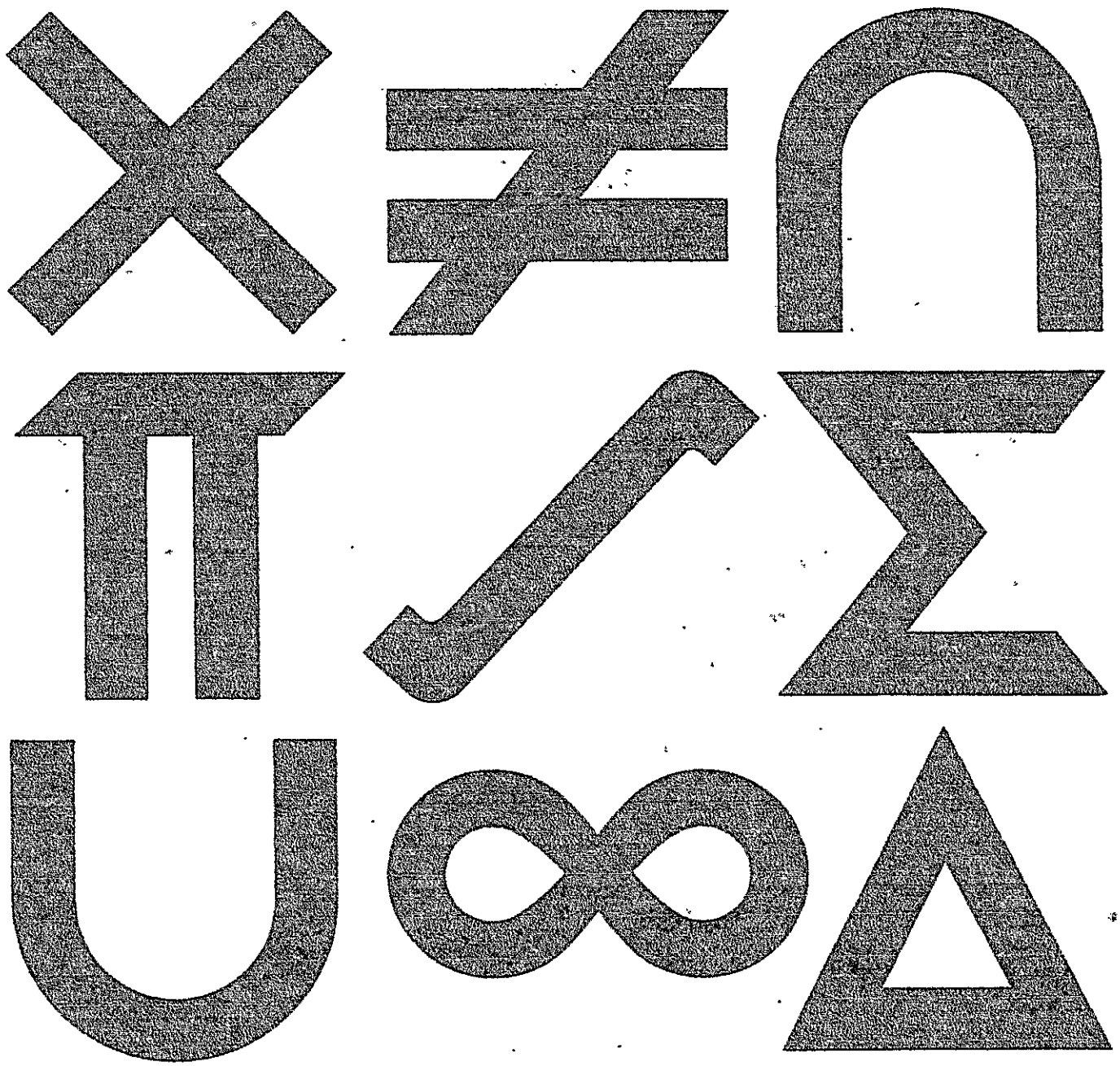
G320-2704, June 1980

AN ATOMIC NETWORK PROGRAMMING LANGUAGE

PAULA S. NEWMAN



1977 LOS ANGELES SCIENTIFIC CENTER REPORTS

+G320-2686 April 1977
T. LANG, C. WOOD & E. B. FERNANDEZ, Database Buffer Paging in Virtual Storage Systems (24 p.)

G320-2687 April 1977
R. C. SUMMERS & E. B. FERNANDEZ, A System Structure for Data Security (41 p.)

+G320-2688 June 1977
T. LANG, E. NAHOURAII, K. KASUGA & E. B. FERNANDEZ, An Architectural Extension for a Large Database Incorporating a Processor for Dick Search (30 p.)

G320-2689 July 1977
C. WOOD, E. B. FERNANDEZ & T. LANG, Minimization of Demand Paging for the LRU Stack Model of Program Behavior (21 p.)

G320-2690 September 1977
B. DIMSDALE, A Geometric Optimization Problem (20 p.)

+G320-2691 September 1977
B. DIMSDALE, Convex Cubic Splines (32 p.)

G320-2692 September 1977
B. DIMSDALE, Convex Cubic Splines II (18 p.)

G320-2693 September 1977
E. B. FERNANDEZ, H. KASUGA, Data Control in a Distributed Database System (26 p.)

1978 LOS ANGELES SCIENTIFIC CENTER REPORTS

G320-2694 March 1978
S. JUROVICS, An Investigation of the Minimization of Building Energy Load Through Optimization Techniques (28 p.)

G320-2695 November 1978
L. LICHTEN, An Approach to Solving Surface Connectivity Problems in Computer-Aided Design (41 p.)

G320-2696 November 1978
C. WOOD, R. C. SUMMERS, E. B. FERNANDEZ, Authorization in Multilevel Database Models (28 p.)

1979 LOS ANGELES SCIENTIFIC CENTER REPORTS

G320-2697 March 1979
P. NEISWANDER, A Review of the National Bureau of Standards Loads Determination Program (NBSLD) (12 p.)

G320-2698 March 1979
C. WOOD & E. B. FERNANDEZ, Authorization in a Decentralized Database System (35 p.)

G320-2699 June 1979
S. A. JUROVICS, Solar Radiation Data, Natural Lighting, and Building Energy Minimization (20 p.)

G320-2700 August 1979
A. INSELBERG, An Integral Equation Arising in a Convective Heat (Mass) Transfer Problem Through a Boundary Layer (19 p.)

G320-2701 September 1979
G. J. SILVERMAN, D. W. LOW, Construction of Optimal Synthetic Weather Data by Convex Combination (14 p.)

1980 LOS ANGELES SCIENTIFIC CENTER REPORTS

G320-2702 March 1980
K. EWUSI-MENSAH, Criteria for Decomposing an Information System Into Its Subsystems for Business Systems Planning (26 p.)

G320-2703 March 1980
K. EWUSI-MENSAH, Computer-Aided Modeling and Analysis Techniques for Determining Management Information Systems Requirements (30 p.)

G320-2704 June 1980
PAULA S. NEWMAN, An Atomic Network Programming Language (29 p.)

G320-2875-5 April 1980
Compiled by KATHERINE HANSON, Abstracts of Los Angeles Scientific Center Reports (104 p.)

The availability of reports is correct as of the printing date of this report.

\+ Appeared in an outside publication. Not available in Scientific Center report form. Please refer to the list of outside publications on the inside back cover for availability of reprints.

• Copies of report are no longer available from the Scientific Center.

IBM LOS ANGELES SCIENTIFIC CENTER REPORT NO. G320-2704

June 1980

AN ATOMIC NETWORK PROGRAMMING LANGUAGE

Paula S. Newman

IBM Corporation
Los Angeles Scientific Center
9045 Lincoln Boulevard
Los Angeles, California 90045

ABSTRACT

There have been many recent studies of approaches to reducing the fragmentation of implementation languages into programming languages, data manipulation languages, command languages, etc. The purpose of this paper is to present some current results of one such study. The results include the definition of a significant part of a language which completely integrates data base accessing into a traditional programming language framework, and the definition and justification of a data model which makes such integration feasible. The model used is an instance of what is called here an "atomic network", a network in which each fact is represented by an individual element.

