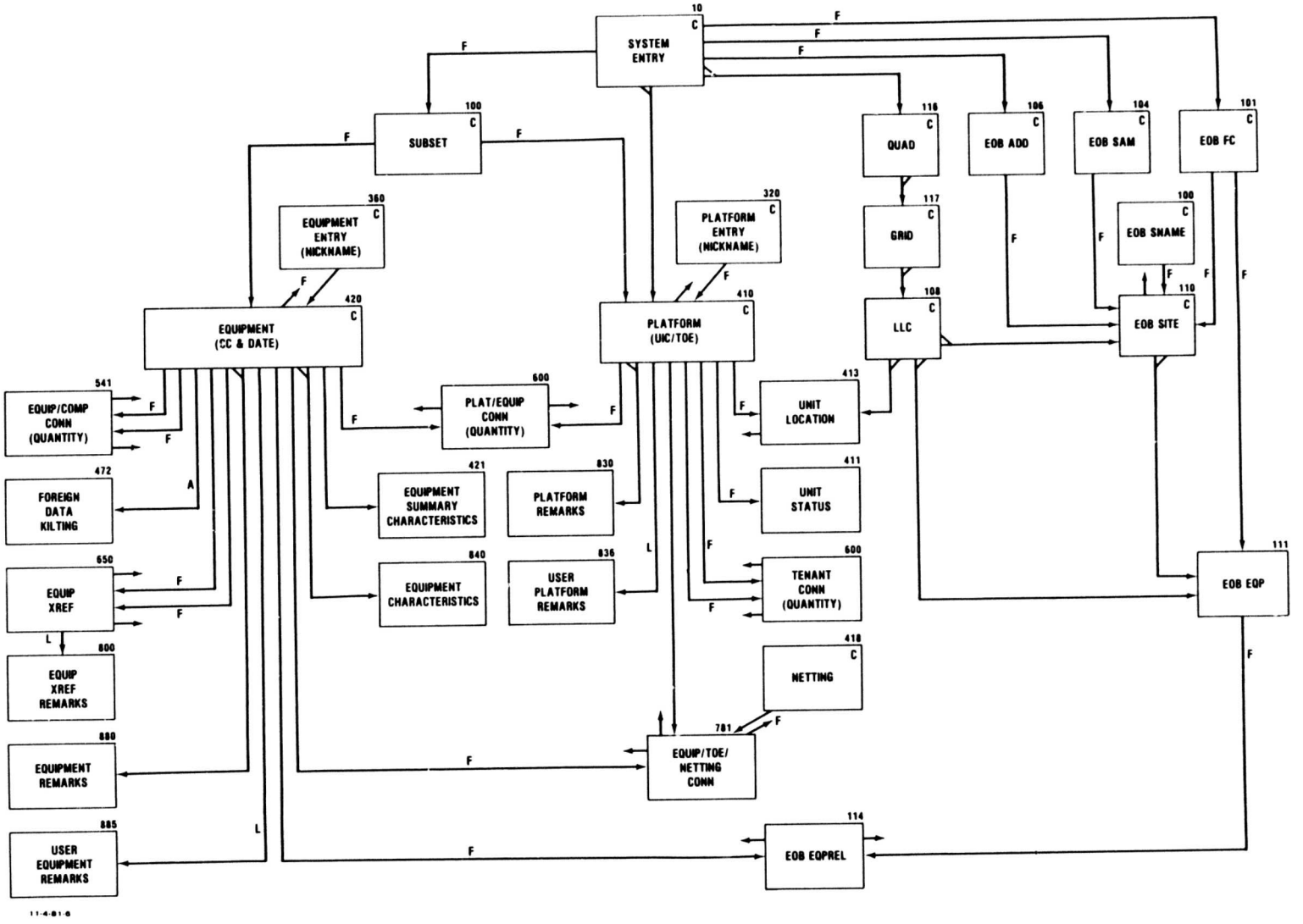


FIGURE 7. EWIS data base structure

2. While Codasyl systems are gaining increasing commercial use (many commercially available data base systems conform in some degree to the Codasyl standard--the relational systems are exceptions) there are severe limitations on the expressive power of Codasyl structures. This means that certain relationships that "naturally" obtain in the real world are difficult to represent in Codasyl structures.

The EWIS data base has been designed as a "world model" information system: that is, it has been designed to represent a collection of interrelated data in such a way that the naturally occurring (semantic) relationships are represented by physical structures in the data base. A measure of its success will be its power to support new queries even though these were not anticipated when the data base was designed. As we have suggested, information system data bases may be distinguished loosely from transaction data bases whose purpose is to support efficiently a specific function: billing, inventory control, etc. Physically, the distinction usually manifests itself in two ways: (a) Information systems are relatively static--updates are made infrequently and usually affect only a small fraction of the data base; and (b) Information systems have a relatively complicated structure--some physical structure must represent each "real-world" relationship.

The EWIS system illustrates these distinctions. Perhaps the most volatile data are geographic information about "platforms" that can move (such as ships). For the most part, minor updates on a weekly or monthly basis would be adequate to maintain the currency of the data base. As for complexity, the EWIS data base contains some eleven record types whose primary function is to store data, eighteen record types for representing "real-world" relationships, and ten records whose function is mainly to improve the efficiency of certain types of data base access. While a data base containing nearly forty record types may appear complex, there are systems (Ref. 6) that are almost an order of magnitude larger in the number of record types. Note that the complexity of the data base, as measured by the number of distinct record types, has nothing to do with its size. The required storage capacity, approximately 25 megabytes, is well within the capacity of recently developed Winchester disk systems available with certain microcomputers.

E. SOME EWIS STRUCTURES

The data base contains (among other things) information about equipments and platforms. An equipment type (EQUIP) is a device (weapon or surveillance system) that can emit a radio frequency. The EQUIP record therefore contains (among others) two fields: EQUIP-ID, the identification code for that type of equipment and FREQ, the frequency at which it can transmit. (The latter is a simplification since a range of frequencies is associated with an EQUIP record.) A platform (PLAT) is something that can support an equipment--a ship, aircraft, or land vehicle, for example. As such, the PLAT record has an identification field, PLAT-ID and a position, POSITION. Again, the latter is an oversimplification, for the representation of positions in the EWIS data base is quite complicated.

Since a platform may support an equipment, a third record is introduced. A SUPPORT record indicates how many of a given equipment type a given platform supports. The SUPPORT record therefore contains a field QTY and is linked, through sets, to the PLAT and EQUIP record. (The term for the SUPPORT record is EQUIP-PLAT-CONN in the EWIS data base.)

Figure 8 shows that the SUPPORT record class is linked to the EQUIP and PLAT record classes through the sets SE and SP, respectively. A given platform record therefore "owns" a set of support records, each of which is owned by an equipment record, one for each type of equipment supported by the platform. Similarly, from a given equipment record one may access, through the SUPPORT class, the set of platforms that supports that equipment type. The SUPPORT record class contains only one field; its main function is to represent the binary ("real world") relationship between platforms and equipment types. As such, it is often termed a "linking" or "connector" record.

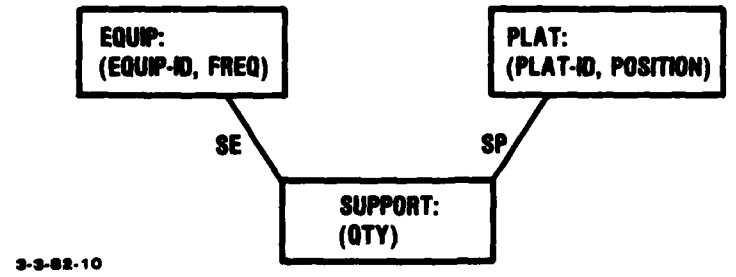


FIGURE 8. A simplified fragment of the EWIS schema

Any query system must be capable of (implicitly or explicitly) dealing with structures such as these. Consider a simple query of the form:

for EQUIP with FREQ < 27.5

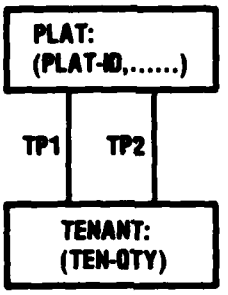
print EQUIP-ID, PLAT-ID, QTY.

This has the appearance of a simple "flat-file" or single relation query and yet it is not, for although EQUIP-ID is a field of EQUIP, the fields PLAT-ID and QTY are fields of some other record. The meaning of this query, if any, is to produce a report that is intrinsically hierarchical: at the top level, a list of EQUIP-IDs is to be displayed, and for each EQUIP-ID, a list of PLAT-ID, QTY pairs, one pair for each platform that supports the equipment. Whether the report is displayed hierarchically or is "flattened" so that EQUIP-IDs are repeated, is a matter of taste (there should be some report writer that can describe the output). The point to be made is that this kind of "traversal" of a network data base is extremely common, and any query language should support it.

There are a number of problems that arise from trying to map simple queries onto a Codasyl structure. The query above does not specify the path that is to be taken through the data base. One solution is to require a complete specification of the path (i.e., the sets SE and SP) and this must be done in any applications program that deals with Codasyl structures and with any query language (Ref. 7) that allows an arbitrary query to be constructed. However, in many cases a "smart" query language may infer the path that is to be taken, and it can certainly be done when the path is unique, as it is in our selected portion of the EWIS data base. However, reference to the full schema will reveal that there are a number of possible paths between the PLAT and EQUIP record classes. The problem of finding the correct path is discussed in the subsequent section on implementation strategies.

A previous report (Ref. 4) stated that any query language should have reasonably powerful arithmetic available to the end users. The EWIS data base contains a great deal of geographic information and information about electromagnetic waves. Any general-purpose language that operates on the EWIS data base must clearly be able to perform real arithmetic, and a library of geographic functions would be needed.

The use of recursive programming techniques in data base applications is often regarded as pointless; however, there are several places in the EWIS data base where recursion may be useful. For example, there is a record class TENANT that links the platform class PLAT to itself:



3-3-82-11

FIGURE 9. A recursive structure in the EWIS data base

The structure in Fig. 9 arises because one platform may be a "tenant" of another. An aircraft could be tenant of an aircraft carrier, a missile could be a tenant of an aircraft, etc. For each tenant relationship, an entry is made in the TENANT record class, which also has an associated quantity: the number of tenant platforms of a given type on the "owning" platform.

Suppose now that we wish to find out what frequencies could emanate from a given platform. This would involve looking at the equipments supported by the platform itself, the equipments supported by its tenants, the equipments supported by its tenants' tenants, and so on. To search these exhaustively calls for a recursive subroutine. A query listed in the EWIS users' manual (Ref. 8) indicates that a search of depth two is all that is needed; but there may be cases in which it is desirable to go at least three deep. Moreover, there are at least two similar structures elsewhere in the EWIS schema that relate the EQUIP class to itself. In each case there is a meaningful recursive query against the data base. Whether or not a query language should support recursion is again a matter of taste, especially if it involves sacrificing simplicity for power; however, recursion is undoubtedly needed in any uniform "network" query language.

The above discussion points to the need to establish a test-bed to study query languages and data organizations. At present there is no single place where samples of current data base structures or query languages can be found in DoD. Such a repository appears to be sorely needed for future work.

F. LIMITATIONS ON THE CODASYL DATA MODEL

The Codasyl data model is limited in its power to represent "real-world" relationships and the designers of the EWIS data base encountered some of these when designing the data base. These limitations should be taken into consideration when developing more sophisticated data models and when building "logical" interfaces or "user views" of Codasyl structures.

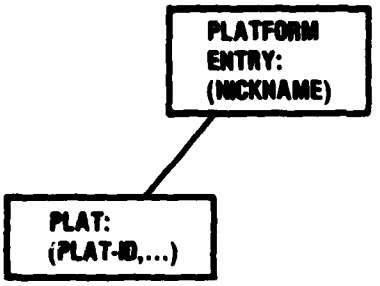
1. Joint Keys

Referring back to Fig. 9, it can be seen that there must be at most one SUPPORT record for a given PLAT and EQUIP record. If a given platform supports more than one equipment, an appropriate entry is made in the QTY field of the SUPPORT record. Thus, the PLAT and EQUIP records (or their keys) should together form a key for the SUPPORT record. However, there is no method in Codasyl for constructing multiple keys and hence repetitions of a SUPPORT record with the same pair of owning PLAT and EQUIP records are not prevented by the Codasyl DBMS; this can only be done by applications programs.

This problem almost always occurs with confluencies (diagrams like Fig. 9) in Codasyl systems. Failure to enforce the uniqueness constraint within applications programs is a frequent source of confusion and "bad" data.

2. Multiple Keys

The structure in Fig. 10 will also be found in the EWIS schema.



3-3-82-12

FIGURE 10. Multiple keys in the EWIS schema

Logically, a platform has an identification number associated with it, the PLAT-ID and a nickname, the NICKNAME. Either may be used to identify uniquely a given platform and both should therefore be keys. Codasyl, however, allows only one key per record class, and in order to accommodate both the PLAT-ID and the NICKNAME keys, a separate record class (PLATFORM ENTRY) is constructed for the sole purpose of housing the NICKNAME key. In order to access a PLAT record, given a NICKNAME, a PLATFORM ENTRY record is first found, and then the (single) PLAT record owned by that PLATFORM entry is retrieved. Note that since NICKNAMEs are unique, in the linking set each set occurrence will contain an exactly one owner and exactly one member. Again, this is a restriction that cannot be enforced by the Codasyl system itself: it is up to the applications programs to do this.

