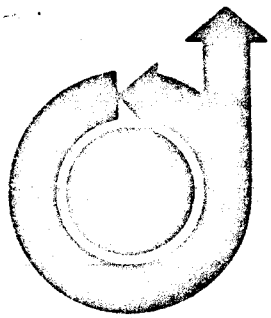


AIAA Paper
No. 73-417

THE INTEGRATION OF DATA MANAGEMENT SYSTEMS ON A COMPUTER NETWORK

by
A. SHOSHANI and I. SPIEGLER
System Development Corporation
Santa Monica, California

AIAA Computer Network
Systems Conference
HUNTSVILLE, ALABAMA / APRIL 16-18, 1973

THE INTEGRATION OF DATA MANAGEMENT SYSTEMS ON A COMPUTER NETWORK*

A. Shoshani, I. Spiegler
System Development Corporation
Santa Monica, California

Abstract

In this paper we discuss an approach to integrating data management systems on a computer network for the purpose of data sharing. Our approach to integration suggests the use of a common language and translation interfaces to target data management languages. Properties of the common language are explored, and a method of implementing the translation interfaces by a meta-compiler is described. More flexibility can be achieved by the use of a natural language processor that permits user requests to be stated in English and translated into the formally structured common language. Finally, some conclusions are drawn regarding the desirability and feasibility of this approach.

Introduction

The need for data sharing arises in many application areas. Examples are reservation systems, hospital networks, library networks, centralized or distributed banking systems, and military and government information systems. These systems accumulate large amounts of data that are of interest to a large community of users. The resulting data bases may become too large for economical duplication for different applications. Aside from that, using the same data for different applications often requires restructuring data, a task that is difficult and expensive. Finally, there is a growing need for real-time and on-line data sharing. Computer networks provide the basic facilities necessary to meet these needs.

Examples of computer networks are discussed in the literature [1, 2, 3, 4]. For our purpose, it suffices to know that a computer network is a collection of computer systems interconnected through a communication network.

In a previous paper[5] we discussed a number of approaches to data sharing in computer networks. The different approaches (labeled as the centralized, integrated, data transformation, and standarized approaches) have properties that make them attractive and applicable under different conditions. We chose to pursue our investigation of the integrated approach because it has the advantage of permitting the sharing of data among network nodes without interrupting the continued use of existing data management systems and existing data bases.

The integrated approach suggests the use of a common language and translation interfaces to target data management languages. Therefore, it permits the continued access of a particular data management system either through its local data management language or through the common language. However, the use of the common language permits the access of other data management systems in the network in addition to the local one. This dual mode of operation would allow users of existing data management systems to use the common language facility without relinquishing their previous local mode of operation. This is an important advantage of the integrated approach, since it allows data sharing to be introduced into computer networks in an evolutionary manner and is more likely to be accepted by existing users. Another advantage of the integrated approach is that it permits the co-existence of many systems, thus allowing for further development of new systems and their integration into the network without disturbing existing systems.

Next, we present an overview of the integrated approach, then discuss the properties of the common language and our experience in implementing translation interfaces using a meta-compiler.

Overview of the Integrated Approach

Conceptually, the integrated approach provides a way in which all users on a computer network can access all data management systems, as shown in Figure 1. A user, regardless of his physical location, can issue a request expressed in the common language to be processed by a remote data management system (DMS). An interface module associated with that DMS accepts the request and translates it into the language of the DMS. The DMS performs the function requested and returns the reply to the user. Note that all data management functions are performed by existing data management systems. The task of the interfaces is merely to perform the translation of the request. There is one interface for every data management system. The returned reply can also be transformed into a common language format. In fact, this is a useful concept if we wish to combine returned data from more than one DMS into a single reply. This situation arises when data are distributed on several systems and need to be shared for a common reply. An example could be a hospital network, where every node is a local system associated with a hospital; we might want to integrate information about all local blood banks so that they could be accessed jointly.

Depending on the physical location of the interfaces, two main variations of the integrated approach are possible. In the integrated-distributed variation, the interfaces are physically co-located with the corresponding data management systems, as shown in Figure 2. In the integrated-central variation, all the interfaces are physically located in one node and implemented on that central system, as shown in Figure 3. One difference in properties between the two variations is that the integrated-distributed one has fail-soft characteristics, because the failure of one node does not disturb data

*This research is supported by the Advanced Research Projects Agency of the Department of Defense under Contract No. DAHC15-73-C-0080.

accessing between other nodes. However, the integrated-central approach should be easier to implement, because all interfaces are implemented and maintained in one central system.