1.0 INTRODUCTION

An application system is normally implemented using a combination of languages, such as programming languages, data manipulation languages, and languages controlling such systems-related functions as process initiation, inter-process communication, etc. This separation of linguistic function is due to a combination of historical accident, and considerations of portability and standardization. As data-base and systems-related functions become more significant, the fragmented approach becomes less justifiable, and a major source of unnecessary complexity.

Accordingly, there has been considerable investigation into methods of integrating additional functions into existing or new programming languages. Most efforts focus on integrating either data-base-related functions, such as [4, 23, 8], or system-related functions, such as [9, 6], but a few, such as [19, 10] address both issues. (Note: the above references are not intended to be exhaustive, but rather representative).

The atomic network programming language, currently incomplete, is intended to be an instance of the latter direction. Integration of data base manipulation is accomplished by using a single, directly-addressed, data model for all data. Distinctions between classes of data are made on the basis of differences in declared scope and lifespan, rather than on the basis of differences in accessing syntax. Also, traditional HLL statement types are adapted for consistency with a database environment. Integration of systems-related function is obtained by a suitable execution environment definition, and language forms for transaction handling and inter-process communication.

The data model used is a version of what is called here an "atomic network", meaning any kind of network model in which each fact is represented by a separate object. (Atomic networks thus subsume structures called variously "semantic networks", "functional models", "associative networks", "entity-relationship models", etc.) The integration of accesses against such a base into a programming language can be done very smoothly, by relating the "entities" and "associations" of the model to sets of scalars and sets of vectors, and then to the variables and functions of classical high level languages.

In the past, atomic networks have more often been studied a) as foundations for artificial intelligence "knowledge bases", and b) as "conceptual models" of data bases implemented using other structures, than as accessible structure. Some relatively early efforts are represented by [21, 22, 11]. Only quite recently, in such efforts as DAPLEX [23],

TASL [14], FQL [1], FST [2], and the process specification language of [10] has there been an explicit recognition of their potential as a base for high level accessing syntax.

It should be noted, however, that while exploitation of the full chain, i.e., networks --> sets --> language is relatively new, many aspects of the chain are not new. As far back as 1962, the creators of SIMSCRIPT recognized that data associations could be referenced as functions [18]. Furthermore, SETL [20], developed in the early 70's, definitively established the second link in the chain.

The purpose of the paper is twofold. The primary purpose is to introduce the data accessing and algorithmic aspects of the language. While many constructs are shared with the efforts referenced above, especially [23] and [20], the language is unique in the extent of the integration, in the balance achieved between succinctness and readability, and in approaches taken to specific problems. The secondary purpose of the paper is to introduce the model, and to explore, to some extent, the interplay between model and language. While alternative atomic networks, and individual components, have been compared with respect to utility for conceptual modelling [3, 15, 16], there has been little discussion of the implications of those alternatives for accessing language.

We will begin with the discussion of an informal data model, sufficient to motivate the language (section 2). The major procedural constructs of the language are presented in sections 3 through 5. Following this the actual model used is defined and justified (section 6), and some information is provided about data definition (section 7). This slightly inverted order simplifies the presentation, as justification of the model is partially based on linguistic considerations. (The language sections can be reread following the description of the actual model to verify the connection.)

2.0 INFORMAL DATA MODEL

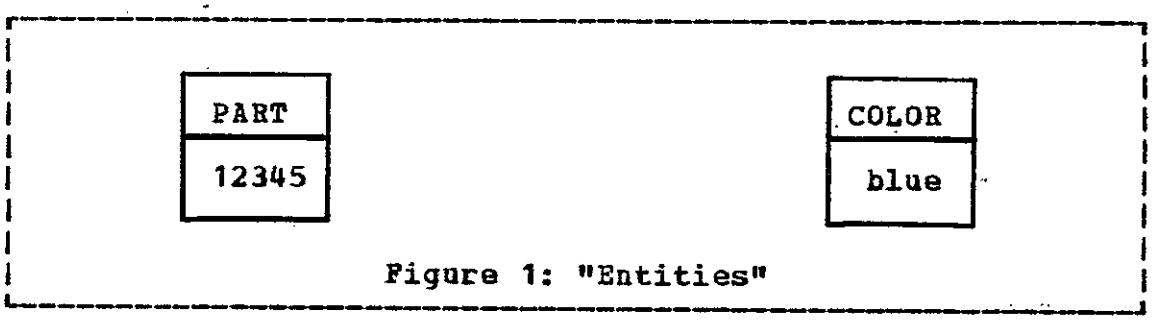


Figure 1: "Entities"

An informal, intuitive data model which might be manipulated by the language is given by the examples of figures 1 through 3. What might be thought of as entities (Figure 1) are represented by nodes identified by a combination of typeset name (PART, COLOR), and member name ("12345", "blue").

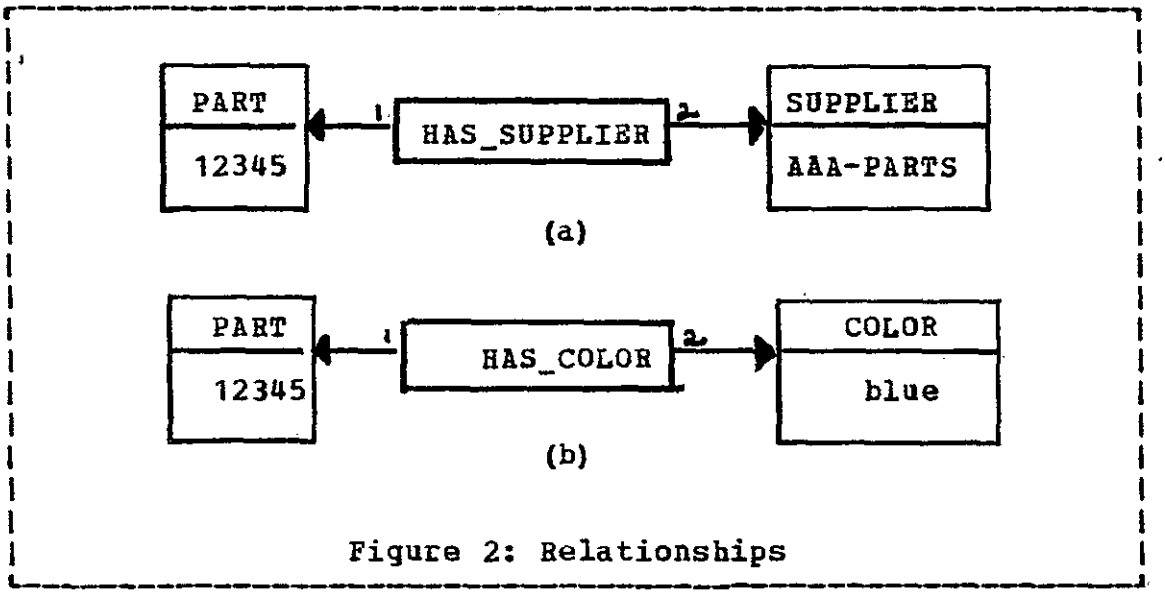


Figure 2: Relationships

Relationships (Figure 2) are represented by relationship nodes identified by a combination of typeset name, and participant identifiers. Numbered arrows emanating from the relationship nodes indicate the ordered participants. Thus Figure 2a represents the "HAS-SUPPLIER" relationship between the PART "12345" in the first role, and the SUPPLIER "AAA-PARTS" in the second. Inverse (permuted) relationships may be defined. No distinction is made between "attribute" relationships and other relationships (Figure 2b). Relationships may participate in other relationships (Figure 3a), and relationships may be n-ary (Figure 3b).

The integrity rules assumed are:

- Names of members within a particular entity typeset must be drawn from a particular predeclared set of names (which may be "infinite").
- All relationships within a particular relationship typeset must have the same degree, and must connect the same types of objects.

It should be remembered that the above model is intended only to make the language examples comprehensible; the actual model used is defined further on.

