# The Data Model of IDE: A Value Network

P. S. Newman

IBM Los Angeles Scientific Center
11601 Wilshire Boulevard
Los Angeles, California 90025

## Abstract

The purpose of this paper is to describe and analyze the data model used in IDE, an experimental integrated data processing environment. The data model is one of a class of models which might be termed "value networks". Models in this class have considerable, as yet unexploited, potential for eliminating discontinuities between programming language forms and data base access forms. The paper describes the IDE value network, discusses how its features contribute to the accessibility of data bases from programming languages, and demonstrates that accessibility is achieved without sacrificing representational expressiveness.

## 1. INTRODUCTION

The purpose of this paper is to describe and analyze the data model used in IDE (Integrated Data-processing Environment), as an example of a class of models having considerable, but largely unexplored, advantages in the areas of accessibility from programs, and expressiveness.

The goal of the IDE project[1] is to demonstrate the feasibility of providing expected types of application development facilities - programming language, data base management, application generation, repositories - in a more unified way than is usually the case.

One of the most important methods used in IDE to unify these facilities is to employ a **single data model** for all persistent data (files and data bases) of the environment, **and** for all local data of PL/IDE, the programming language of the environment.

This approach allows all data, local and persistent, to be accessed using the same, very high level syntax. In using only one data model, it differs from approaches which **add** data base oriented structures and operators to programming languages already containing a full complement of traditional data types. Such additive approaches, examples of which are EAS/E,[2] PLAIN,[3] ADAPLEX,[4] and ADAREL,[5] do obtain consistency between (some) local and external data references, but they also increase the size and complexity of the programming languages by the additions.

The single-model approach is practical, however, only if the data model meets some rather stringent requirements. It must be expressive enough to serve as a data base model, but, since it is used for all data, also simple enough to allow succinct programming language access.

The data model designed around these requirements in IDE falls into a class of closely-related models which might be called **value networks**. Like other network models, such as the entity/relationship model,[6] and the functional model,[7 8] value networks are connected. However, the objects connected are values, rather than a required mixture of values and abstract entities.

The result is a formulation particularly suitable as a base for programming language. Like functional models, value networks are subject to functional reference patterns consistent with HLL syntax. The elimination, in value networks, of obligatory entities allows references to be more direct, and allows modelling of all program-local data. As will be discussed further on, the omission of entities has no ill effect on the expressive power of these models.

Many models have been described which may be considered value networks, including the DIAM,[9] IDEA,[10] and NDB[11] models, and the data model of FST.[12] Some more recent representatives include the ODB[13] (a refinement of NDB) and ODM[14] models.*

However, the fact that many models fall into this class does not eliminate the need for further discussion. First, the above examples vary widely in power and flexibility. The IDE model represents a good synthesis, in that a very small number of mechanisms are needed to define data collections with a high degree of representational flexibility.

---

* The list of value networks given here includes only models intended for use in accessible data bases, not the multitudinous, closely-related "conceptual models" intended for use in data base design. An example of the latter is the ENALIM model.[15]



CH2226-9/85/0000/0246$01.00 © 1985 IEEE

Also, the language potential of value networks has not been recognized or exploited to any extent. Most efforts in this area have not focussed on access language, and/or have taken place in environments which did not admit of formulating new language.

The purpose of this paper is, thus, twofold: (a) to describe the IDE variant of the class, and (b) to remedy the lack of attention to the language potential of the class by illustrating and analyzing its language-oriented advantages. The IDE data model is introduced in section 2.* Sections 3 and 4 provide an overview of the PL/IDE reference syntax, which is heavily influenced by the SETL language.[16] Section 5 then analyzes how the features of the model make possible the language forms illustrated, by contrast with the relational and functional models. Finally, section 6 shows that the linguistic advantages of the model are obtained without sacrificing expressiveness.