Any "logical" query system would hide this set traversal from the user, and an appropriate mapping between the user's view and the schema would have to be constructed. Should the one-to-one correspondence between the PLATFORM ENTRY and PLAT classes fail to be maintained, an end-user query would produce inexplicable (to the user) results. A similar structure is used in EWIS for identifying EQUIP records.

G. NETWORK ARCHITECTURAL STRUCTURES

The first section of this report described two general network architectures that are used in DoD systems. The first is a clearly defined backbone that is maintained separately from the computer centers and to

which computer centers connect, using access lines. This type of architecture is embodied in the AUTODIN II system. The second type of architecture is the ARPANET-like architecture that places the switches for the communications system in the computer centers alongside the computers and then interconnects these switches, using communications lines.

Each system has unique advantages and disadvantages. The advantages of the AUTODIN II-like network accrue from the ability to use a system manager who manages the communications backbone independently of the computer user, thus relieving DoD from management responsibilities if a commercial manager is used. Also, this type of network can obtain the benefits of cost reduction in high throughput leased lines, provided the number of switches is small. For example, in commercial systems a packet switch system might be established with switches in New York, Philadelphia, and Chicago. Businesses in these large metropolitan areas could then readily connect to the switches and by sharing system resources between users the operator of the system could lease high-capacity lines with an overall cost per bit of less than could be obtained if individual users leased many individual lines.

A problem arises with DoD systems when too few nodes are used, however. As an example, the AUTODIN II system with four nodes must cover the entire CONUS. Since there are only four nodes, one node must clearly be on the East Coast, one on the West Coast, one in the mid-West, and the other should be placed in the most reasonable area according to density of systems to be serviced. The AUTODIN II system has done this and has used AUTODIN I sites further to gain economies in maintenance of the switches. The problems which arise have to do with access line costs for the users and with survivability. As an example, Fig. 11 shows the connection of the WWMCCS to AUTODIN II, using the closest switch as the primary means for routing access lines (this figure uses data from the DCA user data base). Notice that the WIN sites connect almost exclusively to Gentile and Fort Detrick, leaving McClellan unused and Tinker with a single user. As a result, if Fort Detrick and Gentile were rendered inoperable, the entire high-speed digital communications system for the WWMCCS would be obliterated. This might be rectified by attempting to dual-home (connecting access lines) the WIN

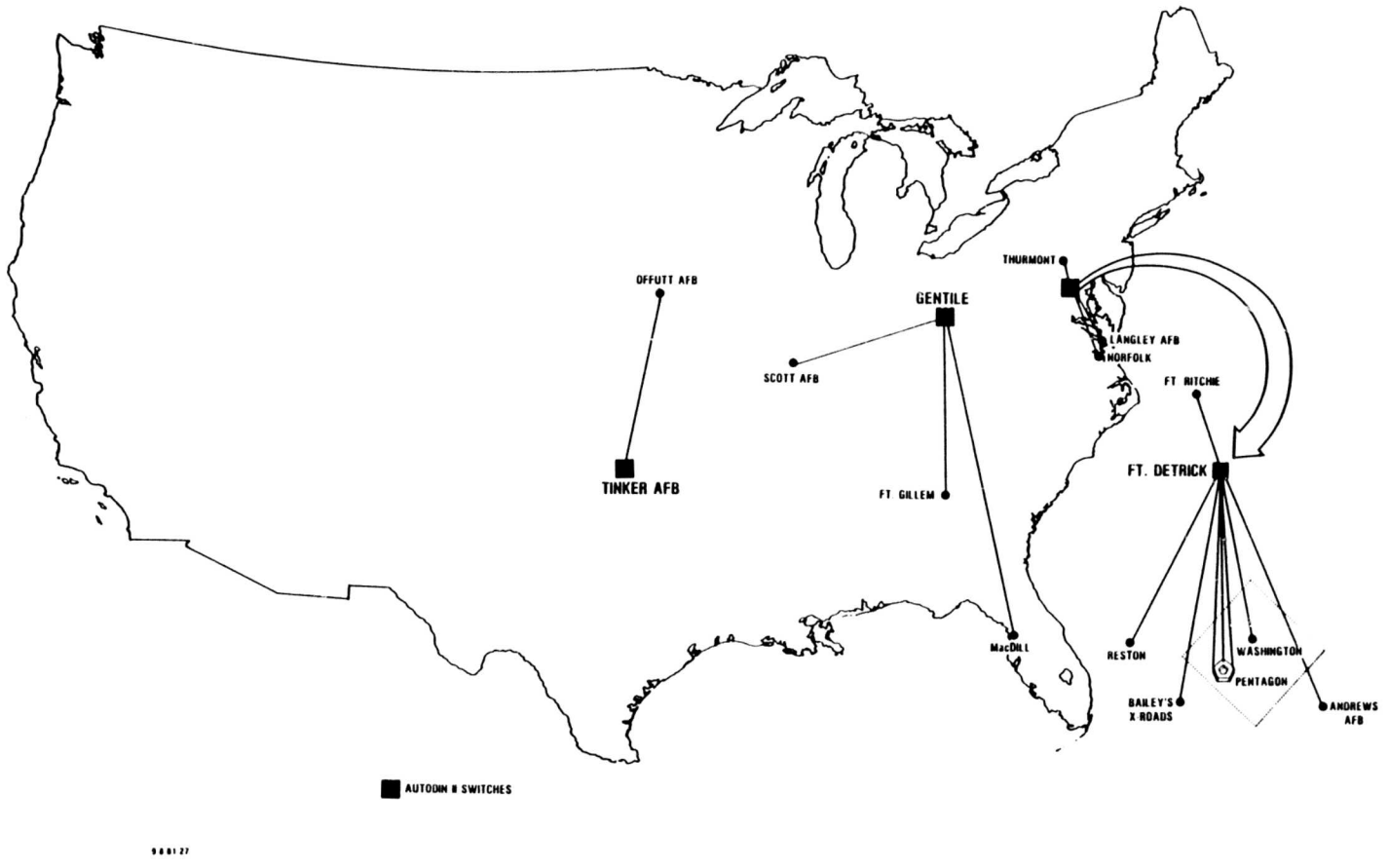


FIGURE 11. WIN-AUTODIN II connections

sites on the AUTODIN II system. Notice, however, that if the dual homing is done by choosing the second nearest AUTODIN II switch to each site, the Gentile and Fort Detrick sites will again be used in most cases with only several connections to Tinker. As a result, this does not increase substantially the survivability of the system although it would increase availability in case of switch malfunctions.

If more connections are made to Tinker or, for example, some connections are made to McClellan, survivability would be increased but at a considerable cost in system operation since the access lines would then become considerably longer. This simple fact makes it quite difficult to connect the WIN in a cost-effective manner while still getting a survivable system.

A further consideration concerns the function of the communication backbone for the user. If the system were to be connected as shown in Fig. 11, the Gentile to Fort Detrick link would carry most of the traffic generated by WWMCCS sites that were serviced by AUTODIN II lines. However, notice that in many, if not most, cases--depending upon the traffic flow--the system does nothing more than to hand a packet from one access line to another access line, and the backbone line between Gentile and Fort Detrick would not be used. If the sites are dual homed, this situation becomes worse, for in this case it is almost always possible to simply pass the packet from one access line to another line, and the backbone connection between Gentile and Fort Detrick would be seldom used.

The distributed ARPANET-like systems do not have these particular difficulties. The communications line interconnecting the switches can be placed logically and a number of lines added until the desired response times, capacity and survivability figures are met. On the other hand, the maintenance of the switches must be performed at the computer centers by the operators of the computer systems being interconnected. Fortunately, the switches themselves are simply minicomputers and are quite small and inexpensive and about as complicated as a peripheral controller. Since the technology and complexity is about that of a small piece of digital gear, the additional maintenance load on the computer centers should be quite small and personnel should already be skilled in maintaining equipment of

this sort. Further, the early statistics on down time for the new ARPANET switches is quite good so maintenance operation should be minimal. Further advantages of the distributed switches approach include the fact that the distributed design makes expansion and deployment straightforward. As an example, the ARPANET is continually being added to with sites coming into the network and new communication lines being added at a fairly frequent time scale. This means also that additional survivability and capacity can be built in readily by simply adding switches and communication lines when desired.

Another advantage of the architecture with few nodes and with a clearly defined backbone type can accrue if the system is multilevel secure. In this case, terminals can connect directly to the system using only crypto boxes at each end of the access lines since the data from the terminals can be handled in the clear inside the switches and since the backbone is protected by crypto boxes placed at the end of each communication line. For the distributed ARPANET-like system, which has no multilevel-secure switches, it is necessary to use PLIs between the terminals and the switches. The PLIs can be collocated with the terminals themselves, or if multiplexers or concentrators are used, the PLIs can be placed at the access point to the switch (IMP) to which the terminals are connected, provided the multiplexers or concentrators are multilevel-secure and that the terminals are all in the same security category. This fact increases the access cost for terminals and makes it desirable in many cases to home terminals on the host computers as opposed to accessing the network switches directly. The early ARPANET was operated in this mode, with TIPs being introduced into the ARPANET later in its time of operation. Notice that this does not limit the terminal users' access to a number of hosts since the hosts to which the terminal is connected can simply negotiate these transactions, as was the case with the early ARPANET. Also, this has the advantage of causing the terminals to be logged in and validated by the host to which they are connected instead of a host which is connected to the network--but not directly to the terminal--and which must determine the terminal's identity by information given the host from the network.

III. QUERY LANGUAGES

A. GENERAL CONSIDERATIONS

The development of a uniform data base query language for C^3I data bases should be of benefit both because analysts will be able to retrieve data in a reasonable form and because it will provide a powerful additional tool for applications programmers.

What form should such a language take and in what kind of programming environment should it be used? In earlier IDA studies (Refs. 4 and 9) reasons were presented to show the language should be based upon a simple, formal data model and it was suggested that there were two realistic possibilities for this model. Specifying the form, however, does not tell us what the language should look like, nor does it tell us what sort of programming and execution environment should be built for it. These are critical factors in the success of query languages, which are often judged by users reactions in the first few hours of use. Some further criteria will be given here that should be used to assess existing data base query systems and that should be used to constrain further development.

B. INTERACTIVE LANGUAGES

A data base query language is, by its nature, interactive. A user should be able to type in simple queries and obtain responses as fast as the system will allow. Unfortunately, according to this definition, even the slowest of batch-oriented systems may be deemed interactive provided the user employs a terminal or some other interactive medium to define the query. In order to distinguish between various query systems, we should examine the method by which a query is processed.

A data base query, or any computer program for that matter, is first translated into some intermediate form, which is then interpreted. In the common compiled languages (COBOL, FORTRAN, etc.), the intermediate code is the machine code for the machine on which the program is to run, the interpreter is the machine, and the translator is usually called the compiler. By contrast, for the simplest versions of BASIC, the intermediate code is the user program, and the interpreter is a relatively sophisticated program that executes the program a line at a time. In this case the translator is trivial, and the interpreter is a "virtual machine," represented partly in hardware and partly in software.

Other interactive languages fall somewhere between these extremes. LISP, APL, and the more sophisticated implementations of BASIC all have a translator that represents the programs as an internal data structure, which can be more efficiently interpreted. The internal representation is such that the user's program can be reconstructed, but not necessarily in exactly the same format*. If a translator (compiler) is fairly sophisticated, it can be used to do a substantial amount of checking and optimization before the program is ever run. The development of languages with rich data types provides the user with more sophisticated ways of communicating to the compiler assertions about the program. The heavily interpreted languages (APL and LISP, for example) provide only minimal syntactic checks on the program when it is translated, but can relate run-time errors in a meaningful way to the source code. Such languages often include "break packages" that allow the user to suspend program execution, explore the state of its environment, and possibly to make repairs to his code and continue execution of the program. In general, run-time errors that occur in a heavily-compiled system are much more difficult for the user to comprehend than errors that occur in the heavily-interpreted languages.

---

*The fact that an interpreter is non-trivial does not imply that the source code can be reconstructed from the internal representation. P-code is an intermediate form often used to represent compiled PASCAL programs on a microcomputer. P-code bears no direct relationship to the source code.

Although there is no reason that the implementation of a programming language should not provide both the translation-time checks of a compiled language (often achieved through data types) and the debugging aids of a heavily interpreted language, there are few cases where this has been achieved. The addition of type information to an incrementally-defined language (such as LISP and APL) presents certain conceptual and implementation problems. The addition of good debugging aids to a compiled language is often neglected for reasons of efficiency, although there are noteworthy efforts to provide PL/1 and PASCAL programming systems that may be both interpreted and compiled.