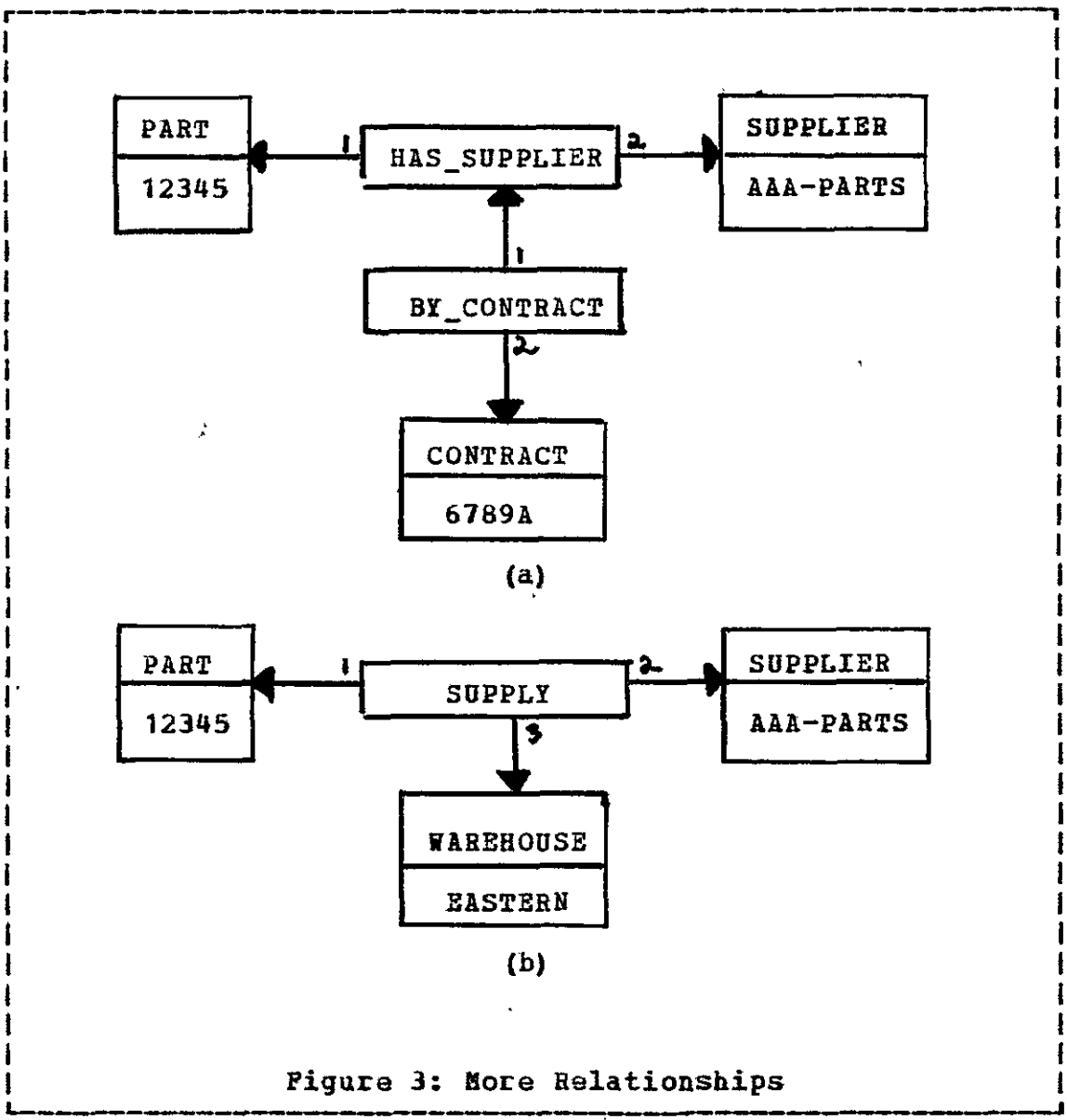


Figure 3: More Relationships

3.0 EXPRESSIONS

Expressions are the most important construct of the language. They are used to express assigned values, repetition specifications, conditions, and on-line queries.

Expressions are built of literals, typeset references, operators, function references, and selection clauses. Each of these constructs will be examined in turn below.

```
"12345"                              literal alpha scalar
25.39                                literal numeric scalar
TRUE                                 literal boolean value
•12345                               literal surrogate
<"12345", "12346">                   literal vector
(_ "12345", "12346"_)                set of scalars
(_<"12345", "AAA_PARTS">_),
  <"12346", "PARTS_INC">_)           set of vectors
```

Figure 4: Literals

Literals. Figure 4 illustrates various types of literals and sets of literals. Note the distinguished parentheses, "(_", used to enclose sets; curly brackets would be preferable, but are often unavailable. (The term "surrogate", introduced in [12], is now fairly well known; its adaptation here is discussed further on.)

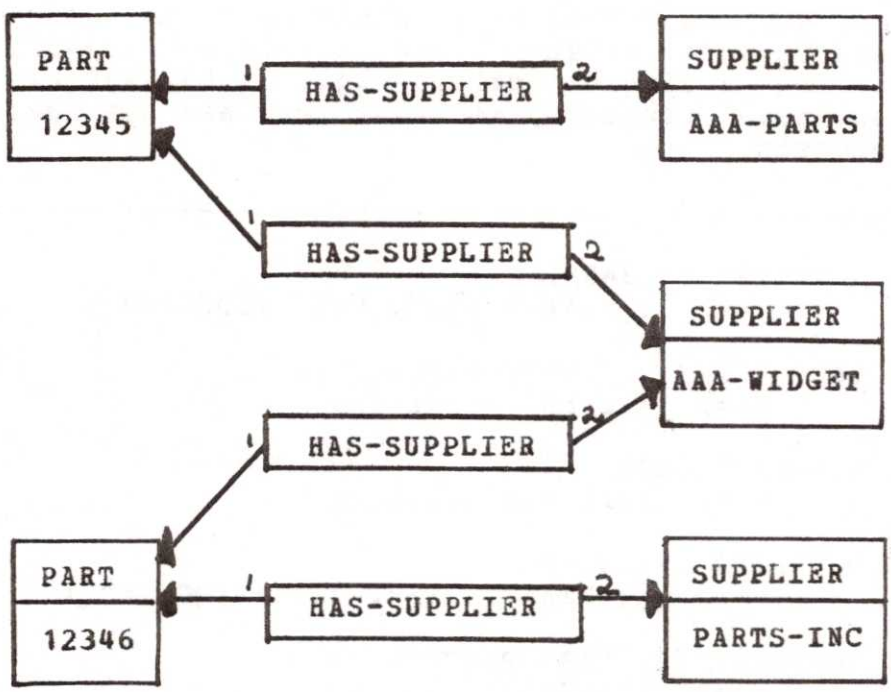


Figure 5: SUPPLIER Data Group

Typeset References. Figure 6, referencing the sample database depicted in Figure 5, illustrates typeset references and their meaning. The value of a typeset reference

```
PART          ====>      (_"12345", "12346"_)

HAS_SUPPLIER  ====>      (_<"12345", "AAA_PARTS">,
                           <"12345", "AAA_WIDGETS">,
                           <"12346", "AAA_WIDGETS">,
                           <"12346", "PARTS_INC">_)
```

Figure 6: Typeset References

is the set of literal names of members of the set. An empty set has the literal value NULL.

In fact every expression in the language evaluates to one or more literals. This interpretation facilitates the use of a primary mechanism for the blending of database and programming syntax, namely, that typesets (which may be restricted to contain only one member), are the variables of the program.

It should be noted that while the language is set-based, the concept of set used is very primitive. No operational distinction is made between a scalar, e.g., "12345", and a set containing a single member, e.g., (_"12345"_). Also, sets do not have structure, in that sets may not have other sets as members.

```
HAS_SUPPLIER("12345")
        ===>  (_"AAA-PARTS", "AAA-WIDGETS"_)

HAS_SUPPLIER((_"12345", "12346"_))
        ===>  all the suppliers

HAS_SUPPLIER(PART)
        ===>   all the suppliers

HAS_SUPPLIER ("12345", ?)
        ===>   (_"AAA-PARTS", "AAA-WIDGETS"_)

HAS_SUPPLIER (?, "AAA-PARTS")
        ===>   "12345"

HAS_SUPPLIER ("12347")
        ===> NULL
```

Figure 7: Function

Function References. In common with SETL [20], DAPLEX [23], and others, the recognition that associations can be referenced as explicit (stored) functions is exploited. In general, given a relationship typeset R whose members are of degree n, then a reference of the form:

- R(E1, ..., Em-1, ? , Em+1, ...,En)

has the meaning "the unique literals denoting objects participating in the mth role of a member of R whose other participants belong to the sets denoted by E1, E2, Em-1, Em+1, ..., En." If the unknown participants are in the nth role, then the structure can be abbreviated:

- R(E1, ..., En-1)

Figure 7 illustrates instances of function references. Note that: a) multi-sets are collapsed, b) the second and third examples evaluate to the union of the expressions HAS_SUPPLIER(12345) and HAS_SUPPLIER(12346), and c) the application of a function to a value for which it is undefined gives the result NULL.

```
HAS_SUPPLIER("12345") INTER HAS_SUPPLIER("12346")
        ===>   "AAA-WIDGETS"

"AAA-PARTS" ISIN HAS_SUPPLIER("12346")
        ===>   FALSE

HAS_SUPPLIER CTNS <"12345", "AAA-PARTS">
        ===>   TRUE

HAS_SUPPLIER:("12345","AAA-PARTS")
        ===>   TRUE

1.1 * COST("12345") > 10.00  ===>  ?
```

Figure 8: Operators

Operators. The operators provided are as follows:

- set operators: UNION, INTER, MINUS
- set comparisons: ISIN, CTNS, EQ, NOTEQ
- value comparisons: = > < etc.
- string operators: || others?

