An information base for procedure independent design of information systems

by LEVENT ORMANCIOGLU
University of Wisconsin-Milwaukee
Milwaukee, Wisconsin

PROCEDURE INDEPENDENT DESIGN

An information system contains a vast number of logical statements in addition to data. These logical statements are used to control the retrieval and use of data for applications. Systems Design is the process of translating the information requirements of an organization to a set of logical statements which operate on the observed data to generate new data. The 'derived data' generated in this fashion could actually be used for decision support or stored back in the data base for future use; however, the storage is rare since it can always be regenerated using the logical statements. The logical statements are often expressed using a procedural programming language mostly aided by Data Base Management Systems which accomplish to decouple the physical data structure and the application system designer's logical view of it. This feature, called 'data independence,' has a number of advantages such as:

a. simpler interface with the applications programmer
b. machine optimized physical design
c. modifications in the physical structure without changing the application programs
d. high portability to different computer systems.

The main difficulty with data independence is the fact that shielding the application programmer from the physical data structure also deprives him of the information needed to develop efficient algorithms. To restore efficiency without sacrificing data independence requires delegating the responsibility of generating algorithms to the information system. This responsibility includes not only the generation of data retrieval algorithms but also application system algorithms since the efficiency of both are strongly interrelated. The application system designer in such an environment would have to avoid specifying algorithms and procedures; and restrict himself only to specifying the functional relationships between the data to be generated and the data in storage. This approach to design is called 'procedure independent' and Information Base is an approach based on Predicate Calculus and Array Algebra to the procedure independent design of information systems.

A number of data sublanguages based on Predicate Calculus have been developed including ALPHA, COLARD, RIL; and a number of others with more user orientation like QUEL, SQUARE, and QUERY by EXAMPLE are based on similar principles. All of them proved to be relationally complete which refers to their selective ability in terms of identifying the subsets of data. However, in addition to data selection, an information system needs the ability to algebraically manipulate the data to generate new data. This need has long been recognized as a need to host a data selection sublanguage with a programming language. Information Base uses APL operators to algebraically manipulate data which has been retrieved using predicate calculus. The capability of iteration between algebra and calculus increases the power of the language.

INFORMATION BASE

The development of the Relational Data Base Theory and the use of first order predicate calculus made it possible to view data as a collection of two dimensional tables and retrieve subsets of data non-procedurally. An information base adds to these capabilities in two ways:

a. by developing a 'function specification language' (FSL) based on predicate calculus and array algebra to express the functional relationships between the data to be generated for applications and the data in storage;
b. by storing the functional expressions in FSL—called functions—in a 'function base' and identifying each function by its unique output. This makes it possible to treat the functions as a set without reference to their sequence.

FSL is an all-purpose language used to specify functions, to define and manipulate data and to enter queries. A query expressed in FSL goes through three stages and a number of iterations among them to produce a response:

a. the query is interpreted as a function to determine its input data requirements;
b. for each data item required but not stored explicitly, the appropriate function is retrieved from the function base;
c. the query is executed as a function on the retrieved data base. The result of this process is then displayed to the query programmer.

The development of the Relational Data Base Theory and the use of first order predicate calculus made it possible to view data as a collection of two dimensional tables and retrieve subsets of data non-procedurally. An information base adds to these capabilities in two ways:

a. by developing a 'function specification language' (FSL) based on predicate calculus and array algebra to express the functional relationships between the data to be generated for applications and the data in storage;
b. by storing the functional expressions in FSL—called functions—in a 'function base' and identifying each function by its unique output. This makes it possible to treat the functions as a set without reference to their sequence.