It is a matter of observation that nearly all the simple interactive programming systems, those systems that allow a program to be incrementally constructed and that provide good run-time debugging aids are the interpreted languages. It is surely the case that these systems provide the new user with a much greater degree of friendliness and encouragement than do the compiled systems. One of the reasons that these interpreted languages are so much easier to use is that the user is always "talking to" the same system. With a compiler, he usually has to understand how to talk to an editor and to the operating system; and without some knowledge of these, it is impossible to construct even the simplest program.

Data base query languages require a mixture of these techniques. They should certainly provide the user with all the benefits of an interactive language. But data bases are inherently typed, and some of that type information should be exploited when the query is defined. Thus, a query such as

select NAME, PRICE from EMPLOYEE

should yield an error as soon as it is defined. The fact that PRICE is not a field (or domain) of EMPLOYEE can be determined by reference to the data base schema alone. It should not be left for an obscure run-time error to inform the user that his query does not make sense.

This requirement for some kind of type checking, and the requirement that programs or queries should be incrementally constructed presents a problem. Suppose the query above had read

select NAME, FF from EMPLOYEE,

where FF is a user-defined function (a computer field or "virtual domain"). If FF has been previously defined, a good system should be able to find what its type is, although this is not a trivial task (Ref. 10); and, if it is incorrectly used in this query, an error should be issued when the query is defined. If FF has not yet been defined, the context in which it is required in this query must be preserved for future checks when it is defined.

However, it is impossible to do all checking in advance--even the simplest queries may generate run-time errors. These can be of two forms: (a) errors caused by limitations of the system, such as arithmetic overflow or insufficient file space; (b) errors resulting from states unanticipated by the user, such as division by zero. Both types of error are not only possible, but common in data base queries. In a programming system that is heavily translated (compiled) these errors are difficult to relate to the source code, and a casual user will have neither the time nor patience to invest in understanding what the error messages mean and how his program generated them.

These remarks indicate that an ideal data base query system should be interpreted, and that the system should be able to relate run-time errors to the user's source code. However, it should also be possible to do some checking when the user defines a query or procedure. Unfortunately, data base management systems are often designed for an environment of compiled applications programs and there are occasionally problems in building such interpreters. These are not insuperable and are discussed in a later section on implementation strategies.

C. HOW POWERFUL SHOULD A QUERY LANGAUAGE BE?

A mistake that is frequently made is to confuse the complexity of a language with its power. If the power of a language refers to its ability to control the hardware on which it is run, this confusion may be justified, for present-day hardware is inherently complicated. However, if power means the ability to provide an abstract description of a computational process, there is no reason that this should entail a language with complicated

syntax or constructs. LISP (Ref. 11) and PROLOG (Ref. 12) may not be the ideal languages for end-users in the C^3I community, but they are among the most powerful (in the second sense of the word) and we know of no languages with a simpler syntax.

Data base access languages may be broadly divided into those that are powerful but difficult to learn, and those that are limited but easy to learn. Into the former category we may place the languages associated with data base management systems such as WWDMS and DBMS20; in the latter we place the English-like query languages associated with the relational systems such as SEQUEL, Query-by-Example (QBE), and perhaps the user languages associated with ADAPT and MULTIBASE. For example, the simplest SEQUEL query such as:

select NAME, SALARY from EMPLOYEE

will involve several lines of code if stated in WWDMS or DBMS20*.

The "easy-to-learn" languages have limitations which are often encountered sooner than their designers would, perhaps, have predicted. The ability of the language to interpret simple arithmetic is either absent or laughably obscure. For example,

select unique 5+5 from EMPLOYEE

is to our knowledge the simplest way of adding 5 and 5 in SEQUEL. Briefly, what has happened here is that "5+5" is not, on its own, a valid expression in SEQUEL. However, arithmetic expressions are permitted in as "constant" domains of a relation (in this case EMPLOYEE). Unfortunately, this produces a 5+5 for each tuple in the EMPLOYEE relation and repeated values must be eliminated by adding the keyword "unique." QBE behaves in a somewhat similar fashion: the expression "5+5" must be entered as a column of a relation and the subsequent behavior of the system in evaluating a single-column, single-row relation is confusing to say the least.

*This comparison is a little unfair. It is true that the program will be several lines long, but many of those lines constitute "job-control", which is absent in an interactive system like SEQUEL.

We have already noted, in connection with the EWIS schema, that there are a number of sophisticated kinds of data access that could not be specified in a simple query language. It may be that only a very small number of users would want to perform such data access and that these problems could be left to knowledgeable applications programmers. However, from our analysis of a number of C^3I data base applications, there are two specific extensions of the "simple" query languages that will be required if these languages are to satisfy the needs of a large number of end-users.

1. Arithmetic. A glance at the EWIS schema should provide convincing evidence that not only geographic computations, but also various kinds of statistical analyses are needed. This appears to be a fairly common need. It is also desirable that users should be able to define their own arithmetic functions in much the same way that they define their own queries.
2. Text processing. Many C^3I data bases are no more than sophisticated cross-reference schemes for text. While searching the cross-reference scheme may be adequately performed by a data base query, searching and manipulating the text is usually impossible (some systems allow the user to "page" the output in a convenient fashion). Operations such as searching text for the occurrence of some word, or performing a "cut-and-paste" operation are not usually possible. We see the problem of incorporating text processing in a query language as much harder than that of incorporating arithmetic. Perhaps the best to hope for this is a good interface between a query language and a suitably structured editor.

We see no reason that a query language should not be both simple and powerful, in the sense that it should allow the user to specify an arbitrary computation. This does not mean that it should be "general purpose" in the sense that PASCAL and Ada are general purpose languages; it should be biased toward the simplification of communication with data bases and should make the expression of data base queries as simple as possible. The features by which a data base query language should be judged are, in our opinion:

1. Simplicity. Simple queries should be simple to state, and the system should provide "immediate" responses to these.

2. Support Environment. Many of the criteria that apply to the programming environments for programming languages apply to data base query languages. Good debugging aids and editors are essential.
3. Power. Provided the first two criteria are not compromised, the language should be as powerful as possible.

The requirement that the user should be confronted with a language that is initially simple to learn and that can subsequently be used for more extended applications calls for a language that has simple associated semantics, and this in turn requires a simple data model--a point that has been expressed at some length in previous reports (Refs. 4 and 9). The semantics of the language will have to be based on a set of "bulk" operators, as they are in both the relational and functional models.

It should also be noted that to err on the side of making a language overpowerful is to err on the side of caution. If a language is powerful enough (but is disliked by the majority of end-users), it should be possible to write end-user applications in that language. A query language that is too weak will leave the user community as a whole with no other recourse than to use whatever programming languages may be conveniently interfaced to the data base and this, in the context of C^3I data bases, will involve several languages.

There is one final point that is relevant to this discussion of the power of a Query Language. After COBOL, the various assemblers, PL/1 and perhaps FORTRAN, the most widely used language for data processing applications is probably APL. Certainly only APL is widely used as an interactive language. But APL was designed for scientific programming and its data base interfaces (where they exist) and its routines for the manipulation of secondary storage are, to say the least, awkward. How could an inefficient language like APL that is apparently mismatched to the needs of data processing have achieved this significance? We believe that it is because it provides a number of the features that we have advocated for query languages: it provides "instant gratification" for the casual user (it can be used as a desk calculator), it provides an excellent interactive programming environment based on the notion of workspaces, it is an extremely powerful programming tool, and, of course, it provides the necessary arithmetical

tools that constitute a large portion of data analysis. Certainly the syntax of a query language should differ from APL. The simplest expressions, the expressions that provide instant gratification, should be data base queries. Also, the advanced matrix operators provided by APL have little place as fundamental operators in a data base query language. However, the designers of future data base query languages would do well to understand what makes APL so attractive to a group of users with such widely differing technical abilities.

D. THE SYNTAX OF A QUERY LANGUAGE

In their simplest form, most query languages are remarkably similar. Consider a simple query addressed to the EWIS schema described earlier:

```
select EQUIP_ID from EQUIP with FREQ > 4.2
display EQUIP_ID of EQUIP where FREQ > 4.2
for each EQUIP such that FREQ > 4.2
   PRINT EQUIP_ID(EQUIP)
```

The first of these is SEQUEL (Ref. 13), the query language of the well-known relational data bases INGRES and SYSTEM-R (Refs. 14 and 15). The second is the query language for a commercially available Codasyl system (Ref. 16). The third is DAPLEX, which is discussed later in this paper. The syntactic similarity, especially of the first two, will be apparent immediately and only the last suggests the control structure through which the query is evaluated.

There is little doubt that, in any new user-oriented query language, the simplest queries should have the general syntactic form of the first of these examples. (A possible exception, Query-by-Example, is discussed below). The query language should be based upon some "bulk" operators, and the user should not have to define, or worry about, explicit control structures in the simplest cases. However, this leaves open some important problems.

1. What model of the data should the user have when constructing such queries?

2. What should the syntax of more complicated queries look like, especially those that involve more than one record class (relation)?
3. If a query is incorrectly or inadequately specified, how "smart" should the query language interpreter be in making automatic corrections?

It is not immediately apparent that, for a simple query, any formal data model is needed, especially since, from our examples, the forms of the first two queries (against the Codasyl and Relational model) are so similar. The situation is very different when it comes to the common problem of traversing a confluency such as the fragment of the EWIS data bases illustrated in Fig. 8. Let us look at the problem of finding the positions of all the platforms that carry equipments that can transmit at a frequency of 4.2. In the relational model we would have to represent the data base by the three relations:

EQUIP(EQUIP-ID, FREQ)
PLATFORM(PLAT-ID, POSITION)
SUPORTS(PLAT-ID, EQUIP-ID)

The SEQUEL query would then be

```
select POSITION from PLATFORM
  where PLAT-ID = SUPPORTS.PLAT-ID
    and SUPPORTS.EQUIP-ID = SUPPORTS.PLAT-ID
      and EQUIP.FREQ = 4.2.
```

In this case, the user must have a notion that SUPPORT is a separate (binary) relation. The query must traverse all three relations.

In the "smart" Codasyl query language (Refs. 16 and 17), the query is remarkably simple:

```
display POSITION with FREQ = 4.2
```

However, this simplicity is deceptive. In the first place, the data base system upon which the query language is constructed has unique field names. This means that the record classes, PLATFORM and EQUIP, that respectively contain POSITION and FREQ fields are unique. Secondly, a "down and up" rule is invoked where, if it is possible to find a path through the schema that goes through some common lower record, that path is used for resolving the query. In this case the path goes down and up through the SUPPORTS records.

This is extremely attractive for a large class of queries against Codasyl schemas, but we believe that it may prove dangerous if used as the basis for a general-purpose query system.

The query above appears to be based upon a user view that there is some class in the system with both a FREQ and POSITION field. In one sense there is, but it does not match what is in the data base. (In the data base as we have described it, there is no method of directly distinguishing two instances of the same type of equipment.) More generally, a query, in which there are two down-and-up paths or in which the fields in question cannot be connected in such a simple fashion, cannot be handled easily by such a processor. Our feeling is therefore that such a language cannot become the basis for a formal general-purpose query processor; however, as a "front end" for such a processor, it will be very attractive to a substantial group of occasional users. Note that this kind of pathfinding is impossible in relational data base systems because there is no semantic indication of the "natural" joins between relations. Various suggestions to improve the semantics of the relational data model have recently been advanced (Ref. 18).

These remarks are related to the criticisms we applied to natural language interfaces in a previous report (Ref. 4).

The kind of query that might be based upon a more explicit view of the schema would be:

POSITION of PLATFORM where some
  FREQ of EQUIP of SUPPORT = 4.2

where the path through the schema is explicitly described by mentioning record classes. Less transparently, but in keeping with the Codasyl formalism, might be

POSITION of PLATFORM where some
  FREQ of SP of SE = 4.2.

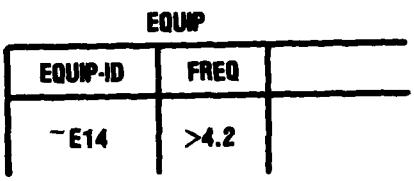
To our knowledge, there is no language with exactly this form, but it does not seem impossible to implement. In fact, the first of these is very similar to the DAPLEX (Ref. 19) representation of such a query.

Since our last report, it has become possible to phrase "set traversing" queries in ADAPT (Ref. 20). A discussion of this is included later in the report.

We should repeat here that the purpose of this section is to bring up some general remarks on the "shape" of a query language. As we have seen, the shape depends on the underlying data model and the degree of which the query processor is expected to find paths automatically through a complex schema. We should also reiterate that we feel it is more important first to construct a general-purpose language and then implement "smart" systems on top of it than to implement a variety of independent "smart" systems.