The interfaces described above are associated with the serving node. An additional variation is to associate an interface with the using node. The using interface can perform additional functions or some of the functions of the serving interface; thus, the nature of messages between nodes will vary. For example, it can accept a request from the user in the common language and represent it in some intermediate language before sending it to the serving interface. This will permit more compact descriptions of requests to be sent between nodes. In the next section (on the common language) we discuss an example of a serving interface--a natural language processor which translates English requests into a precise intermediate language.

An important problem associated with the integrated approach is the control of traffic. Information about the source and destination of requests and returned data must be coordinated. This function is usually taken care of by the message switching mechanisms of the computer network. The situation is much more complex in the case of data returned from a distributed data base which need to be integrated. Another problem that cannot be overlooked is that of privacy and security. It appears that a central node which will check the legality of requests according to a central authorization file would be an appropriate and efficient way to solve this problem.

The Common Language

The use of a common language should disengage the individual users from concern as to where the data are located and in what form data can be accessed or stored. Some obvious properties of a well-defined common language are that it be easy to use and learn and that it be capable of expressing the functions desired. In addition, a common language should be based on general enough data structures to allow the reference of complex information. Currently, we limit ourselves to hierarchical data structures which are powerful enough for a large range of applications (hierarchical structures are discussed later in the section about the intermediate language).

The design of a user-level language usually involves a compromise between the properties mentioned above--ease of use and functional expressive power. It may be advantageous to separate these characteristics--that is, have a user-oriented language for ease of use, and a formal intermediate language powerful enough to express any functions desired. The intermediate language can be as complex as one likes, since it is invisible to the user. A transformation process is necessary from the user-oriented language to the intermediate language. In the next section we describe in detail our investigation of the functional properties of the intermediate language, which we call ILDS (Intermediate Language for Data Sharing). Because ILDS is a precise language, it is possible to build translator interfaces from it to target data management languages by using a meta-compiler. Our experience with implementing translators is described in a later section.

A user-oriented language that is most natural, of course, is English. We make use of CONVERSE[6], an experimental English-language DMS. Placing the CONVERSE "front end," a natural language translator, as an additional block in Figure 1, divides the line representing the common language into input and output parts. The input is an English request on which the CONVERSE front end performs syntax analysis and semantic interpretation to produce well-defined intermediate language (IL) statements as its output. Our experience with the CONVERSE natural language translator suggested the following important conclusion. A context-restricted environment (represented by a target data base) requires a translation process which is more powerful than the syntax-based parsing techniques on which CONVERSE is based. A semantic conceptualization of the data base is required which represents the possible English requests addressed to the data base. These concepts, which represent the possible user view of the data base, need to be represented in the machine if user requests are to be successfully interpreted. Much of our work is currently oriented in this direction. A natural language processor also needs to have access to information about the data base in order to "understand" the English request. Therefore, with every data base, a data base description (DBD) is required which contains the semantic concepts of the data base. This suggests the use of a central file which contains DBDs of data bases to be shared on the network.

