A Dataflow Graphical Language for Database Applications

Bogdan D. Czejdo and Ralph P. Tucci

Department of Mathematics and Computer Science, Loyola University, New Orleans, U.S.A.

In this paper we discuss a graphical language for information retrieval and processing. A lot of recent activity has occurred in the area of improving access to database systems. However, current results are restricted to simple interfacing of database systems. We propose a graphical language for specifying complex applications.

1. Introduction

There have been various attempts to design interfaces for database systems such that complex and ad hoc requests can be specified by pictures. A variety of interactive graphical specification languages have been described in the literature [Y. VASSILIOU, M. JARKE 1984] [J. LARSON 1986] [N. C. SHU 1986]. Many of these languages are based on semantic models such as the Entity-Relationship (ER) model [Z. Q. ZHANG, A. O. MENDELZON 1983] [R. ELMASRI, J. LARSON 1985] [B. CZEJDO, et al. 1991]. These languages can provide valuable assistance in formulating database queries and updates. The proposed solutions were very restrictive, however, and did not have the computational power of a general-purpose programming language.

In this paper we propose to extend graphical database languages to accommodate more complex queries. As a result of our extensions, such queries can be completely specified by pictures.

We have used an Object-Relationship (OR) model [D. W. EMBLEY, et al. 1992] for graphical query specification [B. CZEJDO, et al. 1991]. The paper is based on results described in [B. CZEJDO, R. P. TUCCI 1992] and in [B. CZEJDO, R. P. TUCCI 1993]. A major goal of this paper is to graphically specify information retrieval which involves set operations, computations on sets, and recursively defined views involving sets. We provide a formal definition of such graphical queries.

2. An Informal Description of the OR Model

We use a semantic data model called the Object-Relationship (OR) model [D. W. EMBLEY, et al. 1992], which is based on the Entity-Relationship (ER) model. The OR model views the world as consisting of objects and relationships among those objects. Objects and relationships are classified into object sets and relationship sets respectively. The OR model differs from the original ER model in that it treats attributes as objects, and because it allows a richer set of relationship types among the objects. Figure 1 shows an OR model diagram which describes part of an information system for a company.

An object exists in the real world and is distinguished from other objects. An object set is a set of objects that have some common properties. For example, in Figure 1, the object set EMPLOYEE contains all employees in the company.

A relationship is a meaningful connection among objects. A collection of relationships pertaining to the same object sets can constitute a relationship set. For example, in Figure 1, the relationship set HAS_SALARY connects the object sets EMPLOYEE and SALARY.

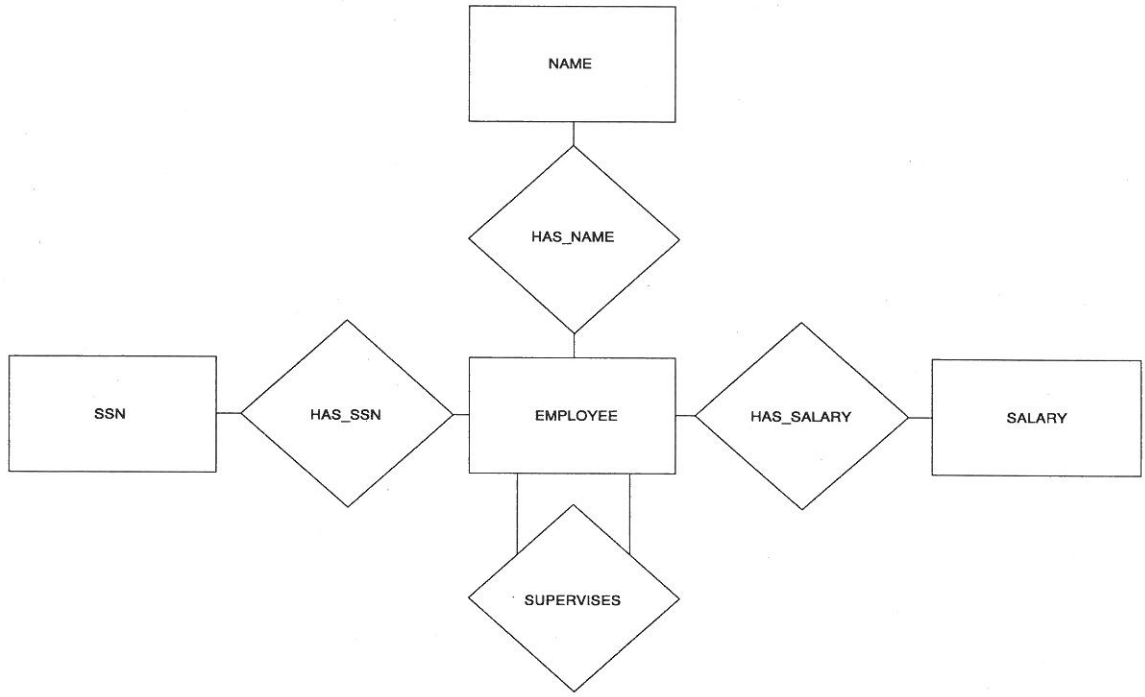

Fig. 1. An OR Model Representing a Partial Information System for a Company

3. A Formal Description of the OR Model

In this section we present a formal model that defines both a schema for the OR model and a graphical representation of the schema. The formal model is a pair **(O, R)** where **O** is a set of object sets and **R** is a set of relationship sets. An *object set* is a pair consisting of a set of objects and an object set descriptor. An *object set descriptor* is a pair (NAME, DOMAIN) described as follows:

(1) NAME is the name of the object set.

(2) DOMAIN is the domain of each object in the set.

We represent an object set graphically by a rectangle containing the NAME of the object set.

A *relationship set* is a pair consisting of a set of relationships and a relationship set descriptor. A *relationship set descriptor* is a pair (NAME, SET_OF_OBJECT_SETS) described as follows:

(1) NAME is the name of the relationship set.