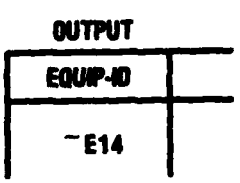
E. QUERY-BY-EXAMPLE

Query-by-Example is an interactive relational query interface developed at IBM (Ref. 21). In QBE, the user is presented with a "picture" of a relation on a screen. Queries are formed by partly filling in these relations with sample data and assertions. The process of filling in these relations is greatly helped by a screen-oriented interface that, through function keys, allows the cursor to be placed rapidly in the desired column of a relation. For example, to find the EQUIP-ID's of all equipments that transmit at a frequency of less than 4.2, the user would first fill in the EQUIP relation:



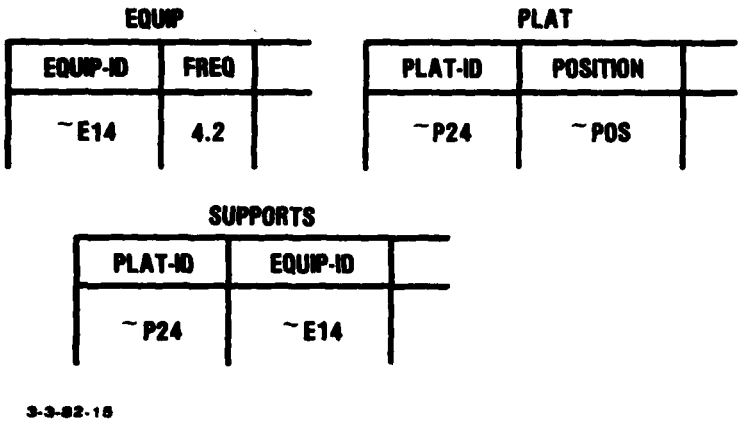
3-3-82-13

In this example the system has drawn the relation header and "skeleton," the user has placed ">4.2" in the FREQ column and the sample equipment identifier denoted by "~E14" in the EQUIP-ID column. The presence of the tilde indicates that this is an example, i.e., a variable. To obtain output, the user would construct a second relation



3-3-82-14

where the second relation mentions the variable "~E14" used in the base relation. For the more complex query used above (printing out the location of all equipment that may transmit at a frequency of 4.2 units), the screen would resemble:



3-3-82-15

Here, the user has filled in the "examples" in the three tables. That is, he has entered the variables "~E14", "~POS", "~P24" and the constant 4.2. Some further notation will be needed to indicate that a table of positions (~POS) must be displayed.

QBE has a number of advantages. At the level of the user interface it is extremely well engineered, exploiting color graphics and screen control for ease of data entry. It leaves the user in no doubt about the data model that is in use: relations are visible tables. The "programming environment," the means by which data and queries are stored and recalled, is also extremely well thought out. Whether its widespread acceptance is due to the underlying formalism or is the result of excellent human engineering has yet to be determined, for it has little competition in the second of these dimensions.

As a general-purpose data base interface it is subject to the same limitations that we have already expressed about the relational languages: even simple arithmetic is awkward, and there is little hope of extending QBE as it stands to a more general programming environment. However, anyone contemplating a user interface should certainly investigate this system.

F. ADVANCED FEATURES OF QUERY LANGUAGES

A number of additional user conveniences are also possible within query languages. Some of these have already been discussed and are implemented in commercial systems. Others are still the subject of research and several ideas have, notably, been contributed by research into natural language interfaces. We describe a few here:

1. Spelling Correction and "Escape Completion"

   A major (and greatly neglected) problem in posing an interactive data base query is accuracy of typing and spelling. Naturally, these problems arise in other situations, notably in interactive operating systems where a user may have to type in lengthy command sequences. Two techniques are gaining widespread use. One is spelling correction in which unrecognized tokens typed in by the user are matched against a dictionary of possible words. If a "close" word is found, the system enters into an interactive dialog with the user and tries to substitute this for what was typed in. The definition of "close" varies among systems, but generally two words are deemed close if they differ by an added (or deleted) character or by the transposition of adjacent letters. See (Ref. 22) for one example of such a system.

   In "escape completion" a user may specify a word by typing in sufficient initial characters in order uniquely to identify it. In order to determine whether a sufficient initial prefix has been typed, the user hits a designated key (usually the ESCAPE key). If enough has been typed, the word is automatically completed; if not, the bell is sounded, or the user is given some other indication that more input is required. For effective use, escape completion requires full-duplex communication with the user. The TENEX and TOPS-20 operating systems (Refs. 23 and 24) exploit escape completion in conjunction with a simple "finite state" grammar, not only for the command language, but also as a subsystem that can be called by user-defined programs.

In a data base query, the terms fall into three categories:

- Keywords of the query language (e.g., select, print)
- Schema terms (e.g., PLAT_ID, QTY)
- Constants and data (e.g., 'PHILADELPHIA', 3.42).

In general, the third category may be distinguished from the other two by syntax (quoted strings or numeric identifiers). If the keywords and schema terms can be placed in a common logical dictionary, then spelling correction or (if the query language has a very simple grammar) escape completion may be employed. However, these techniques are most desirable when used to specify character strings (e.g., 'PHILADELPHIA') that occur in the data base. Their implementation calls for fast string matching or binary search techniques and to our knowledge no data base management system has implemented this. And among data base query languages only the "natural language" systems make any attempt to exploit them.*

2. Informative Failure

Recent research (Ref. 25) has addressed the problem of providing better help when a query fails. The failure may happen in one of several ways, and the particular kind of failure that Kaplan investigated was a query that produces an empty response:

Q: "How many students got A's in COMPSCI101 in 1980?"

A: "None."

It is not hard to imagine the same query and response in most query languages. The answer "none" may have happened for a number of reasons:

- There were no good students in COMPSCI101 in 1980.
- There were no students in COMPSCI101 in 1980.
- COMPSCI101 was not offered in 1980.
- COMPSCI101 is not offered.

*The LADDER natural language system (Ref. 26) maintains a separate lexicon of terms in all three categories. This is space-consuming and could not be implemented in the same fashion for large data bases. See (Ref. 4) for comments on the resulting problems of data redundancy.

In formulating the query, the user (presumably) assumed that there were students in COMPSCI101 in 1980, and if any of the last three of these conditions obtained, this assumption is in error. Kaplan's system uses a simple rule to compute a hierarchy of presuppositions, and by issuing a more complex query to the data base, informs the user of the (most general) presupposition to fail.

There are, of course, other kinds of presupposition that warrant further investigation. In particular, queries that fail because the user has made a false assumption about the structure of the data base are common and result in errors, or worse, misleading responses.

The notion of informative failure in formal query languages was subsequently investigated (Ref. 27).

3. Automatic Schema Navigation

In Section III-D we described queries that could be resolved even though the data base operations (or the path through the schema) were not fully specified. This can be used in a limited fashion against a Codasyl data base since schema terms are unique and the correct path through the schema is (in general) one that follows the set linkages. The extent to which this can be done within the more sophisticated data models (Refs. 18 and 28-30) still needs to be investigated. It is also possible to use data type information to great advantage in such situations. For example, the knowledge that AVERAGE is a function that takes a sequence of numbers and produces a single number can also be used to disambiguate a query.

G. PROTOCOL DEVELOPMENT

The development of protocols in C^3I continues. The primary thrust has been toward finalizing the TCP and IP protocols which are now DoD standards, and new specification releases have been made embodying the improvements and corrections which have arisen during the testing and usage of these protocols by several DoD systems.

At the request of DCA Code 450, a series of meetings was held among representatives of DARPA, C^3I, DCA, and the Services during 1979-81 to resolve outstanding issues concerning military requirements for the DoD standard TCP and IP protocols. At the April 28-29, 1981 Protocol Standards Technical Panel meeting, final resolution of procedure, security, and TCC indicators for the DoD standard internal protocol was announced by the PSTP Working Group chaired by DCA Code 450. Subsequently, these features and several others simplifying TCP and IP operations were incorporated into revised TCP and IP specifications. The minor operating changes required of operating systems have now been made. This is part of a substantial program to maintain the DoD standard protocols.

There are a number of TCP and IP implementations in use today, some of which are available commercially. For example, the U.S. Army 18th Airborne Corps' Data Distribution Center test-bed at Fort Bragg, the DARPA/SAC Strategic C^3 reconstitution test-bed, DARPA/Norwegian Defense Research establishment integrated data/voice system, and the DARPA/UK Royal Signals and Radar Establishment all use TCP and IP. These protocols are also used commercially to support the U.S. Post Office, INTELPOS International Satellite Facsimile Communications Service.

Because of the experience DoD has accrued with TCP and IP and the success of the systems using this protocol, we recommend that the AUTODIN II system use this protocol instead of the modified early version of TCP now being used in that system. This seems to involve less risk and to be technically advantageous since the IP protocol greatly facilitates the making of gateways. This could be a very important factor in the future and, in particular, when local networks come into widespread usage.

Concerning the higher-level protocols, there is now a considerable need for a DoD standard virtual terminal and a file transfer protocol. Different segments of DoD are now supporting the development of front ends, and the questions as to which protocols to offload and what protocol to use in these front ends is now becoming an important consideration. In order to support terminals, it is judicious to use Telnet or some other virtual terminal protocol in the front end, in particular if the front end directly accesses the network (no hosts). If a number of different virtual terminal

and file transfer protocols are allowed to be constructed and used in DoD, standardization will be more difficult and costly when a DoD standard is arrived at. We therefore propose that DCA, which was given responsibility for protocol development, pursue more actively the standardization process for virtual terminals, file transfer protocols, and mail service.

IV. IMPLEMENTATION STRATEGIES

A. GENERAL CONSIDERATIONS

Various methods by which a query processing system may be implemented on a computer network are examined in this section. The previous section showed that the implementation of a query language may be accomplished by some kind of translation; a question that must now be answered is how and where this translation is to take place.

When a data management system is implemented, several methods of access to the data are also provided:

1. Through a set of subroutine calls that may be embedded in some programming language such as COBOL, FORTRAN or an assembler. This is the method used for most applications programming in automatic data processing.
2. Through a powerful programming language designed for that data base management system. The WWDMS language is a good example of this.
3. Through a user-friendly query interface that may only provide limited forms of access to the data base.

There is no absolute reason why the last two of these should be different, but in practice they often are. As we have noted above, the problem of designing a language that is both simple to use and powerful is not trivial. To our knowledge, all data base management systems contain provision for access of the first kind.

In addition to this, some form of preprocessing is nearly always required for any program that performs physical data base access. The reason is that terms appearing in the schema, such as PLATFORM or POSITION, that designate record classes or fields, are not normally directly understood by the DBMS. Instead, they must be translated into some internal code that

indicates, say, a physical offset to be used to identify the field or record class. For this reason, in a data base applications program written in some programming language, a preprocessor is invoked to translate the terms employed in the user program into the internal codes required by the DBMS. The program may then be run through the normal compiler for the language. This means that in the normal execution of any kind of user program, two translations are involved. Of course, the user usually does not have to worry about these two phases; some form of job control language is available to make the two translations look like one step. The distinction between these two translators will become important when we discuss interpretation methods for data base queries.

Within the proposed C^3I networks, the user and the data base will normally be at different host computers. That is, the user's terminal will be physically connected to a different computer to that which supports the data base. In general, there will be several computers that support data bases: we shall term these the data base hosts, and the user's home computer (which may be no more than a terminal with a certain amount of processing power) the user host.

In this configuration, the simplest method for a user to interrogate a data base is for the user host to do very little. It may either provide a terminal-to-host connection with the data base host or emulate a batch entry point. In either case, the user employs one of the three methods above for interrogating the data base; and, if the various data base hosts support different access methods (as they do within the proposed C^3I networks), the user will have to learn the individual querying or programming techniques for each data base.

The next, and considerably more complicated solution, is to provide a translator at the user host that translates from a common user language into the various programming forms required by the data base hosts. The user has now to understand only one language, and no change is required at the data base hosts, to which the user host looks like a terminal or like a batch entry point.

This is the solution to the multilanguage problem that is used by ADAPT and that has been proposed for other systems including MULTIBASE.

It has two extremely important advantages: no changes are required at the data base hosts, and all the coding needed to implement this solution can be carried out in one language on the user host. In fact, it may well be the only immediate solution that is politically or economically feasible in the short term. However, in longer-term planning for new hardware configurations and software standards, there are alternatives that should be investigated.

We shall start by examining some of the disadvantages of this method of providing a uniform interface:

Excessive Translation. In order to translate a user query into code that may be directly interpreted against the data base, three translations may be needed: the translation done at the user host, and the preprocessing and compilation performed at the data base host. If errors occur in any of these translations, they may be difficult to understand, and run-time errors, unless they are interpreted by software on the user host, are liable to prove particularly obscure. In addition, performing three translations (as opposed to direct interpretation--an option considered below) of an ad hoc query, may be a particularly inefficient method of query processing.

Multiple Schema Copies. As described in a previous report (Ref. 4) the user host translation schema requires that a schema (in some uniform data model) be kept on the user host together with the appropriate schema mappings for the remote data bases. The user host's schema is designed by the implementors of the translation scheme, and should it fail to provide an accurate representation of a remote schema, the user will be unable to issue certain queries even though these could be processed against the remote data base. Also, any restructuring in the remote data base will have to be reflected in similar changes to the user host schema. This is a standard problem of data redundancy. The duplicated schema has the advantage that it is locally available not only for query translation, but also for browsing. It has the

disadvantage that it must be kept consistent, in this case manually, with the individual data base schemas.