## 2. THE IDE DATA MODEL

IDE considers data to be grouped into **data collections**. Each module written in the language declares a distinguished data collection for local data. Other data collections represent external data bases. Data collections are made up of **sets**. A set in turn, is made up of **tuples**. The elements of a tuple are ordered. All tuples in a set are homogeneous in the sense that (a) they have the same number of elements, and (b) the elements in a particular tuple position are all drawn from the same **source set**. A source set must be another set in the same data collection.

Some sets are built-in to every data collection, to serve as ultimate source sets. These include the sets REAL, INTEGER, and STRING - respectively the sets of real numbers, integers, and strings (or, rather, their subsets falling within implementation-defined limits). The built-in sets also include subranges of the ultimate source sets, specified by functions. For example, "IntRng(0,9999)" specifies the set of positive integers below 10000, and "StgRng(3,4)" denotes the set of strings having at least three, and at most four characters.

Figure 1 contains a sample data base schema, adapted from a report on ADAPLEX[4]. The set Dept is drawn from the constant set of 4-character strings. The set Major is an association relating Students to Depts. The set Credits_Given is an association relating the **tuples** of Course to their respective Credits given. Finally, the set Student is drawn from the set Person, i.e., it is a subset of Person.

The source set formulation is a rather powerful one, both structuring the network, and providing for its integrity, in an economical fashion. This is made possible by the interaction of set definitions and primitive operations on sets, discussed below.

| SETS | SOURCE SETS |
|---|---|
| Dept | <StgRng(4,4)> |
| Course | <Dept, IntRng(100,500)> |
| Credits_Given | <Course, IntRng(0,6)> |
| Lecturer | <Course, Instructor> |
| Person | <Ssno> |
| P_Name | <Person, StgRng(1,30)> |
| Student | <Person> |
| Advisor | <Student, Instructor> |
| Major | <Student, Dept> |
| Enrolled | <Student, Course> |
| Instructor | <Person> |
| Rank | <Instructor, Rank> |

Figure 1. A Data Collection Schematic

### 2.1 Set Definition

The definition of a set S of a data collection consists primarily of a tuple <S1,.......Sn>, where the Si are the source sets for S.

The **degree** of a defined set S depends on the number n of its source sets, and their degree. If $n > 1$, then the degree of S is n. If one or more of these $n > 1$ source sets themselves have degree $> 1$, then the tuples of S contain tuples as components. That is, one might represent a member of Si as

<a1, a2,....,<aj1,...,ajm>,...,an>

**However**, if S has only one source set, e.g., Sj, then the degree of S is equal to the degree of Sj, and S is a subset of Sj. For example, if Sj has degree m, then one might represent a member of S as <a1,..., am>, rather than as <<a1,....,am>>.

Set definitions also include an indication of **variability**. A set may either be STATIC (constant) or DYNAMIC (variable). STATIC sets are given their constant values as part of their definition. DYNAMIC sets may be further qualified as CONTROLLED, placing limitations on the circumstances under which they may be extended (see below).

The variability properties interact with the primitive operations on sets - insertion and deletion - to assure that source set constraints are respected, but without impeding convenient access. In general, a tuple T = <t1,...,tn> may be added to a set S with source sets <S1,...,Sn> only if for each ti, either (a) ti is already a member of Si, or (b) Si permits implicit insertion - it is DYNAMIC but not CONTROLLED. In the latter case, the addition of T to S will cause the addition of ti to Si.

(Deletion of a tuple is always permissible. The deletion of a tuple T from a set S causes the

* An early version of the IDE data model is described in reference 17.

deletion of all tuples containing T as a component in a position whose source set is S.)

A few more aspects of set definitions are mentioned for completeness. First, sets are declared as SINGLE (constrained to a single member) or MULTIPLE. Also, binary sets (sets of degree 2) are identified as "1-to-n", "1-to-1", "n-to-1", or "m-to-n".