(2) SET_OF_OBJECT_SETS is the set of object sets which participate in the relationship set. We assume that a relationship set relates two or more object sets. (These object sets need not be distinct; if they are not distinct, then the relationship set is recursive.)

We represent an relationship set graphically by a diamond containing the NAME of the relationship set. The diamond is connected to the boxes representing the object sets which participate in the relationship set.

As an example, consider the diagram in Figure 1. The object set descriptor for EMPLOYEE is (EMPLOYEE, string) and for SALARY is (SALARY, integer), and the relationship set descriptor for HAS_SALARY is (HAS_SALARY, {EMPLOYEE, SALARY}).

4. An Informal Description of some Simple Graphical Queries

We now show how to extend the OR model of the previous sections to allow us to construct simple graphical queries. The idea is that object sets represent data and relationship sets represent operators which act on the data.

Suppose, for example, that our EMPLOYEE object set described in Figure 1 contains identifiers for employees such as E_1, E_2, and E_3. The object set SALARY contains two data items, namely 40,000 and 45,000. The relationship

set HAS_SALARY relates all elements in EMPLOYEE with the elements in SALARY, for example, (E_1, 45,000) (E_2, 45,000) and (E_3, 40,000).

In order to retrieve and update the data we need to traverse the graph. We start the traversal at some nodes which we refer to as root nodes, and we end the traversal when we reach certain nodes which we refer to as leaf nodes. Graphically we identify a root node by shading it and we identify a leaf node by using bold characters. For example, in Figure 2, EMPLOYEE is a root node and SALARY is a leaf node.

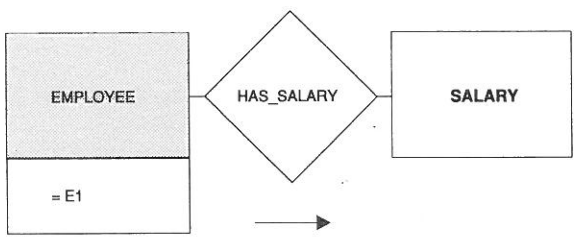

Fig. 2. A Graphical Query to Get the Salary of Employee E1

The arrow under the relationship set HAS_SALARY denotes the order of traversal. For example, in Figure 2, we are interested in going from EMPLOYEE to SALARY, and therefore we specify this direction by the arrow.

The selection condition can be added below any object set. For example, in Figure 2, we are interested only in employee E1, and therefore we restrict the object set EMPLOYEE to only one object.

The root node, the selection, the direction, and the leaf node together specify an unambiguous traversal of the data in this database; this traversal gives us access to all pertinent data.

5. A Formal Description of Graphical Queries

In this section we formally present some extensions to the OR model that allows us to specify queries graphically. The formal model for the graph of a query is a pair **(O, R)** where **O** is a set of object sets and **R** is a set of relationship sets. An *object set* is a pair consisting of a set of object instances and an object set descriptor. An *object set descriptor* is a triple (NAME, DOMAIN, TYPE) described as follows:

(1) NAME is the name of the object set.

(2) DOMAIN is the domain of each object in the set.

(3) TYPE is a set of descriptors for the object set. It can contain the following values:

(a) ROOT if the object set contains the input to a query.

(b) LEAF if the object set contains the output to a query.

(c) GROUPED if the computation is performed separately for each object in the object set.

(d) SELECTION CONDITION

We represent an object set graphically by a rectangle containing the NAME of the object set. A ROOT object set is shaded as in Figure 2. A LEAF object set is denoted with bold characters. A GROUPED object set is denoted by attaching the word GROUPED to the box representing the object set. A SELECTION CONDITION is indicated in a box attached to the rectangle representing the object set. For most of the operations, the ordering is unimportant. In those cases when ordering is important, we assume that there is some ordering among the objects in an object set.

A *relationship set* is a pair consisting of a set of relationships and a relationship set descriptor. A *relationship set descriptor* is a 6-tuple (NAME, SET_OF_OBJECT_SETS, TYPE, VIEW, SET_OF_OUTPUT_OBJECT_SETS, PRIORITY), where NAME and SET_OF_OBJECT_SETS are defined as for relationship sets in the original OR model. The TYPE is either PRIMITIVE DATABASE, PRIMITIVE COMPUTATIONAL, or NON-PRIMITIVE. The VIEW is either the empty set, for a primitive relationship set, or a subset of the formal model for a non-primitive relationship set. The SET_OF_OUTPUT_OBJECT_SETS is a set consisting of the object sets in which values are returned by either a database operation or a computation. The PRIORITY is a positive integer indicating the order in which the relationship set is applied. If there is no priority, then relationships are applied concurrently. A relationship set is denoted graphically as in the original OR

model. The set of output object sets is denoted graphically by one or more arrows next to the relationship set; these arrows point to the output object sets.

6. Primitive Relationship Sets to Represent Computations

In order to retrieve and process information, we need both stored (primitive) database relationships and relationship sets which represent computations. We now list and describe primitive relationship sets which represent computations. An operator is *primitive* if it is in the following table.

TYPE	VALUES
Arithmetic	+ — */COMPUTE_SQUARE COMPUTE_SQUARE_ROOT
Relational	< <= > >= = <>
Logical	AND, OR, NOT
Aggregation	SUM, COUNT
List	HAS_FIRST_ELEMENT, HAS_ELEMENT, HAS_ONE_ELEMENT, HAS_TAIL, IS_APPENDED_TO, IS_CONCATENATED
Set	UNION, INTERSECTION

Any other operator is *non-primitive*. We will see in the next section how to build a non-primitive operator out of primitive operators.