FSL is an all-purpose language used to specify functions, to define and manipulate data and to enter queries. A query expressed in FSL goes through three stages and a number of iterations among them to produce a response:

a. the query is interpreted as a function to determine its input data requirements;
b. for each data item required but not stored explicitly, the appropriate function is retrieved from the function base;
c. the query is executed as a function on the retrieved data base. The result of this process is then displayed to the query programmer.
The basic FSL statement involves an APL expression augmented by a predicate calculus expression. Given an APL expression involving \( n \) vectors of type \( P.t \), the predicate calculus expression augmenting it is interpreted first to produce a response set of \( n \)-tuples. These \( n \)-tuples are then fed into \( n \) vectors of the APL expression for execution.

Predicate calculus expressions consist of well formed formulas (wff) defined as follows:

- Atomic formulas containing constants, variables in the form \( P.t \) where \( P \) is a relation name and \( t \) is a domain name, and arithmetic comparison operators \( =, \neq, \leq, \geq \) are wff.
- If \( A \) and \( B \) are wff, then: \( \neg A, A \lor B, A \land B, A \rightarrow B \) are wff.
- If \( A \) is a wff and \( P \) is a relation name: \( \forall P(A), \exists P(A) \) are wff.
- The formulas obtainable by finitely many applications of \( a, b, \) and \( c \) are wff.

Relation names are used as variable names with quantifiers \( \forall \) and \( \exists \) as indicated in (a) to improve readability. If more than one variable corresponds to a particular relation \( P \), primed variables \( P', P'' \), etc. are used to distinguish variables. Comparison operators are assumed to have precedence over logical operators.

A list of APL operators used in FSL are given in the Appendix. Most of the examples will be restricted to arithmetic operators \( +, - , \times, \div \); maximum \( \Gamma \); minimum \( \Lambda \); size \( p \); concatenation \( ; \); and reduction \( / \); where the last one is used to reduce a vector into a scalar by applying an arithmetic operator repetitively to the elements of the vector. The assignment \( \leftarrow \) operator has a special meaning in FSL. In addition to the usual function of separating the target variable from the body of the expression, it actually forces the FSL expression to be evaluated separately for every element in the target variable to avoid generalized arrays as explained in the following example.

Example:

\[ P.x \leftarrow +/Q.y \times R.z : S \]

where \( S \) is a predicate calculus expression such as:

\[ P.v = Q.w \land R.z = c \]

This FSL expression implies that for every row \( i \) in \( P \), \( [Q.y, R.z] \) pairs satisfying the expression \( S \) are retrieved and fed into APL vectors named \( Q.y \) and \( R.z \). Multiplication of these two vectors and summation of all the elements in the product vector produce the \( i \)-th value in the vector \( P.x \). The number of elements in vectors \( Q.y \) and \( R.z \) are different for every element of \( P.x \); hence it is possible to view the total structure as a generalized array where vectors along a particular dimension have varying number of elements (Figure 1). Another difference from standard APL is the ability to treat character strings as single elements rather than vectors.

The following examples illustrate the flexibility and convenience of FSL; given the information base of a manufac-
Figure 1—An FSL expression visualized as a generalized array and interpreted as a set of simple arrays.

The following queries and functions are first expressed in English, then in FSL:

a. Compute the number of suppliers in Milwaukee.
\[ \text{SUPPLIER.s#} \text{. location} = \text{"MILWAUKEE"} \]

b. Compute the average weight of all parts of type A.
\[ +/ \text{PART.weight} \]
\[ \rho \text{ PART.weight} \]
\[ \rho \text{ PART.type} = \text{"A"} \]

c. SUPPLY.tc is the transportation cost for each supply and depends on the distance, unit transportation cost and the total weight transported.
\[ \text{SUPPLY.tc} \leftarrow \text{ROUTE.distance} \times \text{ROUTE.cost} \times \text{SUPPLY.quantity} \times \text{PART.weight} \times \exists \text{PROJECT} \exists \text{SUPPLIER} \text{SUPPLY.s#} = \text{SUPPLIER.s#} \wedge \text{SUPPLIER.location} = \text{ROUTE.origin} \wedge \text{SUPPLY.j#} = \text{PROJECT.j#} \wedge \text{PROJECT.location} = \text{ROUTE.destination} \wedge \text{SUPPLY.p#} = \text{PART.p#} \]