- arithmetic operators: + - etc.
- boolean operators: AND OR NOT

Set operators and comparisons take any expressions as operands. The other operators require that at least one operand be scalar in intension, i.e., detectable at compile time to be scalar by an examination of declarations. Such operators applied to two scalars produce a scalar; applied to a scalar and a set they produce a set. In general, operators require "appropriate" operands, e.g., concatenation may not be applied to numbers. As the traditional operator aspect of the language is not a focal point, no decisions have been made as yet with respect to "appropriate operands", evaluation order, and implicit conversions.

Figure 8 illustrates the use of operators. Note the use of a shorthand form of CTNS, ":". It can be used with either entity sets or relationship sets, and permits a predicate-like notation.

Built-in Functions. While the general topic of built-in functions will not be considered here, functions operating on sets as a whole, such as COUNT, AVG, SUM, will be discussed to illustrate a particular set of problems and the solutions provided.

Consider the problem of obtaining a number representing the average cost of parts, in a single expression, assuming the schema shown in figure 9. The form AVG(COST(PART)) will not do. Based on earlier definitions, the term COST(PART) obtains a set without any duplicates, whereas AVG would seem to require a multi-set. Furthermore, AVG(X), X a set, obtains the set resulting from the application of AVG to each member of X, which is clearly not what is desired.

"Functional modification" will be introduced to deal with the first problem. If F is a function, then M@F represents a modification M of F. A NONUNIQUE modification might be defined as follows: if F(A), before the elimination of duplicates, gives (_a1, a2, ..., an_) (i.e., for some i, j, ai = aj), then NONUNIQUE@F(A) gives <1, a1>, <2,a2>, etc. Using this convention, two forms of aggregate functions might be defined, one operating on scalars, and one operating on pairs, e.g., AVG, and AVG2. (Note: Another useful functional modification might be INVERSE.)

The second problem can be solved by a variety of means. Possibly the simplest is to enclose arguments intended to be processed as a unit in some distinguished parentheses, e.g., brackets. Thus to find the average salary of employees, one would write: AVG2[NONUNIQUE@SAL(EMP)], or, if it can be

assumed that NONUNIQUE is the most common of the functional modifications, AVG2[@SAL(EMP)].

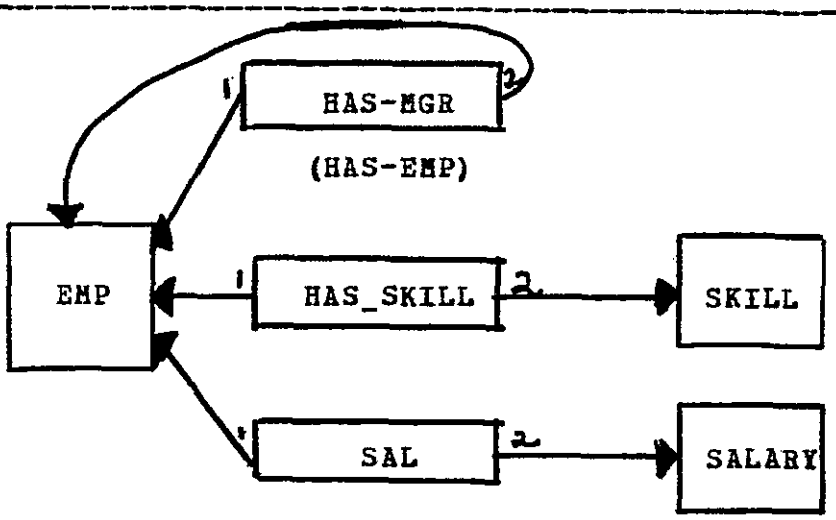


Figure 9: EMPLOYEE Schema

```
?emp WHERE SAL(?emp) > 10.00
     ===>  The employees with salary
           greater than 10.00

?emp WHERE SAL(?emp) > SAL(HAS_MGR(?emp))
     ===>  The employees earning more
           than their managers

<?emp, ?sal> WHERE SAL(HAS_MGR(?emp)) = ?sal
      ===>  Employees paired with their
            managers salaries.

<?emp, SALARY(?emp), ADDR(?emp)> WHERE
            ?emp ISIN EMP
```

Figure 10: Selection

Selection Clauses. Selection clauses function both as ordinary programming language expressions and an extremely powerful and comprehensible linear query language. Consider the first example of figure 10, referencing the EMPLOYEE data group depicted in figure 9. The meaning of the expression is: "What are the values of selection variable '?emp' for which selection clause 'SAL(?emp) > 10.00' has the value TRUE?" Similarly, the third example requests all

pairs of selection variables <?emp, ?sal> which together satisfy the selection clause.

At any particular point in time a selection variable represents at most a single scalar or vector. Each selection variable must have a finite range. The range may be given explicitly, by a term such as "?var ISIN setname", or may be implied by an associated relationship typeset name. For example, ?emp is known to range over the set EMP not because of its name, which is arbitrary (?abc would do as well) but because the SAL relationship is defined only between members of EMP and members of SALARY. (The implied range may also be the intersection or union of one or more sets.)

This is a unique approach made feasible by the requirement that relationships draw their participants in specific positions from specific sets. The most important advantage is that it allows the use of several variables having the same range in a straightforward yet succinct way. This requirement is discussed further in conjunction with the model, below.

```
?mgr WHERE HAS_EMP:(?mgr, ??emp)
   AND HAS_SKILL(??emp) EQ NULL
    ===>   Managers responsible for
           employees with no skills

?emp WHERE HAS_SKILL(?emp) EQ SKILL
    ===>   Employees having all skills

?mgr WHERE HAS_EMPS(?mgr) ISIN
  (?emp WHERE SAL(?emp) > SAL(HAS_MGR(?emp))
    ===>   Managers, all of whose employees
           earn more than they do
```

Figure 11: Quantification Equivalents

Quantification. Existential quantification is indicated by the use of a double question mark, e.g., ??string. This is not necessary for the query form, as such quantification can be inferred for any selection variable not appearing to the left of "WHERE". However, selection variables can function as iteration variables in loops (see below), so in a programmming context the assumed scope of the variable must be made explicit.

The capabilities provided by explicit universal quantification can be handled adequately by set comparison. Figure 11 provides examples of the use of quantification. These examples are particularly good illustrations of the balance of succinctness and clarity achieved by the language.

4.0 ASSIGNMENT STATEMENTS

The language being described is not intended as a "DML", to be embedded in some existing language. Such combinations must always preserve a certain amount of fragmentation. Instead, the flavor of HLL syntax is preserved, but statement forms are modified to suit the combined database / programming environment.

Thus a basic assignment statement with the traditional form "X = y", X a set, and y an element or set, is included. All necessary set manipulations can be performed using the basic assignment statement. However, some additional forms, described below, are introduced.