Reliability. The philosophy of not interfering with the individual data base hosts can mean that continuous maintenance is needed for the low-level interfaces. In the user host translation scheme, the user host looks, to a data base host, like an interactive terminal or batch entry station. Unfortunately, there are no fixed standards for interactive communication with terminals and minor changes in the protocols used by the data base host, such as a change to a prompt character or an error message format, can have disastrous consequences if these are to be recognized by the user host. It is also likely that the administrators of the data base hosts will not concern themselves with changes of this nature, assuming that the "users" are people and will understand what changes have been made. The authors do not know if this has proved a problem in ADAPT, but it is generally recognized as being a drawback of this kind of communication.

B. LOCAL VERSUS REMOTE TRANSLATION

An alternate distribution of programs is to put a different kind of processing power at the data base hosts. The lower portion of Fig. 12 shows a software configuration in which there is a standard network query language and a local processor at each data base host capable of interpreting the network query language. This may appear initially to be a more formidable task since it involves writing code for a variety of different machines and data base management systems. However, we shall suggest that this may not be as difficult as it first appears. Most machines support a common programming language (FORTRAN, for example) and there is, we shall suggest, much more uniformity among data base management systems at the low level of data access routines than at the higher level of query and programming language interfaces.

This approach, to support a uniform network query language, has been advocated in previous reports as a method that, in the long run, will reduce

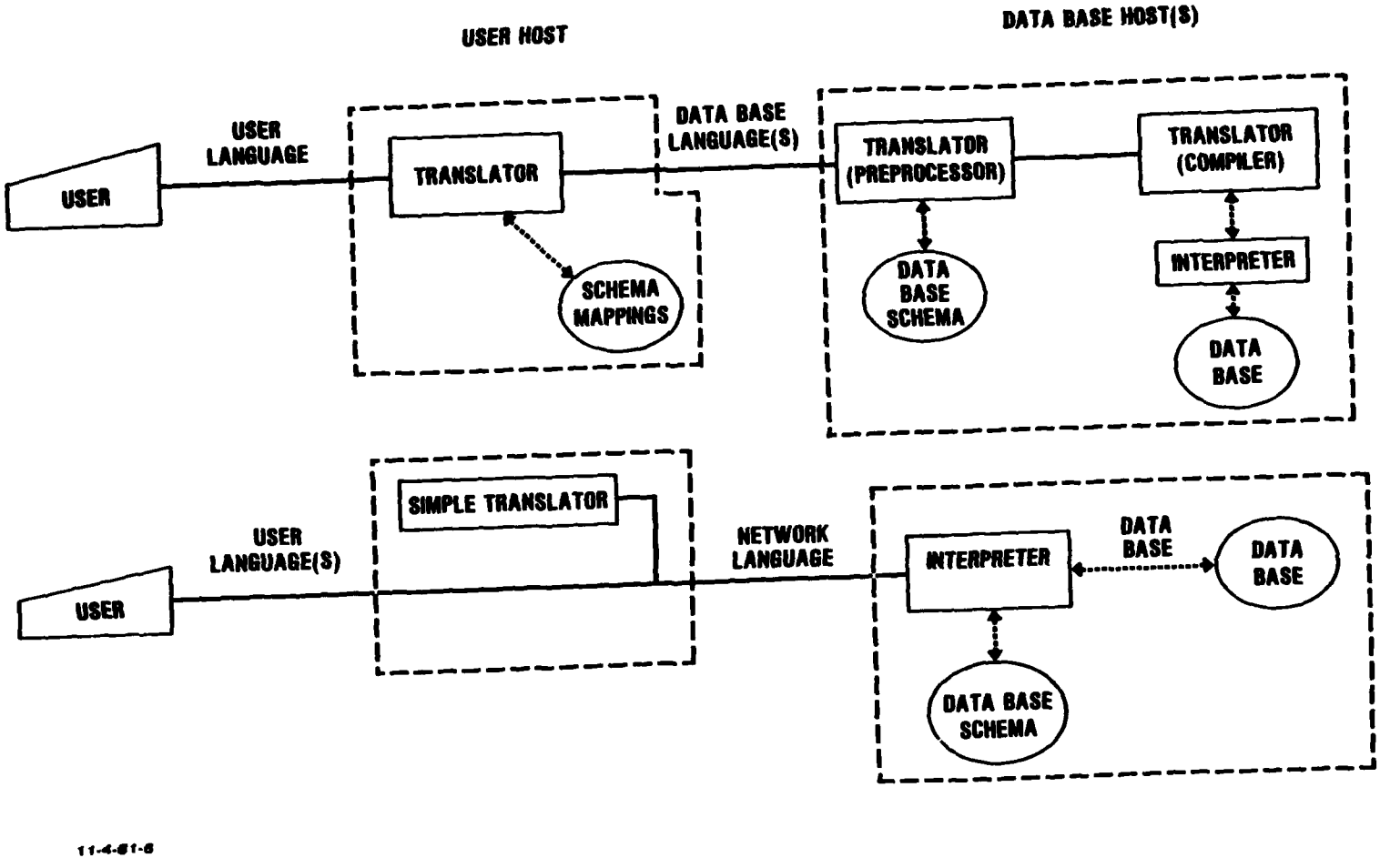


FIGURE 12. Local and remote translation organizations

the amount of software required to produce a new interface between a query or programming language and an existing data base. It also serves as a partial specification of the data base management system itself. In the following section we shall outline techniques for building data base query interpreters and indicate that this can be done with a reasonably compact implementation.

C. IMPLEMENTATION TECHNIQUES

At present there are two general methods for the implementation of high-level data base query languages. One is a translation method, in which the high-level language is translated into code in another language, usually a lower-level language, that is then compiled and run against the data base. For example, a simple query such as

```
select EQUIP-ID from EQUIP where FREQ > 4.2
```

would be translated into the obvious iterative program whose general form would be

```
1.  begin
2.    E"=GETFIRST(EQUIP);
3.    while E is not empty do
4.      begin
5.        if E.FREQ > 4.2 then
6.          PRINT(E.EQUIP-ID);
7.        E:GETNEXT(EQUIP,E)
8.      end
9.  end
```

This is not intended as an example of a specific language; it is merely written in this way to illustrate the control structure. The functions GETFIRST and GETNEXT are the "low-level" data base access routines. They are not usually defined as functions in most data base management systems, but procedures or subprograms of this general form are invariably available.

There are two points to be made about this translation. One is that it is not obvious how to do the translation in general. The program form we have just exemplified will do as a general template for the very simple query preceding it; however, more complicated queries and, should the query language allow it, other function definitions may be more difficult. In fact, nothing short of a full compiler will be adequate. A more serious difficulty arises at line 6, which defines the "output" of the query. In this case it has been (reasonably) assumed that the output is to be printed out. But, in general, the output of the select...from...where expression will be the input to another such expression, as in SEQUEL, where a more general query form is:

select ... from ... where
  X = select ... from ... where .

In this case the PRINT(...) on line 6 of the translated code needs to be replaced by an expression similar to the program itself. How is this to be done in general? And which expression should correspond to the outermost control loop, the inner or outer select ... form of the query? Although these problems may also be overcome with a sufficiently sophisticated translator (compiler), this is not in practice done, and most of the methods we have seen for solving the multilanguage problem by translation sacrifice considerable power in the query language for ease of translations.

An alternative to the translation method is what we shall call the immediate method. This is the technique adopted by many relational systems. The operators of the relational calculus, by definition, operate on existing relations to create new relations, and a query in the relational calculus is nothing more than the definition of a new relation involving simpler relations. For example, in our query above, if it is interpreted as a query against a relational data base, two operations are involved:

1. Restriction. Create a relation from the EQUIP relation containing just those tuples that satisfy FREQ > 4.2.
2. Projection. From the result of the restriction, create a relation containing just the EQUIP-IDs of the remaining tuples.

If we take these steps literally, each operation creates a separate physical file, usually in secondary storage. Although the computation involved is not great, the time taken and the demand on physical resources will be substantial. Of course, most practical relational systems "optimize" this query and do not need to create the intermediate relation between steps (1) and (2). However, more complicated queries invariably call for the creation of intermediate relations*.

There is a third method of building interpreters that overcomes many of the complexities of the translation approach and the inefficiencies of the immediate approach that we shall call the coroutine method. It is based upon a well-known technique in programming languages called "lazy evaluation" that is particularly relevant to computations that involve traversals of very long sequences, which is typically the case with data bases.

To introduce the idea of the coroutine method, consider the operations that might be involved in traversing a relation such as EQUIP at a level close to the access of physical storage. The tuples of such a relation will be specified by some kind of address in secondary storage. We can call such an address a currency pointer** (CP). A currency pointer may be a direct address in secondary storage, or it may be an index from which a direct address may be computed, much as is done for arrays. In order to generate a currency pointer for the first tuple in a relation, there will be a procedure that takes an identifier of the relation EQUIP as argument and returns a currency pointer for the first tuple in the relation. We may call this procedure FINDFIRST, and characterize it as a mathematical function,

---

*The relational data base management system INGRES makes elegant use of the UNIX operating system to "pipe" intermediate relations between the relational operators. This represents a solution akin to the "coroutine" approach, our third method of building interpreters. However, using the operating system for this purpose can introduce a number of inefficiencies and most operating systems do not have such a powerful method of multitasking.

**"Pointer" would undoubtedly be a better term than "currency pointer." We use the latter because it agrees with the somewhat accidental Codasyl terminology.

FINDFS: DBID -> CP.

DBID stands for data base identifier. A data base identifier may be any name in the data base schema. In particular, the term EQUIP is a data base identifier, and the result of applying FINDFS to this identifier is to return a currency pointer to the first tuple in the relation.

In order now to obtain successive tuples in the relation, we need a second function, FINDNEXT, that gets a currency pointer for a tuple that follows a given tuple. Again, mathematically,

FINDNS: DBID x CP -> CP.

That is, FINDNS takes a data base identifier such as EQUIP and a currency pointer for some tuple and produces the currency pointer for the next tuple (or some special value if there are no more tuples). We may think of a (DBID,CP) pair as a very simple control block that contains all the "control" information needed to find the tuples of a relation. There are then two operations we may perform on such a control block:

GET: DBID x CP -> TUPLE
NEXT: DBID x CP -> DBID x CP .

The first of these functions returns (moves into program-specified storage) the tuple referenced by the control block, and NEXT creates a control block for the next tuple in the relation. The point of representing a traversal of a relation in this way is that the control block contains all the information needed both to fund a tuple and to find the next tuple. Note that the control block is a very simple object; internally it is little more than a pair of integers. Also we have suggested that set traversal be performed by a procedure that "creates" a new control block rather than overwriting an existing one. This allows several simultaneous traversals of the same relation--a problem in Codasyl systems that rely on "data currency" for data base traversal.

The main point of the coroutine approach may now be stated. The results of any query is a control block with similar operations: GET and NEXT. We shall call these more general control blocks coroutines; they have exactly the same operations as the basic control blocks that extract data elements from the data bases. In our query

select EQUIP-ID from EQUIP where FREQ > 4.2

we may think of EQUIP as being a coroutine (control block) that produces, on demand, each tuple in the EQUIP relation. The sub-expression

EQUIP where FREQ > 4.2

is represented by a second coroutine that produces, again on demand through the GET function, successive tuples that satisfy the predicate "FREQ > 4.2".

The whole query is now represented as a coroutine that produces successive EQUIP-ID character strings. Again, it will have two operations: GET, which extracts the current character string from the "control block"; and NEXT, which creates a new "control block" that will produce the next character string. Thus, we may regard the various subexpressions of this query as coroutines or communicating "agents" that each perform a small part of the processing required to evaluate the query.

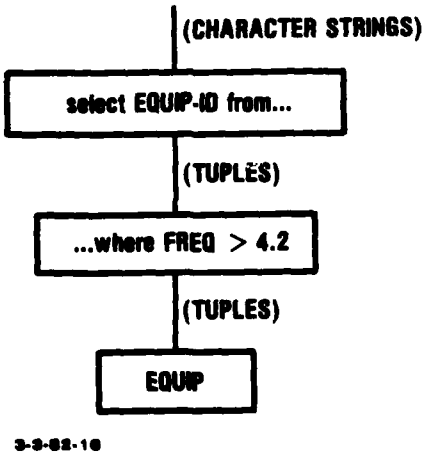


FIGURE 13. A query viewed as coroutines

A query may then be thought of as a "production line" such as is displayed (vertically) in Fig. 13. Each agent performs a transformation on the stream of objects arriving at it. However, in the coroutine approach, the system is driven by output. When a call is made to the select... coroutine to deliver a character string, it in turn calls upon the ...where coroutine to deliver a tuple, and this will repeatedly call upon the EQUIP coroutine to deliver tuples until one is found to satisfy the predicate of the where coroutine. Thus, the whole system has done enough work, and no more, to produce the first character string output. Such systems are often termed "lazy", and are used in a number of advanced programming languages.

How are these intermediate coroutines such as the ...where coroutine implemented? Again, it is nothing more than a relatively simple control block that contains:

(a) An indication of the function it performs. In this case it is a ...where coroutine, or a "filter."