d. SUPPLY.dr is discount rate applying to each supply and it is equal to the discount rate of the supplier involved, or 5 percent more if the quantity is greater than 100.
\[ \text{SUPPLY.dr} \leftarrow (\text{SUPPLIER.dr} : \text{SUPPLY.s#} = \text{SUPPLIER.s#}) + 0.05 : \text{SUPPLY.quantity} > 100 \]

e. SUPPLY.cost is the cost of each supply and depends on the quantity, price, discount rate and the transportation cost.
\[ \text{SUPPLY.cost} \leftarrow (\text{SUPPLY.quantity} \times \text{PART.price} \times (1 - \text{SUPPLY.dr})) + \text{SUPPLY.tc} \times \exists \text{PROJECT} \exists \text{SUPPLIER} \text{SUPPLY.p#} = \text{PART.p#} \]

f. PROJECT.cost is the cost to date of each project and it is defined as the sum of all supply costs for that project.
\[ \text{PROJECT.cost} \leftarrow +/ \text{SUPPLY.cost} : \text{SUPPLY.j#} = \text{PROJECT.j#} \]

g. Create a relation SUP(p#, j#, num) which contains the number of supplies of each part for every project.
\[ \exists \text{SUPPLIER} \text{SUPPLY.s#} = \text{SUPPLIER.s#} \wedge \exists \text{PROJECT} \text{SUPPLY.j#} = \text{PROJECT.j#} \]

h. Which locations are supplying which locations?
\[ \exists \text{SUPPLIER} \text{SUPPLIER.s#} = \text{SUPPLIER.s#} \wedge \text{SUPPLY.j#} = \text{SUPPLY.j#} \]

i. Which locations are supplying which locations exclusively? (supplier supplying only one location)
\[ \exists \text{SUPPLIER} \text{SUPPLIER.s#} = \text{SUPPLY.s#} \rightarrow \exists \text{SUPPLY} \text{SUPPLY.j#} = \text{PROJECT.j#} \]

j. PROJECT.distance is the average distance of supplies supplying each project.
\[ \text{PROJECT.distance} \leftarrow +/ \text{ROUTE.distance} \div \rho \text{ ROUTE.distance} \div \exists \text{SUPPLY} \exists \text{SUPPLIER} \text{PROJECT.location} = \text{ROUTE.destination} \wedge \text{PROJECT.j#} = \text{SUPPLY.j#} \wedge \exists \text{SUPPLIER} \text{SUPPLIER.s#} = \text{SUPPLIER.s#} \wedge \text{SUPPLIER.location} = \text{ROUTE.origin} \]

k. PROJECT.farthest is the farthest location supplying each project.
\[ \text{PROJECT.farthest} \rightarrow \text{ROUTE.origin} \]

From the collection of the Computer History Museum (www.computerhistory.org)
[ROUTE.distance i f/RUTE.distance]: ∃SUPPLY
SUPPLIER PROJECT.location =
ROUTE.destination ∧ PROJECT.j# = SUPPLY.j#
∧ SUPPLY.s# = SUPPLIER.s# ∧
SUPPLIER.location = ROUTE.origin
or:
PROJECT.farthest ← ROUTE.origin: ∃SUPPLY
SUPPLIER ROUTE PROJECT.j# =
SUPPLY.j# ∧ SUPPLY.s# = SUPPLIER.s# ∧
SUPPLIER.location = ROUTE.origin ∧
PROJECT.location = ROUTE.destination ∧
SUPPLIER.loc = ROUTE.origin →
ROUTE.distance ≥ ROUTE'.distance).

USER FRIENDLINESS

The convenience to the final user of an information system is a major concern in language design. Some previous research reported behavioral work to compare different syntaxes in terms of user preferences. Syntax of a language undoubtedly plays a role in determining the level of convenience to and acceptance by the final user; however one has to keep in mind that different types of users with different skills and needs may have different preferences of syntax. A syntax geared toward the naive user may very well turn out to be very cumbersome and inconvenient for more sophisticated users as probably is the case in QUERY by EXAMPLE.