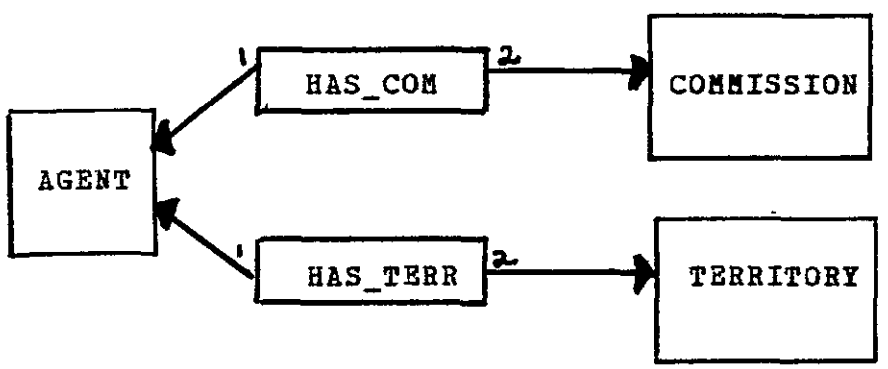


Figure 12: AGENT Data Group

Extended Assignment. Figure 13, referencing the schema shown in figure 12, illustrates the use of basic and extended assignment operators. The form "X += Y" performs insertion, and is equivalent to "X = X UNION Y." The form "X -= Y" performs deletion, and is equivalent to "X = X MINUS Y."

```
AGENT = (_"Smith", "Jones"_);

AGENT += "Reilly";
         ===>   AGENT = AGENT UNION "Reilly";

AGENT -= "Jones";
         ===>   AGENT = AGENT MINUS "Jones";
```

Figure 13: Assignment

This extension reflects the spirit of the language; since sets are things one naturally adds to and deletes from, the change ensures that the readability of HLL's is preserved in the database environment.

Other assignment operators might be introduced to indicate what is assumed by the programmer about the "left of equals" set before execution of the statement, as per figure 14. Such operators would further clarify intent, and foster integrity (i.e., the enclosing transaction would not succeed if the assumptions were untrue).

OP	SYMBOL	"LEFT OF EQUALS" ASSERTION
Equal	=	Don't Care
	==	Empty (Init)
	/=	Non-Empty (Replace)
Add	+=	Don't Care
	++=	Doesn't contain any right-of-eq
Delet	-=	Don't Care
	--=	Must contain all right-of-eq

Figure 14: Other Assignment Possibilities

Functional Assignment. Figure 15 illustrates the use of functional assignment. It is used in SETL [20] and is, obviously, the set or database equivalent of array or structure element assignment. In the examples shown, the meaning is probably self-explanatory. To illustrate the more general case, if R is, for example, a ternary relationship, and X and Y are expressions, then

```
R(X, Y) += Z;
===>   R += <?x, ?y, ?z> WHERE ?x ISIN X
              AND ?y ISIN Y AND ?z ISIN Z;
```

```
HASCOM("Smith") = 1.1 * HASCOM("Smith");
    ===>   Increase Agent Smith's commission
           by 10 percent.

HASTERR("Smith") += "northwest";
    ===>   Add "northwest" to Agent Smith's
           territories.

HASTERR("Smith") -= HASTERR(AGENT MINUS "Smith");
    ===>   Remove any territories from Smith
           also covered by another agent.
```

Figure 15: Functional Assignment

while

```
R(X,Y) = Z;
===>   R -= <?x, ?y, *> WHERE ?x ISIN X
                           AND ?y ISIN Y;
       R(X,Y) += Z;
```

```
AGENT +=  "Alice Epstein"
                (HASTERR = "N.Y.", "N.J."
                 HASCOM = .10 (:: COMTYPE "stdd"),
          "Steven Miller"
                 (HASTERR = etc.
```

Figure 16: Factored Assignment

Factored Assignment. Factored assignment provides a readable form for the addition of an object together with some of its associations. While the syntax is completely general, it has the appearance of a a tailored data entry language. (In fact, the form is used as the basis for declarative syntax, see below.) An example is given in figure 16. The effect of this statement is the same as:

```
AGENT += "Alice Epstein";
HASTERR ("Alice Epstein") = (_"N.Y.", "N.J."_);
HASCOM ("Alice Epstein") = .10;
COMTYPE (<"Alice Epstein, .10>) = "stdd";
AGENT += "Steven Miller";
HASTERR ("Steven Miller") = etc.
```

Note that "COMTYPE" modifies the HASCOM relationship <"Alice Epstein", .10>, rather than the COMMISSION .10. This is indicated by the use of "::".

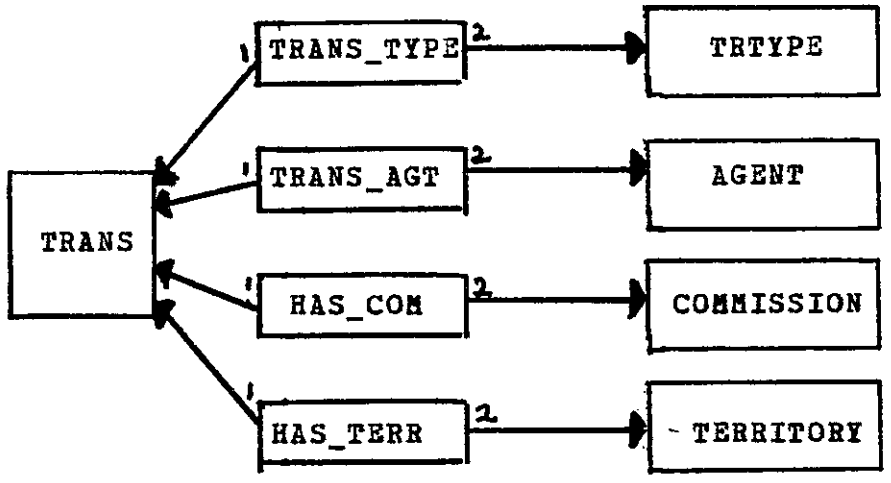


Figure 17: TR (Transaction) Data Group

```
DB.AGENT += ?agent WHERE TR.TRANS_AGT:(?t, ?agent)
            AND TR.TRANS_TYPE:(?t, "NEWAGENT");
```

Figure 18: Multiple Data Groups

Multiple Data Groups. It was stated earlier that atomic network structures were used for all data referenced in a program. Thus they are used for both data base data and local data. This means that a program generally references many disjoint "data groups" having different scopes and lifetimes. To indicate the context of a particular reference, the reference is prefaced by the name of the data group involved (or more precisely, by a symbol which is eventually bound to the name of the data group).

For example, consider an application whose purpose it is to expand the list of agents in the AGENT data group (Figure 12), given the input data group TR (Figure 17). That application might be coded as shown in Figure 18.

5.0 CONTROL STATEMENTS

The language has a group of statements specifying control flow. Some control statements appear as initial statements of "control structures", which have the form:

```
label: control statement;
          .........
          .........
          END label;
```

where labels may be omitted on innermost structures. Statements in this class are DO, which serves as a bracket, REPEAT, which provides iteration, and IF, which provides a conditional execution facility. REPEAT and IF will be discussed further below. Other control statements provided are CONTINUE, EXIT, GOTO and CALL. CONTINUE initiates the next iteration of the enclosing control structure. EXIT leaves the control structure. GOTO is permitted, but not into a control structure from outside its range. CALL will not be discussed further, as issues of argument types and matching have not been decided.

```
REPEAT FOR EACH  <?trans, ?agt>  IN TR.TRANS-AGT;
        DB.AGENT += ?agt
                (HASCOM TR.COM(?trans)
                 HASTERR TR.TERR(?trans));
        END;

REPEAT FOR ?trans WHERE TR.COM(?trans) > .10;

REPEAT FOR EACH ?agent IN TR.AGENT, BY ?agent;
```

Figure 19: Repeat Statements

Repeat Statement. The general form of the repeat statement is:

- REPEAT FOR EACH selection-var|selection-vector IN exp

as illustrated by the first example of figure 19. This form works well as long as "exp" is not a selection expression. If it is, there is unnecessary redundancy, e.g.,

- REPEAT for ?trans IN (?transx WHERE ......