(b) A reference to the predicate, in this case a data structure (or character string) representing the predicate "FREQ > 4.2."

(c) A reference to the coroutine from which it will receive its input. Here it is the EQUIP coroutine.

Now a call of GET to such a coroutine will first realize that it is a ... where coroutine. It will then perform repeated GETs and NEXTs on the EQUIP control block until the a tuple satisfying the predicate is found.

It should be noted that, for the purpose of the data base query, only a few different kinds of coroutines will be needed. However, once these coroutines have been constructed, it is a simple matter to add further coroutines that perform other computational tasks. Thus, an iteration in a conventional programming language such as

```
for I := 1 to N do
```

may also be regarded as a coroutine that (like the data base coroutine EQUIP) generates a sequence of integers from 1 to N.

In this brief exposition, we have not fully described the details of the coroutine approach. However, it should be apparent that it does not

have the space-consuming disadvantages of the immediate approach, nor does it have the preprocessing problems associated with the translation approach. The idea of efficiently implementing queries by "pipelining" is discussed in (Ref. 29), and one of the authors of this report (Buneman) has been extensively involved in an implementation of a query system that can communicate both with Codasyl and Relational data bases. This is reported in (Ref. 31). The implementation is remarkably compact, occupying only a few hundred lines of PASCAL. Although the resulting interpreter is somewhat less efficient (in query execution) than a compiled COBOL or FORTRAN program, the difference is marginal and is swamped by the perceived delays that result from I/O. And the elapsed time in executing a program is often considerably shorter because the coroutine approach usually demands less I/O than either the translation or immediate approaches.

Query interfaces similar to the coroutine method are novel but gaining increasing acceptance. The reason the techniques have not been tried earlier (they have been widely used in programming languages for some time) is probably because the usual programming languages available for data base work, FORTRAN, COBOL and the assemblers, are not well-matched to the problem. They do not allow recursion (this is needed when, for example, a FIND in one coroutine calls for a FIND from another); nor do they possess the dynamic storage allocation primitives that make the implementation particularly simple. However, implementation in FORTRAN is not impossible (one is currently in progress) and the higher-level languages such as PASCAL, PL/1, and more recently Ada, that are becoming increasingly available in C^3I installations may make this approach an attractive possibility.

D. FRONT ENDS

The network front end development in the DoD is progressing along several lines. The major NFE development is sponsored by DCA for the WWMCCS, with the major contractor being DTI. This particular front end is a high-speed front end which is currently centered about a PDP 11/70. This front end will support several terminals, a host, and has line speeds in addition of 100,000 bits/sec into the communications network. The front

end at present is scheduled to support a virtual terminal, file transfer segments, and some mail service, as well as TCP and IP (Fig. 14a). Versions of this front end are now in existence and are being tested at DCA.

The remaining front end developments are centered around microprocessors and offload less than the DCA version and have correspondingly low line rates into the communication network. However, for most users the more modest front end is desirable. A study of the DCA data base for AUTODIN II users indicates that the majority of users require only from 19.2 to 9.6 kilobits/sec, so the high-speed WWMCCS front end represents a needless expense for these users. As a result, a clear need now exists for an inexpensive multi-purpose front end for C^3I and general DoD communications.

To clarify the situation on the function of a front end, in a conventional front end development, the front end offloads the host-to-communications subnet protocol and the host-to-host protocol, and the virtual terminal protocol resides in the host itself. This is shown in Fig. 14a. Notice, however, that the host-to-communications subnet protocol is replaced by the need for a host-to-front-end protocol in the host. In order for the front end to be of service to the host, it must have a relatively straightforward host-front-end protocol in order that calculations on the host-to-host protocol and host-to-communications subnet protocol are sufficiently reduced to make the front end advantageous in terms of total calculations. The offloading of the file transfer protocol or the Telnet/virtual protocol can further increase the advantage to the host in terms of reducing the total computations required for communications transactions. Again, however, the interface and distribution of files in the system must be carefully thought through in order to minimize the host computational load in the sharing of the overall communications computations.

The decision as to which protocols to offload in addition to TCP and IP must be based upon the cost of the front end versus the savings for the host when these protocols are offloaded. This decision can be readily made by the host user, provided the costs for the offloading and different versions of the front end are known.

Our recommendation is that C^3I develop a front end which is sufficiently modular in both hardware construction and program layout and for which the

rogramming language is sufficiently universal so that a series of front ıds can be developed and so that other allied devices such as terminal andlers (Fig. 14b) and gateways can also be constructed using the program ɔdules from the original front end. The need for this is sufficiently ressing and the advantages of such a system are sufficiently great that it ɔuld appear expedient to make this development a competitive one, perhaps y tasking more than a single front end developer for the final device. his would provide incentive to the two developers so that the set of evices which is chosen would be as technologically advanced and well hought out as is possible at this time.

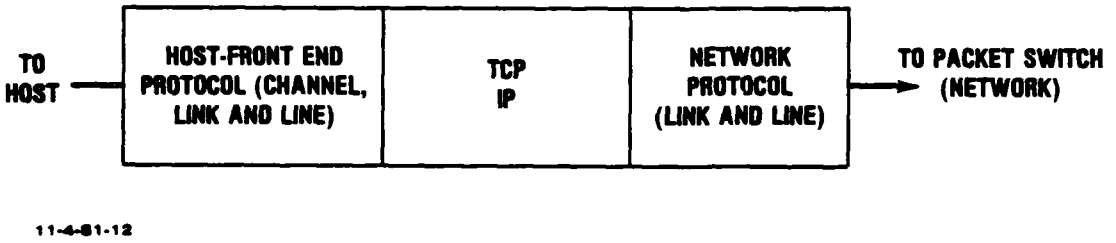


FIGURE 14a. Conventional front end

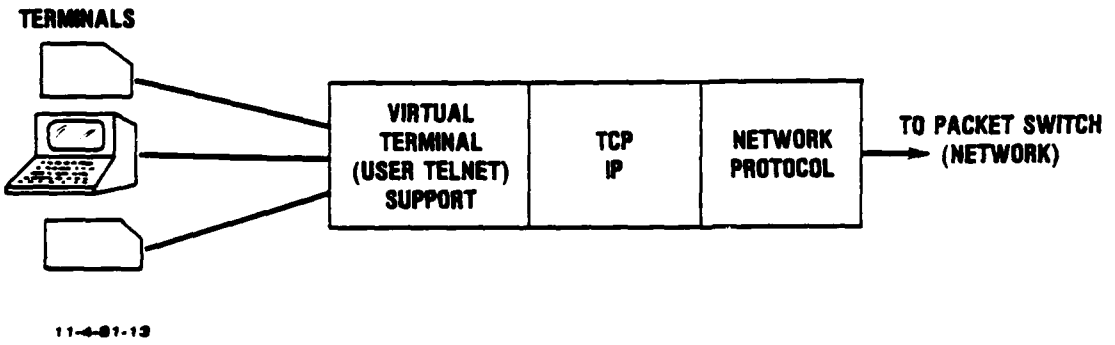


FIGURE 14b. Terminal support NFE

In addition, in order for DoD front end development to be really effective, virtual terminal (Telnet) and host-front end protocol standards are urgently needed. Different agencies are now developing different protocols and this could result in incompatible systems and prohibit the use of a successful front end development throughout C^3I.

V. PROGRAMMING LANGUAGE--DATA BASE INTERFACES

A. GENERAL

Although the main purpose of this report has been to study query language, it is also appropriate to examine some of the problems associated with building "natural" interfaces between existing data base systems and programming languages. The reason is twofold: first, the user community at large is becoming increasingly familiar with some high-level language (PASCAL, for example (Ref. 32), is one of the most widely-used languages on microcomputers). Therefore, many of our "casual users" will, in the near future, know how to program but fail to understand the data base interface. Secondly, any future implementations, in particular those recommended in this report, should be more reliable, and more rapidly completed, if they exploit a high-level language.

B. THE REPRESENTATION PROBLEM

Consider a record declaration as it might appear both in the Codasyl data definition language of a data base and in the data type declaration of a PASCAL program. The Codasyl declaration would be

```
Record name is EQUIP
    key is using EQUIP-ID.
    EQUIP-ID type is character 30.
    FREQ is float.
```

and the PASCAL declaration would be

```
Type EQUIP = record
               EQUIP-ID:  PACKED ARRAY[1..30] OF CHAR;
               FREQ:  REAL
             end.
```

Apart from the key statement in the Codasyl declaration, one might be tempted to think that the differences between the two declarations are essentially syntactic. Therefore, one could argue, it should be a relatively easy matter to embed Codasyl with PASCAL (and other similar languages). Apart from the widely-claimed advantages of using these higher-level languages, there would be two important benefits for people who write applications programs against Codasyl data bases.

(a) The problem of data currency, the problem of communicating with the data base through a fixed set of global variables, would disappear. This problem was described in some detail in a previous IDA report (Ref. 4). It is reasonable to conjecture that data currency was only introduced by the designers of Codasyl to allow a simple representation of the data base interfaces in languages such as COBOL and many of the assemblers, which are inherently nonprocedural.

(b) One of the purposes of a strongly-typed language is to permit a greater degree of compile-time checking. Another source of errors in writing data base applications in conventional programming environments is, we believe, type mismatches. For example, a Codasyl programmer may think that he has obtained a data base address (currency pointer) for a record of one type, when in fact it points to some other type. The result of performing a GET, i.e., moving data into some run-time structure, will create meaningless data. In a strongly-typed system, the currency pointers could be typed by the records to which they refer, and the compiler could check for such errors and catch them before the program runs.

Unfortunately, achieving a natural embedding of Codasyl is not as simple as it may first seem. We shall discuss these difficulties and present some solutions which, while they are not as general as they should be, may nevertheless be useful in the near future to serve the needs of applications programmers in the C^3I software environments. First, let us look briefly at attempts to build strongly-typed languages with a data base "extension."

C. HIGH-LEVEL LANGUAGES WITH A DATA BASE EXTENSION

It has often been observed that none of the recently developed high-level languages which, like PASCAL, have a rich data type system, has adequate mechanisms for representing "persistent" data (i.e., data that persists after the program has terminated). In other words, none of these languages has a data base management system that is properly integrated with its type system. An extreme example is to be found in PASCAL itself, which has sophisticated mechanisms for the representation of complex data structures at run-time, but has, according to the standard, only the sequential file as a mechanism for dealing with secondary storage and hence a very weak formalism for persistent data. This observation has prompted a number of workers to develop either data base extensions to existing languages or to design new "strongly typed" languages with an integral data base management system.

Two examples of new languages with an integral data base management system are Plain (Ref. 33) and Rigel (Ref. 34). (We are counting here only languages in the PASCAL tradition.) Systems that integrate a data base management system with an existing programming language are PASCAL-R (Ref. 35) and, more recently, Adaplex. The former provides a relational extension to PASCAL; the latter provides an interface between Ada and the Daplex language (Ref. 36) (a part of the Multibase proposal cited in previous IDA reports (Refs. 4 and 9). Of these, PASCAL-R is probably the best known and has been in use for some time. It deserves a brief description here.

Central to PASCAL-R is the type constructor relation. Like the construction array, record, etc., in PASCAL, the constructor relation builds a new data type out of more primitive types. The program can contain several

instances of a given relation. A program that modifies an EMPLOYEE file might, for example, represent the old and new sets of employees both as instances of type EMPLOYEES where EMPLOYEES is a relation data type. Also added to the language is a set of built-in procedures to perform the operations of the relational calculus: restriction, projection and join. Relations are physically represented in secondary storage, and inversion tables may be specified to ensure the efficiency of these operations on certain domains. Taken together, these additions have required substantial changes to the PASCAL compiler. The system has been used successfully in a number of applications.

It should be noted that extensions to programming languages such as this are more suitable for "applications" than for casual query. In order to generate a simple restrict-and-project query such as that in the previous chapter, a user must declare the types of the input relation(s), open the data base and write a simple iterative program to print out the results of the relational operations. Although this is not hard, and may be simplified considerably by the use of a good editor that allows an existing program that contains the correct type definitions to be modified, it is nevertheless more cumbersome than writing the same query in SEQUEL, for example. Also, as we have pointed out already, the overhead for compilation may make a "one-shot" query considerably more expensive in such an extended programming language than it would be in a general-purpose query language.

A more general criticism of this kind of programming language extension is the proliferation of control structures within the language. Viewed at a low level, the operation of the relational calculus are control structures having much the same status of constructs such as while...do or repeat... until of the host language. While one is operating purely on relations, one may use the operators of the relational calculus; but as soon as it is necessary to integrate a relation with some other structure in the programming language, such as an array, one must resort to the lower-level control structures of the host language. Stated another way, the relational operators bear a much closer affinity to the operators of LISP (Ref. 11) and APL (Ref. 37) and there is a mismatch in style between these languages and the languages such as PASCAL that contain little or no provision for such

operators. A programmer in PASCAL-R might, for example, wonder why restriction and projection cannot be used for one-dimensional arrays and for any other data type that represents a repeating group or set; there is, after all, no difficulty in understanding how such operators should work. An extremely rich set of control forms has been proposed for the language Daplex, and although the authors are unaware of the details of the Adaplex extension to Ada, there may be similar mismatches in style and brevity of programs that have a very similar semantic structure.