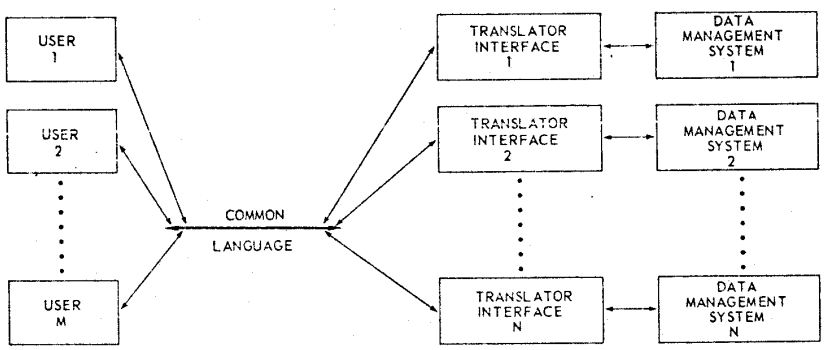


Figure 1. Conceptual Representation of the Integrated Approach

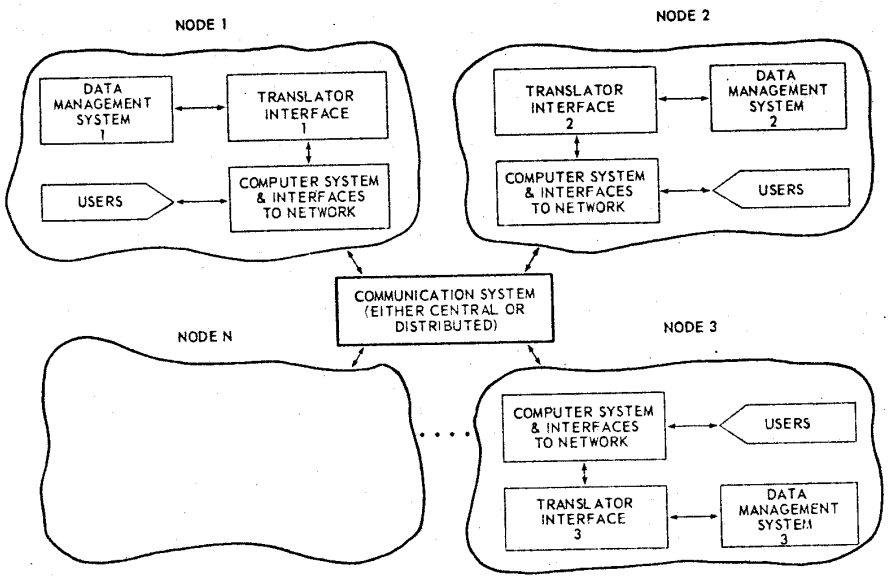


Figure 2. The Integrated-Distributed Approach

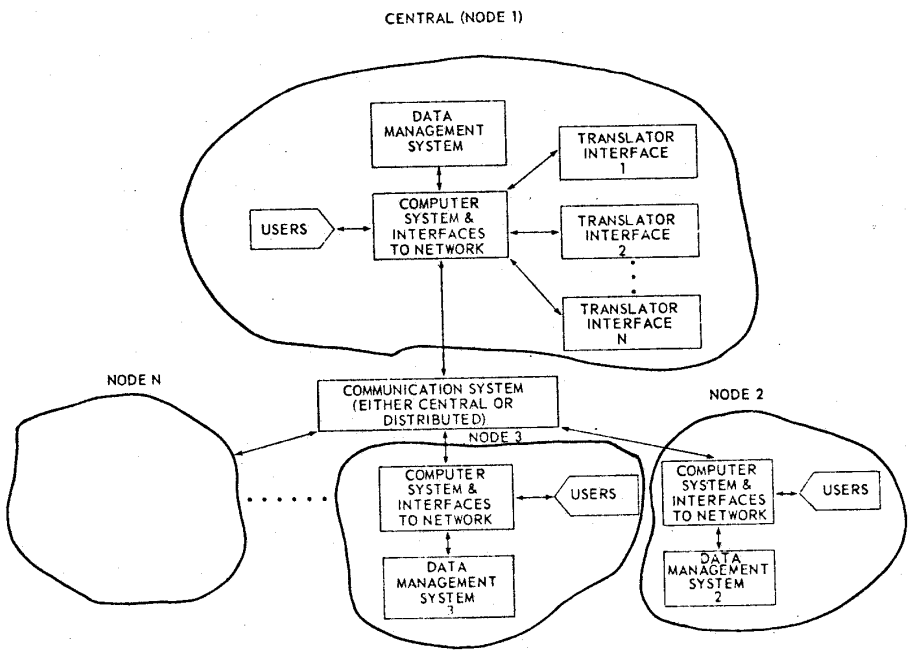


Figure 3. The Integrated-Central Approach

Note that if a precise user-oriented language is used, no information about the data base is necessary. However, a precise knowledge of the data base is required by the user (which is a common assumption made when using current data management languages).

Functional Properties of the Intermediate Language

Our main concern in the development of ILDS was to identify the possible functions which operate on data structures. In putting those functions into a syntax form, we attempt to present the functions clearly even at the expense of having a less concise form. In what follows, the basic assumptions about data structures are presented and a detailed description of the functions operating on them is given.

As mentioned earlier, we decided to limit ourselves to hierarchical data structures because they are sufficient for a large number of application areas. (In the future, we might expand to more general network data structures). Hierarchical data structures (sometimes called tree structures), which are common to most existing data management systems, represent a logical way of organizing data. Descriptions of hierarchical data structures can be found in many references (for example [7, 8]; for the purpose of introducing our terminology, let us refer to an example of a hierarchical data structure in Figure 4. The name of the hierarchy is RD (for the Research and Development Division). It has two branches, represented in the leftmost table in Figure 4. Every column in that table, except PROJECTS, is called a "data element" (DE). The elements in a column are called "values" of the DE. Thus, "TECH" and "DMS" are values of the DE "BNAME". Rows in that table are called entries. (In Figure 4, entries are numbered for the purpose of later reference in examples). The column called "PROJECTS" is called a "repeating group" (RG) because it has many instances of projects associated with every entry. For the purpose of unified reference, "BRANCHES" is also considered to be a repeating group. Thus, in this example, we have three repeating groups (BRANCHES, PROJECTS, MEMBERS). Each table, however, is an instance of a repeating group. Thus, the BRANCHES RG has one instance, the PROJECTS RG has two instances, and the MEMBERS RG has five instances. We say that three DEs and one repeating group "belong" to the BRANCHES RG (BNAME, BHEAD, BSEC and PROJECTS respectively). Similar relationships hold for the other repeating groups. Note that, in general, a repeating group can have more than one repeating group "belonging" to it. For the purpose of simplicity, we assume that DE names and RG names are unique, even though uniqueness of DE names in different RGs is not necessary, since the DEs can be distinguished by the RG they belong to.