A technique which invariably decreases the workload on the final user is abstraction. Abstraction involves decreasing the information content of a query, hence decreasing the effort required to form a query. This process can be called 'incomplete querying' since the queries acceptable by the system do not necessarily have all the information necessary to respond to the query. Incomplete querying obviously requires the system to provide the missing information to complete a query and it is accomplished by storing query segments—which may be queries themselves—and assigning them abstract names. A user, then, is free to use these abstract names to refer to query segments and build a complex query from these segments. Iteration in this process is permissible and actually is the real source of power.

Information Base draws heavily on the concept of abstraction by storing functional relationships as functions, and referring to these functions by name in forming more complex functions. A simple example of this was provided in section 2 where the previously defined variable SUPPLY.cost was used in a query and the definition of SUPPLY.cost was retrieved by the system from the information base. It may be claimed that the user has to provide all function definitions at some point in time; however even if the same user has to provide all function definitions, division of the task into independent segments simplifies the work to a great extent. An information Base Administrator may be employed to maintain the structure and given the task of defining the functions to further improve the situation.

Following the same philosophy, we will name predicate calculus expression segments as functions and keep them in function storage to further simplify queries. Natural candidates for this process are segments involving natural joins of two or more relations. Intelligent choice of names results in expressions close to English expressions in terms of ease of interpretation and construction by humans. The names used here start and end with relation names and imply an access path between those two relations:

\[
\begin{align*}
\text{SUPPLY\_CONTAINING\_PART} & \equiv \text{SUPPLY.p#} = \text{PART.p#} \\
\text{SUPPLY\_RECEIVED\_BY\_PROJECT} & \equiv \text{SUPPLY.j#} = \text{PROJECT.j#} \\
\text{SUPPLY\_BY\_SUPPLIER} & \equiv \text{SUPPLY.s#} = \text{SUPPLIER.s#} \\
\text{SUPPLIER\_SUPPLYING\_PROJECT} & \equiv \exists \text{SUPPLY} \\
& \quad \text{SUPPLY\_BY\_SUPPLIER} \land \\
& \quad \text{SUPPLY\_RECEIVED\_BY\_PROJECT} \\
\text{PROJECT\_ON\_ROUTE} & \equiv \text{SUPPLIER.loc} = \text{ROUTE.origin} \\
\text{SUPPLY\_USING\_ROUTE} & \equiv \exists \text{PROJECT SUPPLIER} \\
& \quad \text{SUPPLY\_RECEIVED\_BY\_PROJECT} \land \\
& \quad \text{PROJECT\_ON\_ROUTE} \land \\
& \quad \text{SUPPLY\_BY\_SUPPLIER} \land \\
& \quad \text{SUPPLIER\_ON\_ROUTE} \\
\text{PROJECT\_USING\_ROUTE} & \equiv \exists \text{SUPPLY SUPPLIER} \\
& \quad \text{PROJECT\_RECEIVING\_SUPPLY} \land \\
& \quad \text{SUPPLY\_BY\_SUPPLIER} \land \\
& \quad \text{SUPPLIER\_ON\_ROUTE} \land \\
& \quad \text{PROJECT\_ON\_ROUTE} \\
\end{align*}
\]
Procedure Independent Design of Information Systems