The redundancy can be avoided by the elision illustrated in the second example. The third example shows optional ordering.

```
IF HASCOM(?agent);
          IS < .05 THEN HASCOM(?agent) = .05;
          IS < .10 THEN HASCOM(?agent) = .10;
          IS OTHER THEN HASCOM(?agent) = .15;
          END;

IF COUNT(HASTERR(?agent)), HASCOM(?agent);
          IS < 5, < .05  THEN HASCOM(?agent) = .05;
          IS > 5, < .10  THEN HASCOM(?agent) = .10;
          END;

IF HASCOM(?agent) IS > .05 THEN GOTO NO-CHANGE;
```

Figure 20: Conditional Statements

Conditional Statement. The conditional execution statement, "IF", represents a merging of the traditional "IF" and "CASE", with a touch of decision table. Examples are shown in Figure 20. The result is a very regular form more easily read than traditional nested forms. Consider the first example. Given only IF-THEN-ELSE, then something like the following would be required (no attention is paid here to terminators):

```
IF HASCOM(?agent) < .05
   THEN HASCOM(?agent) = .05
   ELSE IF HASCOM(?agent) < .10
           THEN HASCOM(?agent) = .10
           ELSE HASCOM(?agent) = .15
```

Similarly, the second example, which illustrates the simultaneous testing of multiple variable conditions, might be coded more traditionally as:

```
IF COUNT(HASTERR(?agent)) < 5
   AND HASCOM(?agent) < .05
      THEN HASCOM(?agent) = .05
      ELSE IF COUNT(HASTERR(?agent)) < 10
              AND HASCOM(?agent) < .10
                 THEN HASCOM(?agent) = .10
```

The traditional form is included for simple tests.

This completes the language description portion of the paper. There Statement types not addressed here are either not yet definitionally stable, or fall into the excluded "systems-oriented" category.

6.0 DATA MODEL: DESCRIPTION AND DISCUSSION

The language can be applied to a variety of models. For example, it can be applied to the rough model given originally by the device of "immediately translating" the value of every term from the data base object ostensibly referred to, to its name. Thus, referring to figures 5 and 6, the value of the term PART is defined as the set of names of objects in the set PART.

Probably the best model to choose, however, is one directly implied by the language. For one thing, it makes reading a program that much simpler. For another, the semantics of the above language are rather complex; anything in the direction of simplification is useful. This suggests that a set model be used (which can be represented as a network if desired, see below).

The specific model chosen is related to that used in SETL [20], and is heavily influenced by conceptual model concepts found in [7, 16, 13, and 24]. Some traditional conceptual model constructs are transformed for syntactic reasons. In general, the transformations are defended by taking the position that conceptual model constructs have two purposes:

- To document what information is present (or needed) in a data base.
- To dictate constraints on implementations (object behavior).

Thus given a syntactically inconvenient construct, it is reasonable to explore what functions it seems to fulfill (not always an easy task), and to determine if they might be provided by other means.

The model is introduced below. Descriptions of particular features are interspersed with discussions of why certain features are incorporated or omitted.

The Basic Model. The specific model chosen consists of four kinds of objects: scalars. relationships (also called vectors), sets of scalars ("scalar sets"), and sets of relationships ("relationships sets" or "vector sets").

At least three built-in sets are provided: NUMBERS, STRINGS, and BOOLEANS. Each user-defined scalar set must be constrained to be a subset either of one of the built-in sets, or of some other user-defined set, in a non-circular fashion. (The members of) each vector set must be constrained a) to be of a particular degree, and b) to draw participants in particular positions from particular (single) source sets.

There are two kinds of primitive actions: add an object (scalar or vector) to a set, and delete an object from a set. The deletion of an object from a set causes the deletion of any relationships in which it participates as a member of that set.

Static and Dynamic Sets. Dynamic sets are "normal" sets. Their content is established and modified by assignment statements, and must be finite. The content of static sets, built-in or user-defined, is determined by definition, can be infinite, and is established at the time a data base is created. It may not be modified by assignment, but only by modifications to definitions (as permitted). One might specify that the content of a static set be defined by an expression, referencing only other static sets, in a non-circular manner. However, such an expression must be permitted to represent an infinite set, and special definitional forms would be useful (e.g., to limit string syntax).

Static sets have several uses, the most important of which is the avoidance of unnecessary code. Normally, before an object can be linked into a given relationship, it must be separately placed into a set over which the relationship is defined. For some kinds of objects, however, such as measures, the fact that they are in a given set (e.g., WEIGHT) is relatively uninteresting, so that it makes sense to define the entire set ahead of time. Thus one can write

- DB.PART += "12345" (HAS_WGT 500);

without worrying about explicitly adding 500 to the set WEIGHT.

Since the model is intended to be used for all data, another use of static sets is in modelling traditional array structures, when necessary. In general, an array is represented as a relationship between a static set and the set from which the array values are drawn. For example, to model an n by m array of arbitrary integers, one might define:

a. a static set M of the first m integers

b. a static set N of the first n integers

c. a static relationship NM of their cross product

d. A relationship ARRAY mapping from NM to the set of integers.

A final use of static sets is in constraining the membership of other sets. This is a natural way of providing the required integrity function without the introduction of additional concepts.

Derived Sets. Algorithmic function definitions are to be included in the language, although they are not described here. As recognized in [23], the power of the language makes it worthwhile to also provide a facility for the definition of derived sets by single expressions. Two cases may be identified: a) when the derived sets are explicitly identified as subsets of other sets, and b) when they are not. In the first case, the derived sets may be used as relationship participant constraints.

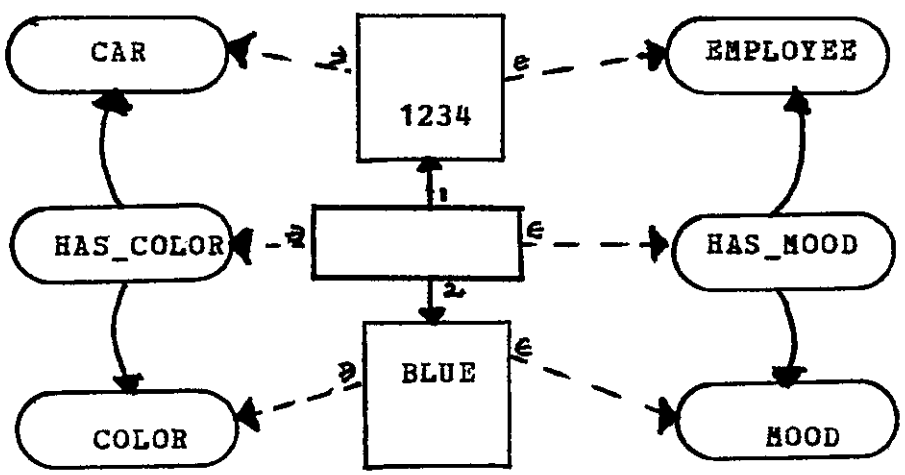


Figure 21: Multi-Context Objects

Some Implications. The model described above has some discomforting aspects, particularly from a data base point of view. First, it implies that the set of objects in a data base is infinite. Second, it allows a data base such as that shown in Figure 21, illustrating that, in general, it is only from the name of a relationship that one can discover the context in which the elements are being related. Thus "BLUE" is being related as a COLOR to the CAR "1234" via the relationship HAS_COLOR, and is being related as a MOOD to the EMPLOYEE "1234" via the relationship

"HAS_MOOD". Finally, the model probably appears overly mechanistic; it has relatively few semantic overtones, and is missing some familiar "conceptual model" constructs.

With respect to the "infinite data base" problem, it should be pointed out that a) the infinite sets are not stored, and b) the uses of infinite sets in expressions are strictly limited.

The second problem, regarding contexts, is visual only, as one cannot generally obtain information about the set of all relationships in which a particular object participates. That being the case, the problem can be resolved by the following diagramming rules, which restore the pictorial simplicity of the original model:

1. Depict only members of dynamic sets, and those members of static sets which participate in some vector (relationship).

2. Display multiple representatives of the same object, one for every set membership by which it qualifies for display (by rule 1). (Note: in the context of additions to the model below, this should read "one for every base-set membership ... ")

3. Connect nodes representing relationships to the appropriate representatives of their participant objects.

(These rules would cause the objects shown in Figure 21 to appear as expected according to the original model.)

The final cause of intuitive discomfort, the lack of semantic constructs, requires more discussion. In the sections immediately following, the reasons for omitting some familiar constructs are explored, and some additions are made to the model to compensate.

What Happened To Entities. Many models explicitly distinguish objects representing "entities" from other scalars (e.g., "attribute values", character strings, etc.). Generally, the names of entities, if they have any, cannot be printed; entities are located via other (non-entity) scalars to which they are related. Often at least one entity must participate in any relationship. While there is no consensus on what entities are [16], they seem to have several purposes in a data model. First, "philosophically", they mirror the consideration that real world objects are different from their real world identifiers. Thus for the sake of fidelity they are represented by blank objects, or special identification systems, etc. This may be justified, however, it can be pointed out that names are in fact the medium by which things are represented in data processing systems,

automated or not.