The terms "hierarchy", "repeating group" (RG), "data element" (DE), "value", and "entry" discussed above will be used in the following description of the intermediate language ILDS. First, we describe a kernel version of ILDS whose syntax is given in the appendix. Then, we explain additional possible functions which can be added to the kernel version, depending on what level of complexity one chooses to employ.

The kernel version has four functions: create, delete, update, and query. These functions are described in terms of three basic elements: qualifier, output, and replacement. The qualifier consists of functions that can be applied recursively to a hierarchy to determine what entries qualify. The entries that qualify can be operated upon by an output function to produce values, or by a replacement function to change values in those entries. Thus, the create function has only a replacement part, which specifies how to create new entries; the delete function has only a qualifier part, which selects entries to delete; the update function has a qualifier part which selects entries to be modified according to a replacement part; and the query function has a qualifier part which selects entries on which an output part operates to return values.

In the kernel version, the qualifier part is most powerful, since it is composed of recursive applications of elementary "entry-functions" (efunction

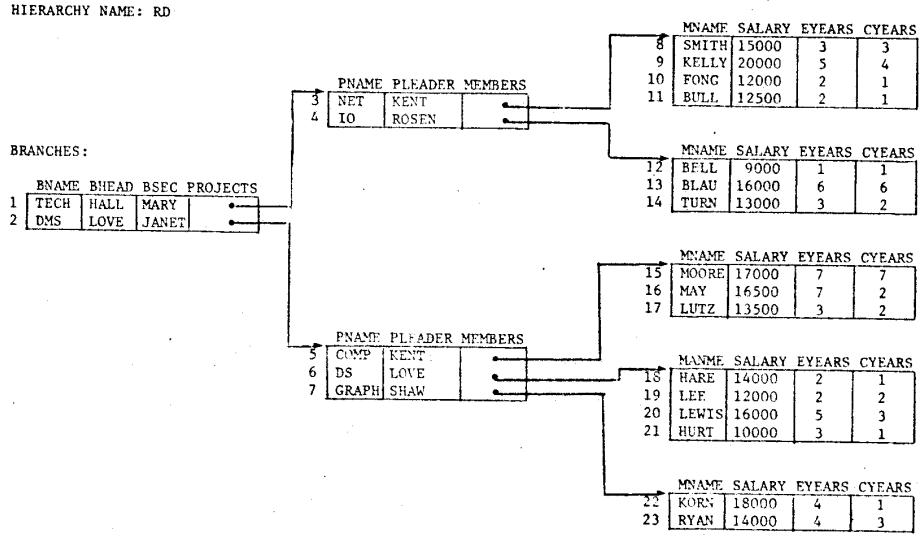


Figure 4. An Hierarchical Data Structure

in the syntax). The syntax form:

qualifier = efunction (qualifier)

means that a qualifier is formed by applying an entry-function on entries that qualified by the inner qualifier to get a new set of qualified entries. Thus, an entry-function is one that maps sets of entries into sets of entries. An additional restriction on a qualifier is that it represents a set of entries of the same repeating group. This fact is reflected in the structure of entry-functions, which will be examined later. To terminate the recursive form we need, of course, a terminal qualifier (tqualifier), which in our case is a hierarchy name (hname). Thus, the first entry-function in a qualifier operates on the named hierarchy. The use of a hierarchy name in a qualifier also provides for operations across multiple hierarchies to be expressed.

Every entry-function has an "implied" RG associated with it, which is the RG for which entries qualify after the application of the entry-function. We will indicate the implied RG of every entry-function when discussing each entry-function. First, however, we need to explain some additional terms used in the syntax. The term "val" means a value or a set of values which can be either explicitly specified or extracted from the data base (we will explain later how values can be extracted). The term "col" (column) is either a DE or a recursive application of arithmetic operations on one or more DEs. For example, "((SALARY/12) + 50)" is a legal "col" (which might mean monthly salary + $50). The term "pred" (predicate) means one of the commonly used predicates: equal, greater, less, greater or equal, less or equal, not equal. The term "sop" (set operator) stands for one of the operators on a set of values: sum, average, maximum value, minimum value, and count number of values in the set.