These last criticisms apply only to the embedding of high-level data base operators in languages (such as PASCAL) that admit only low-level control structures. The more immediate problem of exploiting the type system of these languages to aid in the writing of correct data base applications must be considered carefully, and we now turn to the more pragmatic issues of how such interfaces may be constructed in the context of available software within the C^3I community.

D. INTERFACING EXISTING DBMSs to HIGH-LEVEL LANGUAGES

Almost all data base management systems support some kind of record-at-a-time access to the data base. In the simplest case, the data base is a sequential file, and this presents little difficulty (see, for example, the FILE data type in PASCAL). More sophisticated data management systems, and this term includes everything from indexed files through hierarchical systems to Codasyl systems, deal in data base addresses or currency pointers. Even the relational systems, at a lower level than the relational operators, support this kind of access. In an indexed file, for example, the result of a search for a record with a given index is not the record itself, but an address that may be interpreted by the data base run-time routines to locate the record if it is required. All of the data bases that we have seen (other than "flat files") within the projected C^3I ambit support this kind of access. It is also likely that increasing use will be made of some strongly-typed languages of the PASCAL family simply because they now hold such a prominent position in general-purpose computing. An obvious candidate is Ada, which, even though it was not designed to satisfy data processing

requirements, will be a strong candidate for many applications simply because of its availability.

Let us examine how a simple data base traversal might appear in a modified PASCAL (assume the type declarations have been made):

```
1.  var E:  EQUIP
2.          EREF: DBREF(EQUIP);
3.  begin
4.    FINDFA(EREF);
5.    while EREF<>NIL do
6.      begin
7.        GET(E,EREF);
8.        WRITELN(E.EQUIP-ID, E.FREQ);
9.        FINDNA(EREF)
10.     end
11.   end.
```

or in similarly modified Ada:

```
1.  E:  EQUIP;
2.  EREF:  DBREF(EQUIP):=FINDFA;
3.  while not NULLREF(EREF) loop
4.    E:=GET(EREF);
5.    PUT(E.EQUIP-ID & E.FREQ & NEWLINE);
6.    EREF:=FINDNA(EREF);
7.  end loop.
```

The primary reason that these programs are not "correct" Ada or PASCAL lies in the type declaration DBREF(EQUIP). The meaning of this declaration should be clear: EQUIP has been declared as some kind of record structure and DBREF(EQUIP) is the type of data base references (addresses) for this object. In fact, DBREF(EQUIP) is the data base analog of the PASCAL referencing operator, usually denoted by ↑ EQUIP. The difficulties involved in extending the language to cope with such a data type will be discussed shortly. Let us first examine the rest of the code.

The whole program is a simple loop over all the EQUIP records. We have assumed that the data base management system provides us with two procedures for traversing a record class. In this case we have taken, as an example, the Codasyl DML functions FINDFA (Find First in Area) and FINDNA (Find Next in Area) that manipulate objects of type DBREF. In the PASCAL program, for example, the FINDFA on line 4 procures a DBREF for the first EQUIP record in the data base; the GET instantiates the EQUIP record for the given DBREF; and the FINDNA produces a DBREF for the next record in the data base. Other than the intrinsic advantages of these languages, two benefits are claimed for these idealized programs.

1. By representing the data base interface as procedures or functions that manipulate DBREFs (FINDA, FINDNA and GET) we have removed from the interface any notion of data currency. Note that this is precisely what was required in our specification of the coroutine method for implementing data base query languages. It also considerably simplifies the problem of complex, and possibly recursive, traversals of the data base, some of which were noted in connection with the EWIS schema.
2. In our idealized version of the languages, the compiler would check that the data base interface procedures or functions have been used with the correct data types. For example, on line 7 of the PASCAL program, and line 4 of the Ada program, a compiler could check that the DBREF, D, is a reference to a record of the type that E has--in this case an EQUIP record. Again, when using languages such as FORTRAN or an assembler for data base applications, an incorrectly "typed" currency pointer is a frequent cause of confusion and one that could be flagged at compile time in a strongly-typed language.

There is little difficulty in modifying a Codasyl interface so that the DML routines consist of external functions or procedures, nor should there be any problems in doing the same for other data base management systems in order to avoid data currency. A more difficult problem is presented by the desire to integrate the type system of the host language with that of the data base. To modify the PASCAL compiler so that the data

structures of say, Codasyl, could be represented would require extensive work. In particular, it would call for the creation of a new parameterized type (DBREF) and generic procedures such as FINDFA, FINDNA, and GET.

Ada has two extensions to PASCAL that, at first sight, might appear to make modifications to the compiler unnecessary. These are generic packages, a group of procedures and structures parameterized by a type, and overloaded procedures, procedures whose action is determined by the types of their parameters. Thus, it might be thought possible to create a parameterized type DBREF, and a collection of overloaded procedures such as FINDFA, FINDNA, and GET. Unfortunately, Ada does not allow types to be parameterized by other types, and the problem of parameterizing a package not just with a single type, but with a whole collection of interrelated types and names (which is what a data base is) is, as far as we are aware, beyond the ability of any programming language that has a provision for parameterized types.

It is possible to achieve a compromise in Ada. The following is a version of our original program that is legal Ada:

```
1.  E:REC_EQUIP;
2.  EREF:  REF_EQUIP:=FINDFA;
3.  while not NULLREF(D) loop
4.    E:=GET(EREF);
5.    PUT(E.EQUIP-ID & E.FREQ & NEWLINE);
6.    EREF:=FINDNA(EREF);
7.  end loop
```

The need to use parameterized types has been avoided by constructing separate, but standard, names for the types such as REC_EQUIP and REF_EQUIP. The remainder of the program is as before. What has been done to achieve this is to construct a package of types and sub-programs that represent the data base and the data manipulation routines. The package can be generated automatically from the data definition language for the data base.

It should be pointed out that what we have suggested for Ada is not a solution to the multilanguage problem. Nor is it necessarily of use in building interactive data base interfaces where type information is often

interpreted at run-time rather than compiled into the program. However, it does offer a simple and robust solution to the problem of integrating existting data bases into the Ada language and exploiting the type system of this language. Details of a simple system that presents Codasyl data bases as Ada packages are given in Ref. 38.

VI. PROGRESS REPORTS ON R&D SYSTEMS

A. GENERAL FINDINGS

This section presents summaries of various research and development projects that directly or indirectly bear upon the multilanguage problem within the C^3I community. The general impression is that progress on this problem is slow, so slow in fact, that the rapid advances in data base technology may well make some of the existing development obsolete before it is complete. (A study group, with representatives of all the existing agencies that will be involved in C^3I, could well be formed to evaluate the various development efforts and to establish whether or not they are adequate to the projected needs of the community as a whole.)

B. ADAPT

ADAPT (ARPA Data Access and Presentation Terminal) is a system that provides a uniform user interface to a number of data bases within COINS. It has been developed by Logicon Inc. and funded by NSA. The system operates on PDP-11 minicomputers running UNIX and simulates both interactive and batch interactions with data bases on a number of remote hosts. As such it falls into the "translation" category described earlier. However, ADAPT is also capable of storing data locally and provides the user with the ability to copy and reformat portions of a remote data base for subsequent query. In this, it provides the user with a local data base that can be queried through the Adapt query language though not, to our knowledge, be updated in this fashion.

In previous reports (Ref. 4 and 9) we mentioned that ADAPT may prove deficient in its ability to represent accurately certain aspects of the Codasyl model. At the time of writing these reports, this was not a problem

since none of the data bases for which ADAPT interfaces were contemplated conformed to the Codasyl standard. However, an important extension has now been incorporated into the second version of ADAPT (ADAPT II) that may largely solve this problem, and present the users with a more powerful technique for representing queries and executing them efficiently.

The problem lay in the representation of Codasyl "confluencies." This is a structure commonly used in the defining many-many relationships between two classes, and one that distinguishes Codasyl systems from other (e.g., hierarchical or "flat-file") data base management systems. An example of a confluency is in the EWIS schema and is shown in Fig. 15:

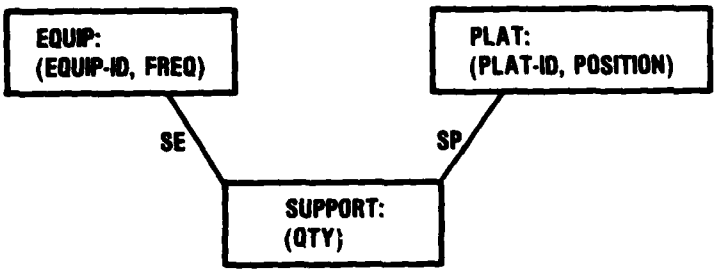


FIGURE 15. A confluency in the EWIS schema

This describes the relation between "platforms" and "equipments." A platform may support zero or more equipments of a given type, and as we have described earlier, the SUPPORT record class describes for a given platform and a given equipment type how many equipments of that type the platform supports. In ADAPT, this logical structure would be defined in the ADAPT UDL (Universal Data Language--the data definition language that describes the user's "logical" view of the data base) as:

```
SET;
    BUILD SE OWNER(EQUIP)
                MEMBER(SUPPORT, MANDATORY):
    BUILD SP OWNER(PLAT)
                MEMBER(SUPPORT, MANDATORY):
END;
```

This declaration assumes that the appropriate record declarations for EQUIP, PLAT and SUPPORT have already been given. Note that this is similar in meaning (but not in syntax) to the Codasyl DDL set declaration.

Sets in ADAPT represent "one-way" relationships. That is to say that the members of a set may be found from the owner but not vice versa. This is a departure from the Codasyl set definition and may create problems with certain queries. To show how queries are formed in ADAPT, consider Fig. 16.

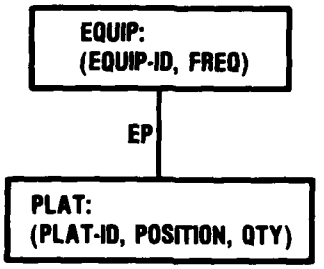


FIGURE 16. A simplified schema

Note that this schema does not represent the previous one faithfully. The same platform information (PLAT-ID, POSITION) will now have to reside in several records in the PLAT record class. To see this, consider (in Codasyl) the problem of finding all the platforms that support a given equipment. In the first diagram this was symmetrical to the problem of finding all the equipments supported by a given platform. In this diagram it is not. In this diagram the set declaration would be:

```
SET
    BUILD EP OWNER(EQUIP)
                MEMBER(PLAT, MANDATORY)
END;
```

The query to retrieve records related to equipment with a frequency of 4.2 is then:

```
L0  FIND IN EQUIP FREQ EQ 4.2
L1  FIND LABEL L0 PATH(EQ,ALL(PLAT));
    SAVREC LABEL L0 LEQUIP;
    SAVREC LABEL L1 LPLAT;
```

This creates local files LEQUIP and LPLAT, and generates a set relationship between them. The request to print out the EQUIP-ID of each equipment that transmits at this frequency together with the PLAT-ID of each platform that supports this type of equipment is then:

```
UDLC EQUIPLAT;
     MAP LEQUIP FILE EQUIP;
     MAP LPLAT FILE PLAT;
     DO RECORD LIST LEQUIP;
         DISPLAY LIST LEQUIP EQUIP-ID;
         DO RECORD LIST LPLAT;
             DISPLAY LIST LPLAT PLATID;
         END;
     END;
END;
```

Here, the set linkage is implicit in the relationship between the two DO loops*. It does not appear possible to represent cases in which more than one set connects the same pair of record classes in the same local data base. (See the section on the EWIS schema for cases in which this arises.)

To perform the "inverse" traversal, e.g., display EQUIP-IDs for all platforms at a given position, is not represented symmetrically in this case. There appears to be no simple method of moving from an "owned" record to an "owner" in the current ADAPT implementation of sets. However, ADAPT sets may have inverses which are also ADAPT sets, and this may represent a solution to the common "down-and-up" traversal of Codasyl confluencies.

---

*The authors are grateful to Dr. M.E. Soleglad of Logicon Inc., for his help in explaining this representation and query.

The addition of set relationships to the ADAPT logical model will greatly enhance its power as a general purpose data base interface and will also enhance the usefulness of "local" data bases maintained by ADAPT. Whether or not the ADAPT logical model is sufficiently simple for the majority of end users and whether the ADAPT language is sufficiently concise and powerful must now be determined by experience.

C. ADAPLEX AND MULTIBASE

The Multibase project is sponsored by DARPA and being conducted by Computer Corporation of America in an attempt to solve the multilanguage problem by representing existing data bases within a functional model. The basis for this project is the data definition and data manipulation languages embodied in Daplex (Ref. 36). Previous reports (Refs. 4 and 9) have described various versions of the functional model and have reported on Daplex itself. We are unaware of what progress has been made in the implementation of multibase. A recent report (Ref. 39) has investigated further the problem of schema mappings and query decomposition. It also appears that the syntax, especially the syntax of the type declarations, has changed to conform better to the syntax of languages in the PASCAL family.