Each of the primitive operators above can be represented as a relationship set. If an operator is binary, then it is represented by a ternary relationship set as in Figure 3 (a), which represents the formula RESULT := BINARY OPERATOR (FIRST OPERAND, SECOND OPERAND). There are two cases of each binary operation. The first case occurs when the FIRST OPERAND contains a single value and the SECOND OPERAND contains a set of values. In this case, a single value is repetitively used in the computation. The second case occurs when both FIRST OPERAND and SECOND OPERAND contain sets of values. In this case, the operation is performed on pairs of elements from each object set. To simplify the discussion, we will use the same symbol for the operator in both cases; the exact meaning of the symbol will be clear from the context.

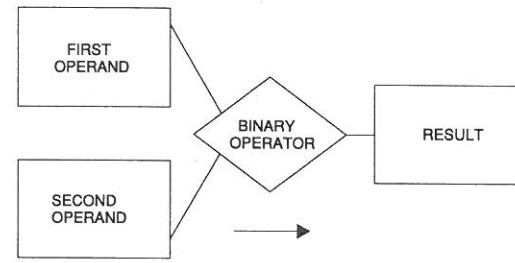

Fig. 3 (a). A Primitive Ternary Relationship Set Representing a Binary Operator

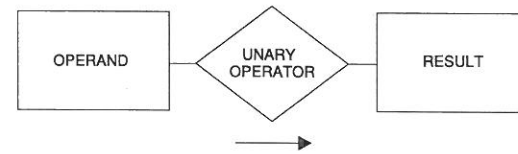

Fig. 3 (b). A Primitive Binary Relationship Set Representing a Unary Operator

If an operator is unary, then it is represented by a binary relationship set as in Figure 3 (b), which represents the formula RESULT := UNARY OPERATOR (OPERAND).

For example, the formula C := A + B can be represented graphically by taking Figure 3 (a) and substituting A for FIRST OPERAND, B for SECOND OPERAND, C for RESULT, and + for BINARY OPERATOR. This action would result in having the triple (A, B, A + B) added to the relationship set "+".

7. Combining Primitive Relationships

We say that a *simple graph* is a graph which contains

(1) a single primitive relationship set together with a direction and all related object sets;

(2) one or more roots;

(3) one or more leaves.

Two graphs G_1 and G_2 are *connected* by identifying one or more object sets from G_1 with one or more object sets from G_2. Connecting two graphs allows us to combine two given queries to specify a more complicated query. For example, suppose we wish to represent the computation X := A + B + C. Using the diagram in Figure 3 (a) twice we represent the sums $TEMP_1$:= A + B and $TEMP_2$:= C + D. Then we identify $TEMP_1$ with D and rename $TEMP_2$ as X to get

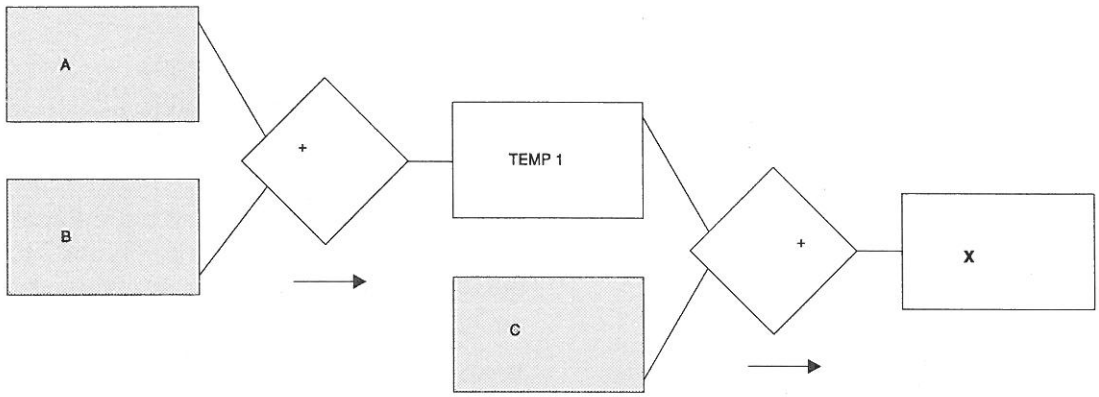

Fig. 4. A Graphical Query to Compute X := A + B + C and Define a New Relationship Set

the graph in Figure 4. We require the user to specify that A, B, and C are roots and that X is a leaf. The resulting graph represents the desired computation.

8. Traversal and Synchronization of Query Graphs

We traverse a query graph by traversing all paths which start at a root and terminate in a leaf. We assume for the moment that any graph can be redrawn as a directed acyclic graph, and we will discuss traversal of graphs with cycles later. The traversal is performed for all object sets in the roots. If there is more than one root, then we can start at all roots and traverse paths from different roots simultaneously, until the paths intersect, at which point synchronization is required. We assume that each relationship set waits for incoming data, and this serves as a synchronization mechanism.

Many queries involve synchronization of two or more object sets at a time. The average is a good example of such a computation. Figure 5 shows a graphical query for such a computation.

The root can be any set of numerical values. First we compute the NUMBER_OF_ITEMS by applying the computational relationship set COUNT. In parallel we compute the SUM_OF_ITEMS using the SUM operator. Next, we divide the value in SUM_OF_ITEMS by the value in NUMBER_OF_ITEMS. There is a need to synchronize these operations in such a way that division is applied only when SUM_OF_ITEMS and NUMBER_OF_ITEMS are already computed.