The significance of the entry-functions is that they represent the most elementary functions for qualifying entries in a hierarchy. The entry functions are described below:

• Select. The select function is the most common qualification function. Its form is: select=col pred val, which means: compare according to "pred" every value specified by "col" with the value "val" and, if the comparison succeeds, qualify the corresponding entry in the RG "implied" by the DE in "col". For example: (SALARY EQ 12000(RD)) means: for the RD division to compare each value in the "SALARY" "col" (here "col" is a DE) to 12000, and if they are equal, qualify the entry in the RG "MEMBERS", which is the RG implied by the DE "SALARY" (the "SALARY" DE belongs to the "MEMBERS" RG). Thus, in Figure 4, entries 10 and 19 in the "MEMBERS" RG qualify. An example of a "col" which is not a simple DE is: ((SALARY/12) EQ 1000 (RD)), which qualifies entries in the "MEMBERS" RG with monthly salary equal to $1000. This, of course, is an equivalent example to the previous one.

• Group. The group function operates across levels of the hierarchy. It qualifies entries in a RG according to a set operation (SOP) on values in a RG which belongs to it. For example: qualify entries in the "PROJECTS" RG for which the average "SALARY" per project is greater than 15000. Its form is: group = rg sop col pred val. For the example above we have: (PROJECTS AVG SALARY GR 15000 (RD)) (entries 5,7 qualified). The implied RG for the group function is, of course, the RG which appears in the function. Note that the group function can operate across more than one level, such as the entries in the "BRANCHES" RG for which average "SALARY" per branch is greater than 15000.

• Scope. This function, too, operates across levels of the hierarchy. Its purpose is to qualify entries in a RG in one level as a result of entries that qualified in a RG in another level. For example, suppose that we want to qualify branches if they contain at least one project that qualified in our previous example. We apply a scope function (which is simply--scope = rg) to the previous example as follows: (BRANCHES (PROJECTS AVG SALARY GR 15000 (RD))) (entry 2 qualified). We can either "scope up" or "scope down", depending on whether the scope RG (RG_s) is at a level higher or lower than the implied RG of the qualifier it operates on (RG_q). When "scoping up", we qualify entries in RG_s for which there is at least one entry qualified in RG_q. When "scoping down", we qualify all entries in RG_s for every entry that qualified in RG_q. The previous example is one of "scoping up", the following example is one of "scoping down": (MEMBERS (PNAME EQ IO (RD))), which simply qualifies all members of the IO project.

• Compare. The compare function qualifies an entry by comparing values from two columns for that entry. For example, suppose that we want to qualify members that have the same number of years of experience (EYEARS) and years with the company (CYEARS). The syntax representation for this example is: (COMP EYEARS EQ CYEARS (RD)). Entries qualify by comparing EYEARS and CYEARS pairwise by entry (entries 8, 12, 13, 15, 19 qualify). The syntax form for this function is: compare = COMP col pred col. Both "col" elements in this function must belong to the same RG. This RG is, therefore, the implied RG.

• Loop. This function is similar in form to the compare function: loop = LOOP col pred col. However, instead of comparing values from the columns pairwise, it compares one value from the first column with each value from the second column; then it repeats the process for all other values from the first column. For example, suppose we want to qualify entries in the "BRANCHES" RG for which a branch head (BHEAD) is also a project leader (PLEADER). This example is expressed by: (LOOP BHEAD EQ PLEADER (RD)) (entry 2 qualify). Note that the columns can belong to different RGs; therefore, their order is important. We adopt the convention that the implied RG is the one to which the first column belongs.

• Logical Qualification. This function is a compound one which permits qualifiers to be joined logically by the AND and OR logical operator (lop). It is expressed in a recursive form to allow qualifiers to be joined logically as many times as necessary. For example, if we want to qualify members whose salary is greater than $18,000 or who have more than 6 years of experience, we join two select functions as follows: ((OR (SALARY GR 18000)(EYEARS GR 6))(RD))(entries 10,15,16 qualify). It is permitted to join qualifiers whose implied RGs are in different levels in the hierarchy; however, the implied RG of the joined qualifier is always the lower of the two.