Computer Corporation of America is now involved in a substantial effort to use Daplex as the basis for a data base extension to Ada. As we have already mentioned, Ada was designed as a language for embedded systems and real-time programming. However, its power as a general-purpose programming language and the fact that it is destined to be a DoD standard available on most machines means that it is a likely candidate for a data processing language. Should this be the case, a data base extension (which is already desirable for many of the targeted applications of Ada) will become essential.

One of the more powerful features of Ada is its ability to support groups of subprograms that are compiled independently of one another. The normal method of grouping such subprograms is in a package. The proposed extension to Ada, called Adaplex (Ref. 40), encapsulates a data base in a similarly compiled external unit called a module. The type definitions expressed within a module define the data base. The major distinction

between a module and an Ada package is that the data contained in a module "persists" after any given program has terminated.

An Adaplex data base is expressed in terms of functions and sets. Informally, what is normally regarded as a record class in a data base is represented as a set with a collection of functions defined on it. The functions may map the objects of the set into either primitive objects (numbers, character strings, etc.) or into members of some other set. The functions may also be multi-valued and map a given object into a set of other objects. The same name (e.g., STUDENT) may be used both to express the intention of a data type (the type definition) and its extension (the set of all objects that conform to that type). For this purpose, a new data type called entity is introduced. Superficially, an entity declaration is similar to a record definition. However, the "fields" are specifications of (single- or multi-valued) functions. Moreover, subset and overlap constraints may be placed on entity sets. For example, GRADUATE-STUDENT may be constrained to be a subset of STUDENT.

Essentially, the Adaplex data model is a variant of the previously discussed functional models with additional constraints for specifying a subtype or generalization relationship. If it could be cleanly axiomatized, and unfortunately this is not done in the present reference manual [Adaplex], it would be an ideal candidate for the standard data model that we have advocated.

No implementation details are available to us at this time, but it appears that the underlying data structures needed to support Adaplex are straightforward. In fact, no form of indexing is mentioned in Adaplex itself, and indexing would only be required by a "smart" underlying data base manager that could automatically improve the efficiency of certain access paths.

The notion of equality in Adaplex differs from the standard notion in (relational) data base models. Two Adaplex entities (records, tables) were not deemed to be equal because their fields (function values in Adaplex) are equal. Moreover, there is no notion of a key built into the language or implementation, although the effect of keys on updates may be achieved by the addition of (ostensibly expensive) update constraints. The set of

STUDENTs, therefore, should be regarded as a set of objects, two of which may be, at a given time, indistinguishable in all respects. It is not clear, however, what the expression AGE(STUDENT) is to yield. It is a set of integers, with no duplicates or a bag, which allows duplicates. Notice that this distinction considerably affects the computation of AVERAGE AGE(STUDENT).

It will be additionally attractive if this extension of Ada can be used as an interface, not just for a purpose-built data management system, but also as an interface for existing data bases (as was the intention of Multibase). This would provide a tool that should be attractive to all programmers involved in automatic data processing. Also, as we have noted, Adaplex represents an important step in building a programming language with a data base extension, something that has been severely lacking until now. However, it is unlikely that Ada will ever be regarded as a "query" language. The fact that it is compiled and strongly-typed detracts from its use as a simple interactive method for composing data base queries. The learning effort needed to program even simple data base queries in Ada, while it is less than for many existing languages, may still be too great for the casual user and may be too inefficient for the generation of "ad hoc" queries.

D. FUNCTIONAL QUERY LANGUAGE (FQL)

One of the authors (Buneman) has been involved in the implementation of an interactive functional programming system for data bases. Some of the details of this were described in Ref. 4. Implementations have been developed recently that can talk both to a Codasyl system (SEED, Ref. 16) and a relational system (INGRES, Ref. 14). What distinguishes this implementation from the other techniques discussed here is that it is schema independent. There is no need to create a new schema mapping for each new data base. In fact, the system needs only to know what data base management system maintains the data base, and the end user can immediately open that data base, examine the schema presented in functional form, and issue data base queries. This does not preclude schema mappings in cases when it is

desirable to produce "views," but this is done through precisely the same medium, function definition, that is used to query the data base.

The implementation is based on the "coroutine" method described earlier and occupies a few hundred lines of PASCAL and FORTRAN (implementations in both languages are being developed), making it suitable for the "local translation" method discussed earlier. Presently under investigation is a suitable "programming environment"--a set of workspace commands similar to those of APL is being constructed, and a "personal" data base which is attached to a workspace and available for update by the user.

E. MAPPING RELATIONAL TO CODASYL QUERIES

During the past year some work at the University of Aberdeen, Scotland has come to our attention (Refs. 41 and 42). It is a system that maps queries made against a relational data model into queries against a Codasyl data base. In previous reports we have claimed that the main obstacle to building a uniform network interface and to solving the multilanguage problem was the lack of a standard, and simple, data model. Two candidates were presented: the functional and relational models. The disadvantage of the relational model was, we argued, that it is difficult to represent Codasyl (and network data bases in general) as relational data bases, and even more difficult to translate queries against the relational model to queries against Codasyl.

The work at Aberdeen, which is being directed by Dr. P. Gray (Ref. 43) is particularly relevant because it is implemented against IDS, the commercial version of WWDMS integrated data store and currently in use for a large agricultural data base. The system exploits a predefined mapping from a relational schema into IDS structures. Using an "extended" relational calculus that provides for most queries and a large number of statistical queries over the data, users formulate queries that may be run against IDS structures. The approach taken is to translate the user queries into a FORTRAN program (this is another example of the translation approach), and at the same time optimize the query to produce the most efficient traversal of IDS data base. Also, in conformance with the relational model, duplicate

members are removed from sets, thus requiring intermediate files to be created and sorted.

If the relational model is to be adopted as a standard, this work, or work like it, will be extremely important in maintaining interfaces to existing data bases which, for reasons of efficiency or operational circumstances, cannot be converted. In any case, it presents users, especially those interested in any kind of statistical analysis, with a powerful tool for the analysis of IDS data bases.

Another research project being conducted at Aberdeen is the construction of a relational data base management system that will support existing applications even though they were written for some other (non-relational) data base management system. The Prototype of a Relational Canonical Data model with local Interfaces (PRECI) consists of a generalized data base management system that will support user views, and hence applications programs for other data base management systems [a,b]. PRECI supports both retrievals and updates and is designed as a test vehicle for research in data bases. We do not know at present whether any query language is directly supported by PRECI, though presumably a query language for one of the other data base management systems--interfaces for Codasyl, IMS, SYSTEM 2000, and total are all being considered--will run against a PRECI data base.

This project is, in a sense, the opposite of what we have advocated in this report. It provides a single data management system for a variety of existing applications programs and query languages (rather than a single query system for a variety of data base management systems). We think, however, that the technique may prove useful, in the context of network access to data bases. In order to provide a more flexible service to the variety of end users that will have access to an existing data base, it may be appropriate to maintain the data base under a more flexible data management system. In this case it would be highly desirable to allow the physical data base to be converted while the existing applications programs can be used through a "view" of the new schema. If the PRECI project is successful, it may considerably reduce reprogramming costs where some kind of data base conversion is necessary.

REFERENCES

1. Bartee, T.C., and O.P. Buneman, "An Analysis of the Missile Warning Display Error Detection," IDA Letter Report to ASD (C^3I), Information Systems, 24 July 1980 (SECRET)*.

2. Bartee, T.C., "AUTODIN II Alternatives," IDA Letter Report to ASD (C^3I), Information Systems, 1 October 1981.

3. United States Department of Defense, "Reference Manual for the Ada Programming Language," July 1980.

4. Bartee, T.C., and O.P. Buneman, "Data Base Access in C^3I Computer Networks," IDA Paper P-1489, Institute for Defense Analyses, June 1980.

5. "Current ADP Capabilities for Fiscal Year 1980," CCTC, DCA Headquarters, Washington, D.C. 20305.

6. Electronic Warfare Information System (EWIS) Data Base Specification. Command and Control Technical Center System Planning Manual SPM DS, March 1980.

7. Buneman, O.P. and R.E. Frankel, "FQL--A Functional Query Language," ACM SIGMOD Proceedings, Boston, MA, 1979.

8. Electronic Warfare Information System (EWIS) Users Manual. Command and Control Technical Center Systems Manual CSM UM, April 1979, Database Design.

9. Bartee, T.C. et al., "Computer Internetting: C^3I Data Communications Networks," IDA Paper P-1402, Institute for Defense Analyses, April 1979.

10. Milner, R., "A Theory of Type Polymorphism in Programming," J. Computer and System Sciences, Vol. 17, pp. 348-375, 1978.

11. McCarthy, J. et al., LISP 1.5 Programmer's Manual.

12. Warren, D. and L. Pereira, 1977. "PROLOG--The Language and Its Implementation Compared with LISP," Proceedings of the Symposium on Artificial Intelligence and Programming Languages, SIGPLAN Notices, 12, 8 / SIGART Newsletter, 64, 109-115.

---

*In the preparation of this report, no reference was made to classified material from this IDA document.

13. Chamberlin, D.D. and R.F. Boyce, "SEQUEL: A Structured English Query Language," Proc. ACM SIGMOD, 1974.

14. Held, G., M. Stonebraker, and E. Wong, "INGRES: A Relational Database System," Proc. AFIPS 1975 NCC Vol. 44.

15. Astrahan, M.M. et al., "System R: A Relational Approach to Data Base Management," ACM Trans. on Database Systems 1, No. 2, June 1976.

16. SEED User Manual, International Database Systems, 2300 Walnut Street, Philadelphia, PA 19103, 1980.

17. Harvest Reference Manual, International Database Systems, Philadelphia, PA 19103, 1978.

18. Codd, E.F., "Extending the Database Relational Model to Capture More Meaning," ACM Transactions on Database Systems, December 1979.

19. Shipman, D.W., "The Functional Data Model and the Data Language Daplex," ACM Trans. Database Systems, 6,1, 1981.

20. Soleglad, M.E. et al., "ADAPT II Specifications," Logicon Inc.

21. Zloof, M.M., "Query By Example," Proc. NCC 44, May 1975.

22. Morgan, H.L., "A Spelling Correction Algorithm," Comm ACM, 1974.

23. TENEX Users' Manuals, Bolt, Beranek and Newman, 50 Moulton Street, Cambridge, MA 02238.

24. Tops-20 Manuals, Digital Equipment Corporation, Maynard, MA.

25. Kaplan, S.J. and A.J. Joshi, "Cooperative Responses: An Application of Discourse Inference to Database Query Systems," Proc. 2nd Workshop on Theoretical Issues in Natural Language Processing, Urbana, Illinois, 1978.

26. Hendrix, G.G., E.D. Sacerdoti, D. Sagalowicz, J. Slocum, "Developing a Natural Language Interface to Complex Data," ACM Trans. on Database Systems, June 1978.

27. Janas, J.M., "Towards More Informative User Interfaces," Proc. VLDB Conference, Rio de Janeiro, 1979.

28. Hammer, M. and D.J. McLeod, "The Semantic Data Model: A Modelling Mechanism for Database Applications," Proc. ACM SIGMOD, 1978.

29. Smith, J.M. and P.Y. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," Comm. ACM, Vol. 18, 10, 1975.

30. Smith, J.M. and D.C.P. Smith, "Database Abstractions: Aggregation and Generalization," ACM TODS, Vol. 2, 2, 1977.

31. Buneman, P., R.E. Frankel and R. Nikhil, "An Implementation Technique for Database Query Languages," ACM Trans. on Database Systems (in press).

32. Wirth, N., "The Programming Language Pascal," Acta Informatica, Vol. 1, 1, 1971.

33. Wasserman, T., "The Data Management Facilities of Plain," Proc. ACM SIGMOD Conf., 1979.

34. Rowe, L., "User Views in Rigel," Proc. ACM SIGMOD Conf., 1979.

35. Schmidt, J.W., "Some High Level Language Constructs for Data Type Relation," ACM Trans. Database Systems 2, 3, 1977.

36. Shipman, D., "The Functional Data Model and the Data Language DAPLEX," ACM Trans. on Database Systems, March 1981.

37. Iverson, K.E., "Operators," ACM TOPLAS Vol. 1,2, 1979.

38. Buneman, P., L. Menten, D.J. Root, "A Codasyl Interface for Pascal and Ada," Moore School Report, University of Pennsylvania, 1981.

39. "Multibase--A Research Program in Heterogeneous Distributed Database Technology," Computer Corporation of America.

40. Adaplex Reference Manual, Computer Corporation of America Technical Report, 1981.

41. Deen, S.M., "A Canonical Schema for a Generalized Data Model with Local Interfaces," Computer Journal, 23, 3, 1980.

42. Deen, S.M., D. Nikodem, and A. Vashishta, "Design of a Canonical Database System (PRECI), Computer Journal, 24, 3, 1981.

43. Gray, P.D., "Implementing Relational Queries Against a Codasyl Database," University of Aberdeen Technical Report, 1980.