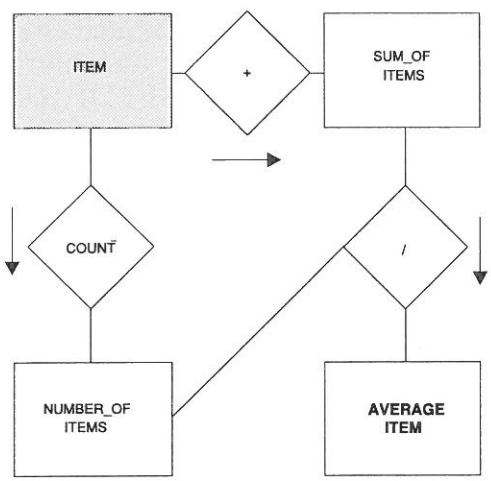

Fig. 5. A Graphical Query to Compute Average

9. Defining Non-primitive Relationship Sets

The way to define a non-primitive relationship set (view) is to draw a graph to represent a query. As for any other query, this graph consists of a root (or roots), leaves, and relationships. The SET_OF_OBJECT_SETS of the newly defined relationship set should be equal to the union of all roots and leaves of the defining view. The TYPE of this relationship set is NON-PRIMITIVE. The VIEW is the graphical query. The SET_OF_OUTPUT_OBJECT_SETS of the newly defined relationship set should be equal to the set of all leaves of the defining view.

For example, we can define a non-primitive relationship set EMPLOYEE_HAS_SSN_AND_SAL which links directly an employee's SSN and

SALARY, as shown in Figure 6 (a). The definition of this relationship set is given in Figure 6 (b). This definition gives us the ability to specify queries simply, such as the query to retrieve all SALARY's of EMPLOYEES with a valid SSN.

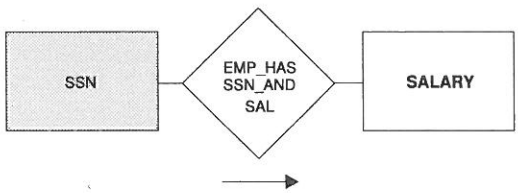

Fig. 6 (a). A Query to Retrieve Employee's Salary Which Uses a Non- primitive Relationship Set EMP_HAS_SSN_AND_SAL

Defining non-primitive relationships allows us to enforce modularity in our graphical queries. Hence, the language we are describing is structured. Having shown how to specify operators by using relationship sets, we will use the terms "operator" and "relationship set" interchangeably throughout the rest of the paper.

10. Explicitly Recursive Graphical Queries

In information processing the need arises for recursion. As an example, let us consider the object set EMPLOYEE containing the elements E_1, E_2 and E_3.

The relationship set SUPERVISES consists of the following pairs

$$(E_1,\ E_2)$$
$$(E_2,\ E_3)$$

and the relationship set HAS_SALARY consists of the pairs

$$(E_1,\ 45,000)$$
$$(E_2,\ 45,000)$$
$$(E_3,\ 40,000).$$

As an example of a recursive graphical query, let us consider the diagram in Figure 7 to get all the salaries of the given employees and those under their supervision.

When the definition of a relationship set includes the relationship set itself, we say that the query containing the relationship set is *explicitly recursive*. Figure 7 represents an explicitly recursive query.

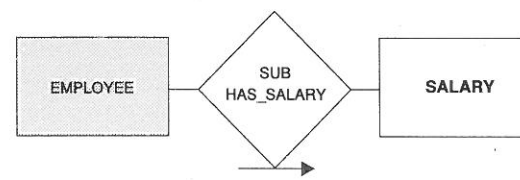

Fig. 7 (a). A Graphical Query to Get All Salaries of Given Employees and Those Under Their Supervision

For Figure 7, the traversal starts from the object set EMPLOYEE. Then all connected relationship sets are considered. The relationship set HAS_SAL places the salary of the given employee into the object set EMP_SALARY. The traversal of this path cannot continue until all input object sets for the relationship set UNION are available. The relationship set SUPERVISES places all subordinate employees into SUB_EMPLOYEES, and the relationship set SUB_HAS_SALARY places all their salaries into SUB_SALARY. Finally, the union of the values in SUB_SALARY and EMP_SALARY is computed and placed into SALARY.

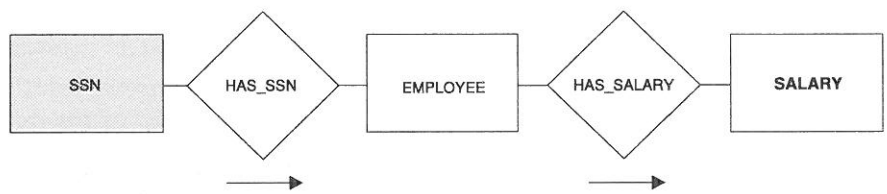

Fig. 6 (b). A Graphical Query to Define the Relationship Set EMP_HAS_SSN_AND_SAL

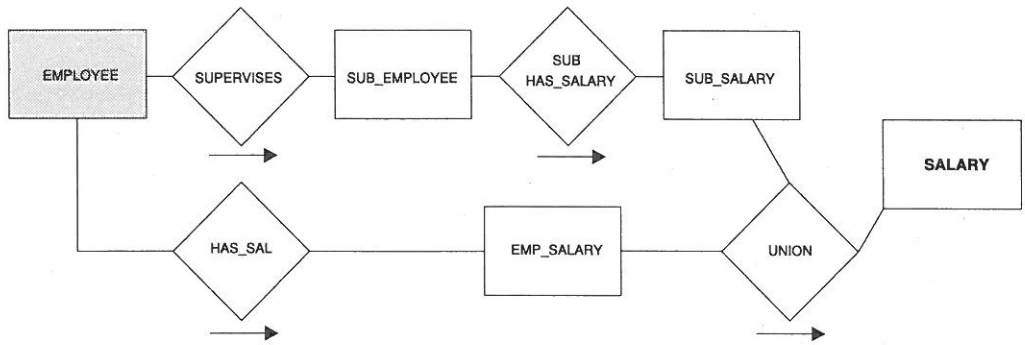

Fig. 7 (b). A Graphical Query Defining a Non-primitive Relationship Set SUB_HAS_SALARY

11. Implicitly Recursive Graphical Queries

A recursive relationship set is a relationship set which relates an object set with itself. When we use such a relationship set in a query graph, we call the query *implicitly recursive.*

The traversal rules do not change if we have a program graph with recursive relationships. Suppose an object set O is related recursively to itself, and suppose that the object subset O_1 in O is to be processed. The traversal of the graph starts with O_1, and eventually a new object subset O_2 of O is reached, where the objects in O_2 are related to objects in O_1. The traversal then continues with O_2. Hence, if an object set is related recursively with itself, we simply relate objects in the same object set and apply traversal recursively for them also.