When an entry-function is being applied to a qualifier, scoping functions often become redundant as in the following example:

(PLEADER EQ KENT (PROJECTS(BNAME EQ DMS (RD))))

which means: in the DMS branch, qualify these entries in the PROJECTS RG for which project leader is KENT. (In this case only entry 5 qualifies). The scope function PROJECTS in the form above is not necessary since the form:

(PLEADER EQ KENT (BNAME EQ DMS (RD)))

implies a scoping down between the two implied RGs "MEMBERS" (from BNAME) and "PROJECTS" (from PLEADER). Scope functions are necessary, however, as can be illustrated in the following example:

(MEMBERS (PROJECTS (YEARS GR 6 (RD))))

which means: for the project which has a member with more than 6 years of experience, qualify all members. After selecting on "YEARS" entries 15, 16 qualify. After scoping down to "MEMBERS" RG, entries 15, 16, 17 qualified.

We mentioned earlier that values can be extracted from the data base. This is done by applying a value-function (vfunction) to a qualifier. Thus, the value-function maps entries into values. The simplest value function is: vfunction = col. It allows the extraction of all the values under a "col" for those entries that qualified. Thus, in the example of Figure 4, the form (MNAME) (SALARY EQ 12000 (RD)) returns the values FONG and LEE. A more complex value-function is the single-value-function (svf) which extracts one value only by applying a set-operation (sop) on the values of a column. For example, (AVG SALARY) (EYEARS GR 4(RD)) extracts the average salary of members with more than 4 years of experience. An extension of the svf is the multiple-value-function (mvf) which extracts multiple values by grouping the entries it operates on. For example, (PROJECTS AVE SALARY) (EYEARS GR 4(RD)) extracts the average salary per project for members with more than 4 years of experience.

The value-function is used, most naturally, for output, but it is also used to extract values to be used for further qualification. For example, to qualify members with salary greater than the average salary, we need to extract the average salary using a value-function as follows: SALARY GR (AVG SALARY (RD)). Another output function is TF (true-false), which returns false if no entries qualify, otherwise true. This function is necessary for answering questions about the data base, such as (TF) (PLEADER EQ LOVE (RD)) to represent: "is LOVE a project leader?"

Additional Properties of the Intermediate Language

The kernel version of ILDS is not an attempt to represent all possible functions on a hierarchical data base, rather, it provides us with a powerful enough tool for our experimentation with translating from it into target data management languages and from English into it. In the course of developing ILDS, we recognized many functional properties which are not expressed in the kernel version. We describe them briefly below:

- Output attachment. A qualifier that is formed by a repeated application of entry-functions has many levels of qualifiers imbedded in it, one for every application of an entry-function. It might be useful to attach an output function to the different levels, rather than to the top-level qualifier only. Output attachment is important for separating the output process from the qualification process.

- Multiple application of entry-functions to a qualifier. Once a qualifier was expressed it might be desirable to allow more than one entry-function to apply to it. This will form a "tree structure" whose nodes are qualifiers and whose arcs are entry-functions. Output attachment can be made to the nodes of the tree. An easy way to express both this function and the output attachment function is by a numbering scheme for the qualifiers in different levels. Entry-functions and output-functions refer to the qualifier they operate on by its number.

- Additional output functions. These include the important functions of output formatting and sorting.

- Additional reference functions for qualification. These include:

  - "Every" function--to qualify, for example, projects for which every member has salary greater than $15,000. This function is also called "the universal quantifier".

  - "Same" function--to qualify, for example, projects with the same project leader.

  - "Order reference" functions--to extract elements from ordered lists in a data base. For example, if a list of people is ordered by age (oldest first) in a data base, one can obtain the youngest of them by requesting the last person on the list.

- Programming language functions. These are functions commonly used in programming languages, such as conditional statements (IF statements), repeated execution (FOR statements) and subroutines.

- Additional data management functions. In the kernel version of ILDS, our main emphasis is on qualification for the purpose of retrieval of data. Many additional data management functions are surveyed in the literature[9,10], such as data definition functions and data modification functions. (The kernel version has only a basic, simple replacement part for the update and create functions.)

The Translator Interfaces

We decided to implement several translator interfaces in order to gain actual experience, possibly identify unforeseen problems, and develop a better idea of the feasibility of the integrated approach. We describe below some functions that an interface must perform.

- Translation from the common language to the target data management language (DML). The translation process has to be tailored for the specific DML, in that it needs to take advantage of special functions that can be expressed by the DML. If these are two ways of representing a request in the DML, the more efficient one must be chosen. In the case of ILDS, we have to consider whether a quali-

fier with multiple entry-functions can be expressed jointly through a single request in the DML.