There are, however, more practical applications of the concept. First, entities are useful when existing identification systems fail. For example, a) there may be several alternative systems in use for a type of real world object, no one of which is universal, or b) the same object must be seen as a member of several unrelated typesets, each with its own naming system, such as employee and stockholder [17]. In such cases, the imposition of a new identification system, local to a data base, but universal within that data base, has some distinct advantages. Second, the distinction between entities and other types of scalars can serve the same purpose as the static / dynamic set distinction provided here.

The use of entities, however, is linguistically awkward. It has the effect of constantly forcing oblique references to objects, e.g.,

```
•   HAS_TERR (?agent WHERE AGT_NAME (?agent) = "Smith")
```

instead of

```
•   HAS_TERR ("Smith")
```

Things become even more awkward in the context of multiple data groups, because an object would be represented as a different entity in each group (which also seems to remove some of the philosophical justification). One could not for example, state:

```
•   DB.AGENT += TR.AGENT;
```

but rather would require something like:

```
•   FOR EACH ?agent IN TR.AGENT;
        DB.AGENT += WK.TEMP = NEW_ENT(DB); (a built-in fn)
        DB.AGT_NAME (WK.TEMP) = TR.AGT_NAME (?agent);
```

Given a) the linguistic inconvenience, and b) the fact that the model is intended to be used for all data (not just "data base data"), it is clear that though the concept of entity can be useful, it cannot constitute a basic element of the model. To provide the functions of entities, when needed, two additional constructs are provided: "surrogates", and "propagation subsets".

The concept of surrogate is adapted from [12]. An additional built-in set, SURROGATE, is provided, being an infinite set of distinguished objects having the form •real-number, together with a built-in function "NEWSURR(databasename)". Then if entities are desired in a

data base, the user may define a set called ENTITY (or anything else), constrained to draw its members from SURROGATE. For consistency with the language, surrogates (the members of SURROGATE) are printable.

Propagation subsets are related to similar concepts in [7, 13, 24]. The sets which have been considered so far will be referred to as "base sets". If a set S1 is declared to be not a base set, but a "propagation subset" of some other set S, it has the effect that the addition of an object to S1 implicitly causes its addition to S, if not already there. The constraining set of S1 is the same as the constraining set of S (unless S is itself a propagation subset, etc.). However, relationship participants can be constrained to propagation subsets.

Propagation subsets are useful, in general, as implementations of the "generalization hierarchy" of [24], but they are not absolutely required except in an entity context. When entities are not used, the equivalent function can be obtained by:

a. defining some classifying relationships (e.g., ACCOUNT_TYPE)

b. defining some derived sets based on the classifying relationships (e.g., CHECKING_ACCTS, SAVINGS_ACCTS)

c. using the derived sets as relationship participation constraints.

With entities, however, a sufficiently large number of objects belong to a single generalization hierarchy, that the lack of explicit subsets is too unwieldy, both definitionally and operationally.

What Happened to Attributes. Many models considered "conceptual" distinguish strongly among various types of associations. Some of the distinctions made have a semantic flavor (e.g., event associations, attribute associations), while others seem structurally based (e.g., only binary relationships allowed, others handled as entities). While semantic distinctions may provide interesting information, they are not included in the model, for two reasons. First, there is no consensus on what categories are useful, and what they represent. (This may be because such determinations are more properly made in the context of "knowledge base" systems.) Second, distinctions in constructs generally imply distinctions in syntax, and hence complexity.

As an example, consider attribute associations. While attributes have a flavor of one-sidedness, and a seeming purpose of describing one of the participants, these are

clearly rather subjective notions [15], especially in our non-entity environment. Moreover, if attributes were included in the model, they would be syntactically annoying; given an additional database construct, presumably there should be an additional language construct, e.g.:

- ?agt.SALARY = 100 AND MANAGER(?agt) = "Jones"

instead of

- SALARY(?agt) = 100 AND MANAGER(?agt) = "Jones"

Given the lack of theoretic justification, the syntactic problem is considered decisive. With respect to representing structurally different relationships by different constructs, it is very difficult to understand why this should be done. The subject will not be pursued here, except to note that the motivation should not be one of reference syntax; the language forms described above allow both n-ary relationships, and relationships participating in other relationships, to be referenced without difficulty.

Why Not Use "Natural" Relationship Names. The model requires that all members of a relationship set draw participants in the same role from the same set. In other, more "natural" models, the same association name can be used to relate many kinds of objects. For example, CONTAINS might relate both OFFICES to FURNITURE and DEPARTMENTS TO EMPLOYEES. This is indeed the way associations are referenced in the "real world", where contexts can be implicit. In an accessible data model, however, contextual information is needed both in the data, and in references to it.

With respect to the data, consider first relationships between non-surrogates. Here only some sort of relationship name or name adjunct can indicate which objects are being related, e.g., which CONTAINS relationship pertains to OFFICE "15C" and which to DEPARTMENT "15C". Next, even in the case of surrogates, each representing a single object, there is a problem. Consider a HAS_RESPONSIBILITY relationship, linking an object representing both an employee and an officer of a club to a task. There must be a way of indicating in what role the person has that responsibility.

References to relationships must also include contextual information, to indicate what specific relationship and selection variable domains are intended. Such contextual information can be provided in many ways. The use of distinguished relationship names has advantages both of succinctness, and of consistency with what is clearly necessary in the data itself. While it might be argued that unique relationship names are difficult to remember, this problem can be alleviated by establishing conventions for

relationship naming. For example, relationships between X and Y might normally be named "HAS_Y", with "X_HAS_Y" reserved for cases where more specificity is needed.

Summary. In summary, it might be observed that in the process of integrating data base accesses into a classical HLL framework, both the language forms and the data constructs undergo some adjustment.

7.0 DATA DEFINITION

The discussion of data definition will be very brief, and is intended only to provide a certain amount of perspective on the types of assumptions being made.

Definition Data Groups The language is assumed to exist within a development / execution environment incorporating many atomic network data groups, among them application data groups, and definition data groups. Each global data group has an associated definition data group specifying such things as:

- The names of, and membership constraints on, its scalar sets.
- The names of, and participant constraints on, its vector sets.
- Definitions of derived sets.
- Other integrity constraints on data base content, expressed in either specialized form (for frequent constraints), or as general predicates.

Definition information is expressed using a prescribed group of scalar and vector sets appropriate to data definition, and is entered using normal means of accessing data groups.

Definitions in Application Programs. Application programs are also assumed to contain definition information, namely: a) declarations of subsets of global data groups used in the program, and b) declarations of local data groups. Declarations look like factored assignment statements, and are considered to create local definition data groups, loaded at compile time. For example, the following statements might be used to define a local data group named TEMP:

```
DCL TEMP SCALARS = TRANS (CONSTRAIN = INTEGER),
                   AGENT (CONSTRAIN = CHAR);
```

- DCL TEMP RELS = TR_AGT (PART = TRANS (::INROLE = 1),
  AGENT (::INROLE = 2));

Derived sets and functions can also be defined, within a program, to span data groups, for example:

- DEFINE BAD_SUPPLIER = ?supp WHERE DB.DELIVERY_WEEKS ...

- DEFINE GRANDCHILD(?x) = DB.CHILD (DB1.CHILD(?x))

8.0 DIRECTIONS FOR FURTHER WORK

As should be obvious, the above represents only a small step toward the goal of language integration. Some obvious next steps in the specification area include:

- Completion of the language specification to include systems-oriented aspects. What is needed is an appropriate model of the structure and dynamics of an application system, in the context of which decisions, such as how to handle process initiation, messages, transaction boundaries, etc., can be made.

- A considerably more precise expression of language semantics, and the imposition of restrictions on expressions, both to ensure "computability".

Some work has been completed in these areas and should be available shortly. In addition, a considerable amount of work is needed in understanding how to apply current techniques in optimization of HLLs, data retrieval, and data storage design, to the language.

9.0 ACKNOWLEDGMENTS

The development of the model and language took place in two stages. In 1976, the author worked closely with Bill Kent; many aspects of the model were developed jointly at that time. In the second stage, 1978-79, discussions with Dennis McLeod, Farhad Arbab, and Guillermo Rodriguez were extremely helpful.