As an example, let us consider an object set EMPLOYEE from Figure 1 related to itself by the relationship set SUPERVISES. We can construct a graphical program equivalent to that discussed in previous section, as shown in Figure 8, to get all the salaries of given employee and those under his supervision.

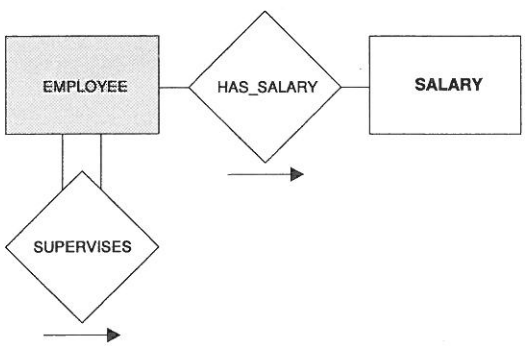

Fig. 8. A Graphical Program to Get all the Salaries of a Given Employee and Those Under His Supervision

Let us assume that we have the data from previous section. The traversal starts from the component E_1 of EMPLOYEE. Then all connected relationship sets are considered. The relationship HAS_SALARY places a single object 45,000 into the object set SALARY. The object set SALARY is the last component of the path. The relationship SUPERVISES places a single object E_2 into the object set EMPLOYEE. Then the traversal is invoked recursively for this new object. As a result, again all connected relationship sets are considered. The relationship HAS_SALARY inserts the object 45,000 into the object set EMPLOYEE. Applying the relationship SUPERVISES places the object E_3 into EMPLOYEE. Then the traversal is invoked recursively for this new object. As a result, again all connected relationship sets are considered. The relationship HAS_SALARY inserts the object 40,000 into the object set EMPLOYEE. Applying the relationship SUPERVISES returns no objects; therefore, the traversal halts with the three objects 45,000, 45,000, and 40,000 placed into the output object set.

12. Examples of Recursive Programs for N!

Let us compare two methods of defining a program to compute N!. Figure 9 (a) contains such a graphical program.

The first definition of a relationship FACTORIAL to compute N! can be given using explicit recursion, as shown in Figure 9 (b).

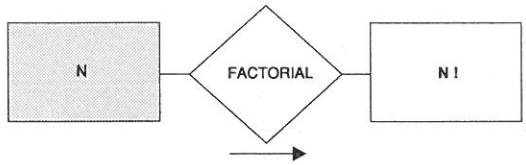

Fig. 9 (a). A Graphical Program to Compute N!

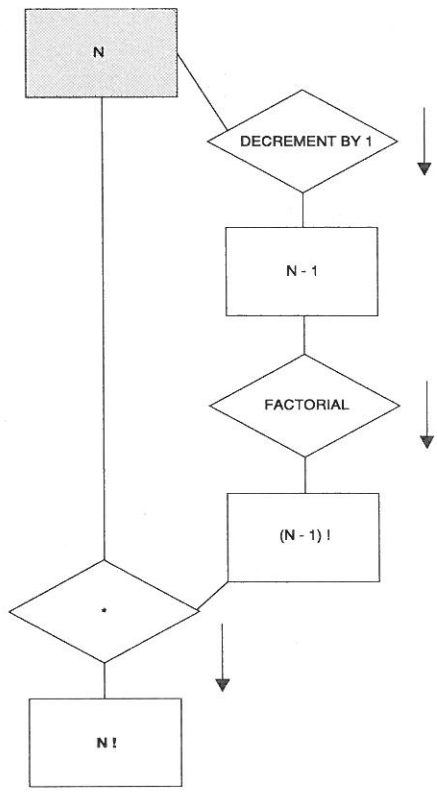

Fig. 9 (b). The Definition of the Relationship FACTORIAL Using Explicit Recursion

This diagram includes four object sets N, N–1, (N–1)!, and N!, each of which has the set of positive integers as domain, and three relationship sets DECREMENT_BY_1, FACTORIAL and *. We assume that the relationship set DECREMENT_BY_1 is defined in such a way that it returns the value of N decreased by 1, but only for N>1. In case N = 1, no value would be placed into N–1, and this terminates the recursion. The relationship FACTORIAL computes the factorial of N–1. Finally, N is multiplied by the value in (N–1)! In case (N–1)! contains no elements, the relationship * returns the value of N in N!

A second way to define the relationship FACTORIAL is to use implicit recursion. This method requires a recursive relationship, as shown in Figure 9 (c).

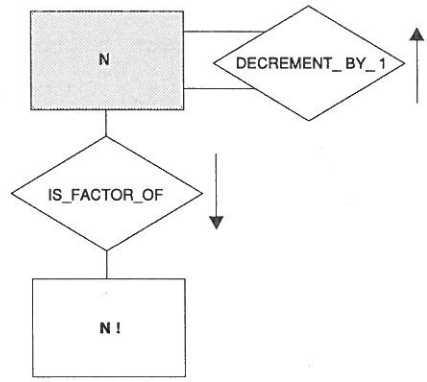

Fig. 9 (c). The Definition of the Relationship FACTORIAL Using Implicit Recursion

The traversal of the diagram is performed as follows. Let us assume the root N contains one element, say 3. The traversal starts from the element 3 of N. The relationship IS_FACTOR_OF inserts the object 3 into the object set N!. This is the last component of the path, so this part of the traversal is complete. As a result of applying the relationship DECREMENT_BY_1, a single object 2 is placed into the object set N. Then the traversal is invoked recursively for this new object. As a result, again all connected relationship sets are considered. The relationship set IS_FACTOR_OF inserts the object 6 into the object set N!. The relationship set DECREMENT_BY_1 places the object 1 into N. Then the traversal is invoked recursively for this new object. As a result, again all connected relationship sets are considered and the relationship set IS_FACTOR_OF inserts the object 6 into the object set N!. The relationship set DECREMENT_BY_1 returns no value, as we assumed before. Hence the traversal is complete, and N! contains the single value 6.