- Appropriate refusal of a request. If the target data management system is not capable of performing a request expressed in the common language, then the interface needs to refuse it properly. Rather than refusing it as "cannot be done", an indication as to why it cannot be done might help the user to rephrase his question.

- Translation of returned data. Replies from the data management system must be put in a common data format if there is a need to integrate replies coming from different systems (this point was discussed earlier). Furthermore, a common data format is necessary for returning complex data such as reports. In that case, a report which is produced by a system in order to be output on some local I/O device needs to be shipped to another remote system which might have different I/O devices. A "post-processor" at the remote node would then receive the report in the common data format and output it on an I/O device designated by the user who issued the request. The problem of shipping data and files on computer networks has been recognized as a general one, and work in this direction has been initiated[11]. If general mechanisms for data transfer are available on a computer network, then the interface should use them for returned data.

- Translation of error messages into a common error format. Every data management system usually has its own error-message format. It is useful to have them translated into a common format so that users have to be familiar with only one error format.

- Control functions. The interfaces need to have control mechanisms to handle multiple requests coming from different users on the network. This type of control mechanism should normally be part of network functions performed by every node. An additional control mechanism is necessary in order to establish communication between the interface and the target data management system. Control functions should use interprocess communication mechanisms available on the computer network, such as the experimental one described in[12].

Our experiments with implementing interfaces concentrated in the area of translation from the common language to the target data management languages because we wanted to learn more about this process of translation and its implications. An important conclusion was that if temporary storage functions (or their equivalent) are available in the target data management system, then it is possible to enhance the apparent functional power of that system by splitting complex requests in the common language into a series of requests in the DML. For example, in the case of DS/2 (an SDC data management system) a "SUBSET qualifier" function restricts the data base for the next operation according to the qualifier. The translator to DS/2 can take advantage of this function by representing, say, a complex query request as a series of SUBSET functions followed by a simple request that can be expressed in the DS/2 DML.

As mentioned before, we implemented the translation by means of a meta-compiler (an SDC product called CWIC[13]). This involved writing programs in the CWIC language, a task that was quite complex but not lengthy. We implemented two translators from an earlier version of ILDS (which was based on relational data structures) to the DS/2 and TDMS[7] data management languages. We spent 2-3 man-months to implement and debug each translator on the CWIC system.

SUMMARY AND CONCLUSIONS

The integration of data management systems on a computer network might be achieved by using the methodology of a common language and translation interface. In addition to the advantage of integrating existing data management systems and data, this approach facilitates the evolutionary development of new systems. The dual mode of accessing data--using the data management language or the common language--should be of great value for gradual adaptation to computer network data sharing. A disadvantage of this approach is that translator interfaces add a level of language translation. However, our experience shows that the translation of data management languages is a manageable task. Furthermore, considering that control, communication, and data transfer functions need be performed in any approach for data sharing on a network, the addition of translation tasks may not be so significant.

The separation of the common language into two levels--the English language for user convenience and an intermediate language for precise expressive power--helped us to concentrate on exploring the functional properties desirable in a common language. The common language should represent a "union" of data management languages, so that it can represent requests for all data management systems. Our approach was to identify the elementary functions on logical data structures and allow for their recursive application. A kernel version of this language was defined and experimental translators from it to two example data management systems were built by the use of a meta-compiler.

Our experience shows that these translators were relatively easy to implement with a meta-compiler (about 2-3 man-month per translator). Another important conclusion was that it is possible to add functional power to target data management systems by translating a single request in the common language into a series of requests in the target data management language.

The use of English as a common language is not useful for sophisticated users. It is very difficult, for example, to express in English a precise report request. In addition, a description of the target data base must be available to the natural language processor for it to process English requests. Therefore, a precise common language is necessary whether or not an English input mode is available.

To summarize, the integration of data management systems on a computer network using techniques described in this paper offers many advantages. It should lead to sharing of existing and new data, to gradual acceptance of the computer network data sharing concept, to the natural survival of better data management systems and elimination of less useful, less efficient ones, and to the elimination of duplicated work in the area of data management system development. Its feasibility is predicated on a well defined computer network with

advanced interprocess communication and data transfer capabilities.

Acknowledgment

We would like to thank Gunther Luther of SDC for his help and many ideas in the development of ILDS.

Appendix: Kernel Version of ILDS

The syntax for ILDS is expressed in BNF form except for braces ({ }) which are used to indicate that the elements between them may appear one or more times. Note that parentheses are terminal symbols in the syntax.