10.0 REFERENCES

1. P. Buneman, R.E. Frankel, "FQL, A Functional Query Language" ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass. (May 1977) 52-58

2. M. Berthaud, M. Duponchel, "Toward A Formal Language for Functional Specifications", Proc. IFIP Working Conf. on Constructing Quality Software, North Holland (1978)

3. H. Biller, E.J. Neuhold, "Concepts for the Conceptual Schema", In Architecture and Models in Data Base Management Systems, G.M. Nijssen, Ed., North Holland (1977) 1-30

4. M.W. Blasgen et. al., "SYSTEM R: An Architectural Update", IBM Research Report RJ2581, San Jose, California (July 1979)

5. R.J. Brachman, "On the Epistemological Status of Semantic Networks", In Associative Networks: Representation and Use of Knowledge by Computers, Academic Press (1979) 3-50

6. P. Brinch Hansen, The Architecture of Concurrent Programs, Prentice Hall (July 1977)

7. E.F. Codd "Extending the Database Relational Model to Capture More Meaning", ACM Transactions on Database Systems 4,4 (Dec. 1979) 397-434

8. C.J. Date, "An Architecture for High-Level Language Database Extensions", Proc. ACM SIGMOD Int. Conf. on Management of Data, (June 1976)

9. J.A. Feldman, "High Level Programming for Distributed Computing", Communications of the ACM, 22,6 (June 1979), 353-368

10. N. Goldman, D. Wile, "A Database Foundation For Process Specifications", Proc. Int. Conf. on Entity-Relationship Approach to Systems Analysis and Design Los Angeles (Dec 1979) 426-445

11. R.L.Griffith, "Information Structures" IBM Tech. Rep. TR03.014. IBM, San Jose, Calif. (May 1976)

12. P. Hall, J. Owlett, S. Todd, "Relations and Entities" In Modelling in Data Base Management Systems, G.M. Nijssen, Ed., North Holland (1976)

13. M.M. Hammer, D.J. McLeod, "The Semantic Data Model: A Modelling Mechanism for Database Applications" Proc. ACM SIGMOD Int. Conf. on Managment of Data (May 1978)

14. B. Housel, V. Waddle, S.B. Yao, "Functional Dependency Model for Logical Database Design" Proc. 5th Int. Conf. on Very Large Data Bases. Rio de Janeiro, Brazil (Oct 1979) 194-208

15. W. Kent, "Entities and Relationships in Information", Architecture and Models in Data Base Management Systems, G.M.Nijssen, Ed., North Holland (1977) 67-92

16. W. Kent, Data and Reality, North Holland (1978)

17. W. Kent, "Limitations of Record-Based Information Models", ACM Transactions on Database Systems, 4,1, (March 1979) 107-131

18. H.M. Markowitz, B. Hausner, H.W.Karr, SIMSCRIPT: A Simulation Programming Language, Prentice Hall (April 1963)

19. N.S. Prywes, A. Pnueli, S. Shastry, "Use of a Nonprocedural Specification Language and Associated Program Generator In Software Development", ACM Transactions on Programming Languages and Systems 1,2 (October 1979) 196-217

20. J.T.Schwartz, On Programming, An Interim Report on the SETL Project, Computer Science Department, Courant Inst. Math Sci., New York University (1973)

21. M.E. Senko, "DIAM II: The Binary Infological Level and its Data Base Language FORAL", Proc. Conf. on Data Abstraction, Definition, and Structure, Salt Lake City (March 1976)

22. G.C.H. Sharman, "A New Model of Relational Data Base and High Level Languages", Technical Report TR.12.136, IBM United Kingdom, (Feb. 1975)

23. D.W. Shipman, "The Functional Data Model and the Data Language Daplex", ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass. (May 1977) Attendee Supplement

24. J.M. Smith, D.C.P. Smith, "Database Abstractions: Aggregation and Generalization", ACM Transactions on Database Systems, 2,2 (June 1977) 105-133

List of Figures

SCIENTIFIC CENTER REPORT INDEXING INFORMATION

1. AUTHOR(S) :

Paula S. Newman

2. TITLE :

An Atomic Network Programming Language

3. ORIGINATING DEPARTMENT

L. A. Scientific Center - 60G

4. REPORT NUMBER

G320- 2704

5a. NUMBER OF PAGES	5b. NUMBER OF REFERENCES
27	24

9. SUBJECT INDEX TERMS

computer application
programming
Very High Level Language
semantic network
application development
entity/relationship model

6a. DATE COMPLETED	6b. DATE OF INITIAL PRINTING	6c. DATE OF LAST PRINTING
March 1980	June 1980	

7. ABSTRACT :

There have been many recent studies of approaches to reducing the fragmentation of implementation languages into programming languages, data manipulation languages, command languages, etc. The purpose of this paper is to present some current results of one such study. The results include the definition of a significant part of a language which completely integrates data base accessing into a traditional programming language framework, and the definition and justification of a data model which makes such integration feasible. The model used is an instance of what is called here an "atomic network", a network in which each fact is represented by an individual element.

8. REMARKS :

1977 IBM LOS ANGELES SCIENTIFIC CENTER OUTSIDE PUBLICATIONS

T. LANG & E. B. FERNANDEZ, Improving the Computation of Lower Bounds for Optimal Schedules, IBM Journal of Research & Development, Vol. 21, No. 3, May 1977, 273-280.

A. INSELBERG, (G320-2684) Variable Geometry Cochlear Model at Low Input Frequencies: A Basis for Compensating Morphological Disorders, IBM Journal of Research & Development, Vol. 21, No. 5, September 1977, 461-478.

T. LANG, E. NAHOURAII, K. KASUGA, E. B. FERNANDEZ, An Architectural Extension for a Large Database System Incorporating a Processor for Disk Search, Proceedings of the 3rd International Conference on Very Large Data Bases, IEEE Computer Society, or ACM, Tokyo, 1977, 204-210.

T. LANG, E. B. FERNANDEZ, R. C. SUMMERS, A System Architecture for Compile-Time Actions in Databases, ACM 77 Proceedings of the Annual Conference, Seattle, Washington, October 17-19, 1977, 11-15.

E. B. FERNANDEZ & C. WOOD, (G320-2685) The Relationship Between Operating System and Database System Security: A Survey, Proceedings of COMPSAC 77, 1st International Computer Software Applications Conference, IEEE Computer Society, Chicago, Ill., November 8-11, 1977, 453-462.

T. LANG, C. WOOD & E. B. FERNANDEZ (G320-2686), Database Buffer Paging in Virtual Storage Systems, ACM Transactions on Database Systems, Vol. 2, No. 4, December 1977, 339-351.

1978 IBM LOS ANGELES SCIENTIFIC CENTER OUTSIDE PUBLICATIONS

B. DIMSDALE, (G320-2692) Convex Cubic Splines, IBM J. Res. Develop., Vol. 22, No. 2, March 1978, 168-178.

E. B. FERNANDEZ, T. LANG, C. WOOD, Effect of Replacement Algorithms on a Paged Buffer Database System, IBM J. Res. Develop., Vol. 22 No. 2, March 1978, 185-196.

A. INSELBERG, (G320-2669) Cochlear Dynamics: the Evolution of a Mathematical Model, Siam Review, Vol. 20, No. 2, April 1978, 301-351.

S. A. JUROVICS, Optimization Applied to the Design of an Energy Efficient Building, IBM Journal of Research and Development, Vol. 22, No. 4, July 1978, 378-385.

E. B. FERNANDEZ, R. C. SUMMERS, T. LANG, & C. D. COLEMAN, (G320-2683) Architectural Support for System Protection and Database Security, IEEE Transactions on Computers, C-27, No. 8, August 1978, 767-771.

R. C. SUMMERS & E. B. FERNANDEZ, An Approach to Data Security, Proceedings of the 8th Australian Computer Conference, September 1, 1978.

D. W. LOW, A Directed Weather Data Filter, IBM Journal of Research & Development, Vol. 22, No. 5, September 1978, 487-497.

STEPHAN A. JUROVICS & DAVID W. LOW, Optimizing the Passive Solar Characteristics of Buildings, Presented at the Winter Annual Meeting of ASME, San Francisco, California, December 10-15, 1978, 43-51.