13. Examples of Recursive Programs for Quicksort

As another example, let us present two graphical programs to implement QUICKSORT. These programs are in Figures 10 (a) — (d).

Figure 10 (a) contains a single non-primitive operator which takes a list as input, sorts the list using QUICKSORT, and returns a sorted list.

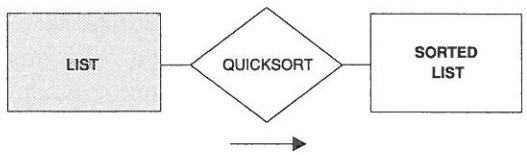

Fig. 10 (a). QUICKSORT

Figure 10 (b) contains the first definition of QUICKSORT using explicit recursion. This figure contains three new non-primitive operators, IS_LEFT_LIST, IS_RIGHT_LIST, and IS_PIVOT. This part of the program takes the original list as input. The list is partitioned as in the standard QUICKSORT algorithm using IS_LEFT_LIST and IS_RIGHT_LIST. Then QUICKSORT is applied recursively to both sublists, and the resulting sorted lists are concatenated together with the pivot value. We assume that the relationship IS_PIVOT returns the list containing the first element of the original list. If the input list is a singleton, then IS_LEFT_LIST and IS_RIGHT_LIST both return the empty set. In this way we make sure that the output lists in LEFT_LIST and RIGHT_LIST are smaller than the input list, so that the algorithm is sure to terminate.

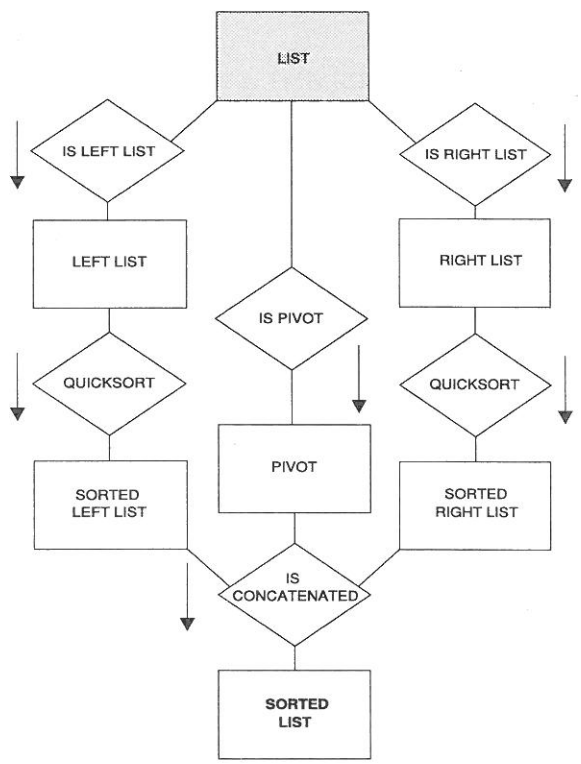

Fig. 10 (b). Definition of the QUICKSORT Relationship Set Using Explicit Recursion

Figure 10 (c) contains a definition of the non-primitive relationship set, IS_LEFT_LIST. First, the initial element of the list is placed into PIVOT, and all the remaining elements of the list are placed into ELEMENT by the relationship HAS_TAIL_ELEMENT; this relationship was constructed by combining the two primitive relationships HAS_TAIL and HAS_ELEMENT. Then the elements of the original list are compared to PIVOT, and those elements which are less than or equal to PIVOT are concatenated to the list in LEFT_LIST by the relationship LESSER_IS APPENDED; this relationship was constructed by combining the two primitive relationships $\leq$ and IS_APPENDED_TO. Therefore, LEFT_LIST contains the list consisting of all elements which are less than or equal to the pivot element, except for the pivot element itself.

Similarly, we can define the non-primitive operator IS_RIGHT_LIST by replacing LESSER_IS_APPENDED by GREATER_IS_APPENDED. The list in RIGHT_LIST contains all elements which are strictly larger than the pivot.

To show how this program works, let us assume that LIST contains one element, say (2 3 5 1 6). The traversal starts from the original list.

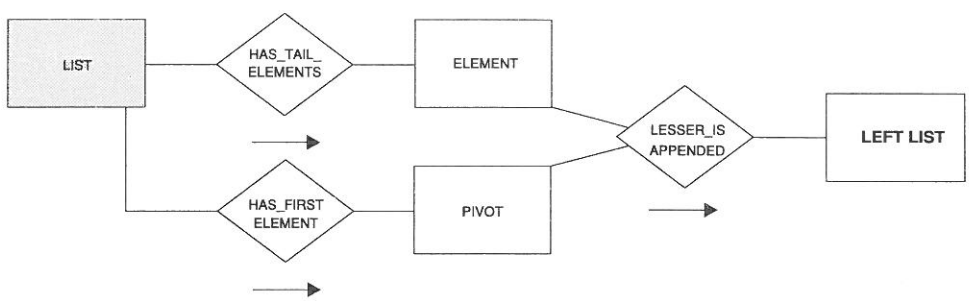

Fig. 10 (c). Definition of the IS_LEFT_LIST Relationship Set

Then all connected relationship sets are considered. The relationship IS_LEFT_LIST returns the list (1) in LEFT_LIST. Then QUICKSORT is applied again, yielding the same list (1) in SORTED_LEFT_LIST. The relationship set IS_PIVOT returns the list (2). The relationship IS_RIGHT_LIST returns the list (3 5 6) in RIGHT_LIST. Then QUICKSORT is applied again, yielding the sorted list (3 5 6) in SORTED_LEFT_LIST. Finally, the lists (1), (2), and (3 5 6) are concatenated and placed into SORTED_LIST. We assume that that SORTED_LEFT_LIST, PIVOT, and RIGHT_LIST are concatenated from left to right. We can also implement QUICKSORT using implicit recursion. In this case we need to assign priorities to relationship sets to guarantee the order of execution. Figure 10 (d) contains the definition of QUICKSORT containing two recursive relationships, IS_LEFT_LIST, and IS_RIGHT_LIST. This program takes the original list as input. If the list has one element, then it is appended to the sorted list in SORTED_LIST; otherwise, the list is partitioned, as in the standard QUICKSORT algorithm, into a left list and a right list, each of which is sorted recursively.