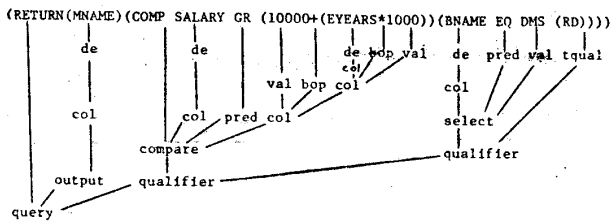
The Syntax

```
ilds       ::= (<create>)|(<delete>)|(<update>)
                 |(<query>)
creat      ::= CREATE (<replacement>)
delete     ::= DELETE (<qualifier>)
update     ::= UPDATE (<replacement>)(<qualifier>)
query      ::= RETURN ({<output>})(<qualifier>)
output     ::= <vfunction> | TF
qualifier  ::= <efunction>(<qualifier>)|(<tqualifier>)
efunction  ::= <select>|<group>|<scope> | <compare>
                 |<loop>|<lqual>
select     ::= <col><pred><val>
group      ::= <rg><sop><col><pred><val>
scope      ::= <rg>
compare    ::=  COMP  <col><pred><col>
loop       ::=  LOOP  <col><pred><col>
lqual      ::  (<lop><lqual><lqual>)|(<qualifier>)
val        ::= (<vfunction>(<qualifier>))|({<pval>})
vfunction  ::= <col>|(<svf>)|(<mvf>)
svf        ::= <sop><col>|<sop>(<mvf>)
mvf        ::= <rg><sop><col>
col        ::= <de>|(<uop><col>)|(<col><bop><col>)
                 |(<col><bop><val>)|(<val><bop><col>)
pval       ::= <string>|<number>
tqualifier::= <hname>
replacement::= {(<de><val>)}
lop        ::=  AND | OR
pred       ::=  EQ | GR | LS | GQ | LQ | NQ
uop        ::=  -
bop        ::=  + | - | * | / | **
sop        ::=  SUM | AVG | MAX | MIN | COUNT
hname      ::=  identifier
de         ::=  identifier
rg         ::=  identifier
```

To shorten the syntax presentation here, the elements string, identifier and number are not expressed in detail. (Their interpretation is the normal one: number--any integer, fixed point or floating point number; string--a concatenation of one or more letters, digits and special characters, identifier--is a string whose first character is a letter).

An example of a query

The following is a query written in the ILDS language and the representation of its parsed tree. The query requests names of employees in the DMS branch whose salary is greater than the amount of $10000 plus $1000 for every year of experience.



References

1. Farber, David J. "Networks: An Introduction", Datamation, Vol. 18, 4 April 1972, pp. 36-40

2. Bell, G. C. et al. "Computer Networks", Computer, September-October 1970, pp. 13-21.

3. Roberts, L. G., and Wessler, B. "The ARPA Computer Network", in Computer Communication Networks, edited by Abramson and Kus, Prentice Hall, 1972.

4. McKay, D. B., and Karp, D. P., "Network 440--IBM Research Computer Sciences Department Computer Network", IBM Research Report RC3431, July 1971.

5. Shoshani, A., "Data Sharing in Computer Networks", WESCON Conference, Computer Networks Session, Sept. 1972 (Also, ARPA NIC Document 12623, Stanford Research Institute, Oct. 1972).

6. Kellogg, C., et al, "The CONVERSE Natural Language Data Management System: Current Status and Plans", Proceedings of the Symposium on Information Storage and Retrieval, University of Maryland, 1971.

7. Bleier, Robert E., "Treating Hierarchical Data Structures in the SDC Time-Shared Data Management System (TDMS)", Proceeding of ACM National Meeting 1967, pp. 41-49.

8. Engles, Robert W., "A Tutorial on Data-Base Organization", Annual Review of Automated Programming, Vol. 7, Part 1, 1972, pp. 1-64.

9. Fry, J. P. et al, "Data Management Systems Survey", MTP-329, The MITRE Corporation, May 1969.

10. "Feature Analysis of Generalized Data Base Management Systems", CODASYL Systems Committee, April 1971.

11. Crocker, S. D. et al, "Function-oriented Protocols for the ARPA Computer Network", Proceedings of AFIPS 1972 SJCC, pp. 271-280.

12. Carr, S., Crocker, S., Cerf, V. "HOST-HOST Communication Protocol in the ARPA Network", Proceedings of AFIPS 1970 SJCC, pp. 589-598.

13. Book, E., Schorre, D. V., and Sherman, S. J., "The CWIC/360 System, A Compiler for Writing and Implementing Compilers", ACM SIGPLAN Notices, Vol. 5, No. 6, June 1970, pp. 11-29.