Given the above function base, some of the examples of section 3 are simplified as follows:

a. \[ \text{SUPPLY}\_tc \leftarrow \text{ROUTE}\_distance \times \text{ROUTE}\_cost \times \text{SUPPLY}\_quantity \times \text{PART}\_WEIGHT: \]
   \[ \text{SUPPLY}\_USING\_ROUTE \land \text{SUPPLY}\_CONTAINING\_PART \]

b. \[ \text{SUPPLY}\_dr \leftarrow (\text{SUPPLIER}\_dr: \text{SUPPLY}\_BY\_SUPPLIER) + 0.05: \text{SUPPLY}\_quantity \gt 100 \]

c. \[ \text{SUPPLY}\_cost \leftarrow (\text{SUPPLY}\_quantity \times \text{PART}\_price \times (1 - \text{SUPPLY}\_dr)) + \text{SUPPLY}\_tc: \]
   \[ \text{SUPPLY}_\text{CONTAINING}\_PART \]

d. \[ \text{PROJECT}\_distance \leftarrow +/\text{ROUTE}\_distance \div p \]
   \[ \text{PROJECT}\_USING\_ROUTE \]

e. \[ \text{PROJECT}\_farthest \leftarrow \text{ROUTE}\_origin \]
   \[ \text{ROUTE}\_distance \text{ROUTE}\_USING\_ROUTE \]

CONCLUSIONS

A non-procedural design language is extremely useful in providing the application system designer with means to communicate the system requirements to other designers or to the machine without efficiency considerations. Procedure has to be introduced to the information system by someone—or preferably some system—who has the full knowledge of physical data structure, operating system, and the hardware configuration, and it has to be introduced with efficiency as the only concern. If there is interdependence between requirements and the efficiency attainable to meet those requirements, an iterative approach should be taken rather than combining the two tasks, simply because of the size, effort, and the number of different skills involved in systems development process. The interpretation of FSL expressions and automated implementation of systems require automated access path selection and automated generation of application algorithms. Optimization of this process is beyond the scope of this paper. The current implementation work is geared toward the generation of feasible implementations directly from the information base design. The implementation language is naturally APL.

BIBLIOGRAPHY


APPENDIX

**APL operators in FSL**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Monad</th>
<th>Dyadic</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus</td>
<td>Add</td>
</tr>
<tr>
<td>-</td>
<td>Minus</td>
<td>Subtract</td>
</tr>
<tr>
<td>×</td>
<td>Signum</td>
<td>Multiply</td>
</tr>
<tr>
<td>÷</td>
<td>Reciprocal</td>
<td>Divide</td>
</tr>
<tr>
<td>∑</td>
<td>Ceiling</td>
<td>Maximum</td>
</tr>
<tr>
<td>▽</td>
<td>Floor</td>
<td>Minimum</td>
</tr>
<tr>
<td>+</td>
<td>Exponential</td>
<td>Power</td>
</tr>
<tr>
<td>○</td>
<td>Natural Logarithm</td>
<td>Logarithm</td>
</tr>
<tr>
<td></td>
<td>Magnitude</td>
<td>Residue</td>
</tr>
<tr>
<td>!</td>
<td>Factorial</td>
<td>Combinations</td>
</tr>
<tr>
<td>?</td>
<td>Roll</td>
<td>Deal</td>
</tr>
<tr>
<td>ρ</td>
<td>Size</td>
<td>Reshape</td>
</tr>
<tr>
<td>i</td>
<td>Ravel</td>
<td>Catenate</td>
</tr>
<tr>
<td>‹</td>
<td>Index generation</td>
<td>Index of</td>
</tr>
<tr>
<td>φ</td>
<td>Transpose</td>
<td>Dyadic transpose</td>
</tr>
<tr>
<td>E</td>
<td>Transpose</td>
<td>Membership</td>
</tr>
<tr>
<td>↑</td>
<td>Take</td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td>Drop</td>
<td></td>
</tr>
<tr>
<td>φ</td>
<td>Reverse</td>
<td>Rotate</td>
</tr>
<tr>
<td>θ</td>
<td>Reverse</td>
<td>Rotate first</td>
</tr>
<tr>
<td>/</td>
<td>Literal value</td>
<td></td>
</tr>
<tr>
<td>[ ]</td>
<td>Reduction Compression</td>
<td></td>
</tr>
</tbody>
</table>

For a detailed explanation of how to use these operators, reader is referred to any APL manual.