To show how this program works, let us assume that LIST contains one element, say (2 3 5 1 6). The traversal starts from the original list. Then all connected relationship sets are considered in the order specified by the priorities. The relationship IS_LEFT_LIST is applied first, returning the list (1). All connected relationship sets for the list (1) are considered, and (1) is appended to SORTED_LIST. Then the list (2), consisting of the pivot of the original list, is processed. Again, the list (2) is concatenated to the list (1) in SORTED_LIST, yielding (1 2). Now the relationship IS_RIGHT_LIST is applied to the original list, returning the list (3 5 6). The traversal begins again with this new list. The relationship IS_LEFT_LIST returns the empty list. The list (3), consisting of the pivot of the current list, is then appended to the list in SORTED_LIST, yielding (1 2 3). The graph is then traversed again for the remaining list (5 6), etc.

14. Merging Database and Computational Relationships

The semantics of database and computational relationships are identical. Therefore they can be merged freely in order to define proper applications. For example, let us consider a query to compute the standard deviation for salaries for all employees.

Before we compute the standard deviation, we need to be able to compute the average. The graphical query to compute average is shown in Figure 5.

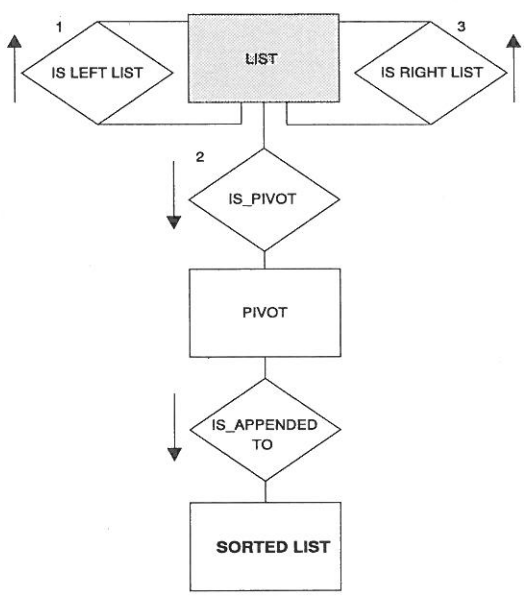

Fig. 10 (d). Definition of the QUICKSORT Relationship Set Using Implicit Recursion

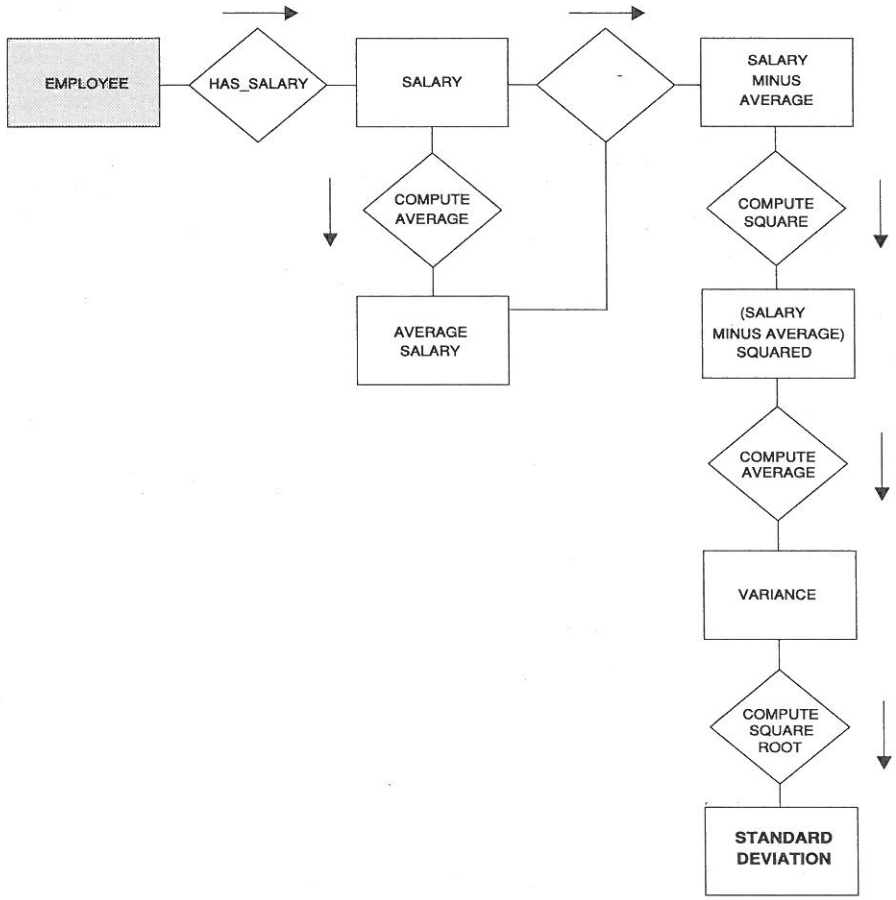

Fig. 11. A Graphical Query to Compute the Standard Deviation of Salaries of All Employees

We compute the average of salaries all employees and store this value in object set AVERAGE SALARY. Then we subtract the single value in AVERAGE SALARY from each salary in SALARY, giving us the values in SALARY MINUS AVERAGE. Next we square each of the values in SALARY MINUS AVERAGE, giving us the values in (SALARY MINUS AVERAGE) SQUARED. We compute the average of these values to get the value in VARIANCE, and we finally take the square root of the value in VARIANCE to get the value in STANDARD DEVIATION. The graphical query is shown in Figure 11.

15. Grouping Operations

The grouping operations are very important in traditional database systems. They allow the user to perform queries for individual objects and relate the result with the individual object. For example, let us consider the query to compute, for each employee, the average salary for himself and all of his subordinates. The query shown in Figure 12 performs this task.

Please note that if we do not group the objects in the employee set, we would obtain the single average value.

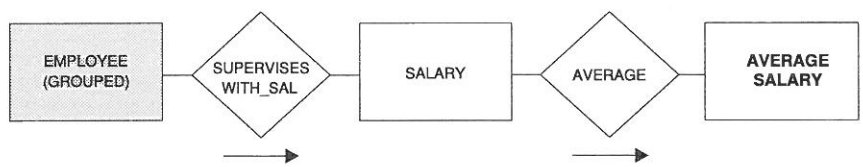

Fig. 12. A Graphical Query to Compute, for each Employee, the Average Salary for All Subordinates

16. Rapid Prototyping Tools

Computer-Aided Software Engineering (CASE) tools have been developed in the last few years to assist in creating large-scale, complex systems and software [M. M. TANIK, R. T. YEH 1989]. The use of such tools can dramatically improve software quality, reduce system costs and improve productivity of software engineers.

The methods described in the previous sections allow us to design new rapid prototyping tools to specify unambiguously the details of the problems to be solved in a graphical way. In the first phase, the semantic model such as shown in Figure 1 can be designed using standard CASE tools. In the second phase, the root and leaf nodes are selected, and the direction of traversal is specified. The resulting graphical query is flexible enough to support both database and complex computational operations.

Summary

In this paper we have discussed a method for accessing database systems based on visual specification. We have proposed an interface based on an Object-Relationship (OR) model. Using such an interface, the user specifies graphical queries by manipulating OR schema diagrams displayed on the terminal screen. The graphical programming interface provides a convenient and dynamically changing frame of reference. Immediate feedback is provided whenever an operator is invalid in the current context.

The proposed visual language is based on data set operations and includes computational relationships and recursion. The language can be naturally implemented as a concurrent system. These features allow for a significant increase in the expressive power and speed of execution of the proposed graphical language in comparison to other database query languages.

References

B. CZEJDO, R. P. TUCCI and D. W. EMBLEY (1991) Graphical Specification of Recursive Queries. In *Advances in Computing and Information — ICCI'91, Lecture Notes in Computer Science 497* (sc F. Dehne, F. Fiala, W. W. Koczkodaj, Ed.), pp. 207–218, Springer-Verlag, Berlin.

B. CZEJDO , R. P. TUCCI (1992) Using an Object-Relationship Model for Rapid Prototyping. In *Proceedings of the 1992 Symposium on Applied Computing*, pp. 299–307, ACM Press, New York.

B. CZEJDO, R. P. TUCCI (1993) A Graphical Language for Information Systems. In Proceedings of the 15th International Conference on Information Technology Interfaces, Pula, Croatia.

R. ELMASRI and J. LARSON (1985) A Graphical Query Facility for ER Databases. In *Proceedings of the 4th International Conference on ER Approach*, Chicago, IL.

D. W. EMBLEY , B. D. KURTZ, S. N. WOODFIELD (1992) Object-Oriented Systems Analysis: A Model-Driven Approach. Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, New Jersey.

J. LARSON (1986) Visual Languages for Database Users. In *Visual Languages* (S. CHANG, T. ICHIKAWA, P. A LIGOMENIDES, Ed.), Plenum Press, New York.

N. C. SHU (1986) Visual Programming Languages: a Perspective and Dimensional Analysis. In *Visual Languages* (S. CHANG, T. ICHIKAWA, P. A. LIGOMENIDES, Ed.), Plenum Press, New York.

VASSILIOU and M. JARKE (1984) Query Languages - a Taxonomy. *In Human Factors and Interactive Computer Systems* (Y. VASSILIOU, Ed.), Ablex, Norwood, 1984.

M. M. TANIK and R. T. YEH (1989) Rapid Prototyping in Software Development. Computer, May, 1989.

Z. Q. ZHANG and A. O. MENDELZON (1983) A Graphical Query Language for ER Databases. In *ER Approach to Software Engineering* (S. DAVIS, Ed.), North Holland, Amsterdam.

Received: October, 1993
Accepted: January, 1994

Contact address:
Bogdan D. Czejdo
Department of Mathematical Sciences
Loyola University
New Orleans, LA 70118, U.S.A.
Telephone U.S.A. (504) 865–2663
Fax (504) 865–3347
E-mail: bumsbdc@music.loyno.edu

BOGDAN D. CZEJDO is a Professor in the Department of Mathematics and Computer Science at Loyola University in New Orleans. He received the M.S. and the Ph.D. degree from Warsaw Technical University in 1972 and 1975, respectively. His research interests include database systems, knowledge based systems and visual languages. He has published more than 50 technical papers in these areas. He is a member of the ACM and the IEEE.

RALPH P. TUCCI is currently an Associate Professor and Chair of the Department of Mathematics and Computer Science at Loyola University in New Orleans. He received a Master's Degree in Mathematics in 1972 and a Ph.D. degree in Mathematics in 1976 from the University of Wisconsin at Milwaukee. He also received a Master's Degree in Computer Science in 1985 from Tulane University. His research interests include abstract algebra and visual languages.