Given the above formulation of set definitions, **source sets** economically provide all of the following capabilities:

- Value constraints: All tuple-elements are ultimately derived from source sets - built-in or user-defined - representing specific value ranges or enumerations.
- Network structure and integrity: Sets connecting members of other sets form the network. No association can exist unless its components are established in their respective source sets. This provides the integrity which can be achieved in non-connected models only by additional provisions (e.g., the "referential integrity" provisions proposed for the relational model by Date.[18]).
- Specialization: When a source set S1 is identified for a set S2, then members of S2 can participate in all associations defined on both S1 and S2. This achieves the specialization capabilities called for by Smith and Smith.[19] For example, in the schema of Figure 1, all persons may participate in a P_Name association, but only those persons in the Student subset may have Instructors, Majors, etc.

## 2.2 Representation of HLL Data Types

The model is used in PL/IDE to represent equivalents of typical built-in programming language data types. The equivalents of scalar constants are provided by STATIC, SINGLE sets. Variables are DYNAMIC, SINGLE sets. HLL structures (e.g., Pascal records, PL/I structures) are modelled as collections of associations with a particular component in common (e.g., the name, major-field, etc., associations for a particular student). (Further on we describe provisions for operating on non-homogeneous aggregations of sets as a unit.)

To aid in the modelling of arrays, cross-products of integer ranges are provided as built-in sets. Then an n-dimensional array can be modelled as a binary association between such a cross product and the set from which the array element values are drawn.

Lists are not modelled directly, but, rather, using the abstraction features of the PL/IDE language, not described here. While they could be modelled as tuples of variable length (as is done in SETL), this alternative is tentatively discarded because of its implications for compilation.

Turning now from built-in types to type systems, source sets provide capabilities related to those of programming language type systems. Instead of declaring many variables of the same type, one declares many sets with the same source set. The language uses this information to constrain what can be stored in a set. (However, PL/IDE is not a strongly-typed language in the sense of ADA.™ Only ultimate source sets - e.g, integer, real - are used to constrain the applicability of built-in operations, and to disambiguate overloaded operations.)

## 2.3 Other Value Networks

As mentioned above, there are other data base models which can also be considered value networks, such as NDB and its refinement ODB, ODM, and the data model of FST. These models do not use the "source set" formulation of IDE, in the sense that ultimate value constraints are either omitted, or require additional mechanism for their expression. Also, the models generally provide less representational flexibility, and do not exploit their potential as bases for programming language.

The NDB/ODB model is closer to the entity/relationship (E/R) model than IDE, in that the "data element" nodes of the network have both entity identifiers and values. However, the entity identifiers of NDB/ODB are, unlike those of the E/R model, externally referenceable. They play an important role in the (intentionally) relatively low level interface of NDB, designed to be used via call from existing programming languages. With regard to representation, NDB/ODB does not permit associations of degree > 2, associations among associations, or subsets.

The FST data model is a true value network, and has an associated higher level language which, while less powerful than PL/IDE, is also related to SETL. The representational restrictions of FST are similar to those of NDB/ODB.

The ODM model also has some similarities to IDE, in that it is one of the few accessible data models which permit associations among associations. However, ODM is intended to be a very simple model, to be used as a base for the construction of richer ones. As it is thus to be a kind of data base primitive, there is no provision for data collection definition, or typing, and thus no constraints. Also, all sets seem to be of degree 2 - i.e., one can create scalar objects, but not sets of such objects.

## 3. SET REFERENCE

The IDE model is specifically designed to allow a succinct, convenient data reference syntax. In

™ ADA is a registered trade mark of the U. S. Government (Ada Joint Program Office).

particular, it allows the adaptation of SETL-like reference forms to the more data-base-oriented model of IDE. We begin by discussing references to module-local data, and then move to the handling of references to external data collections.*

### 3.1 Setnames

A defined set is referenced by its name. Setnames may appear as operands of scalar and set operators. Only those setnames defined as single-position, single-member sets may appear as operands of scalar (e.g., arithmetic, boolean, and string) operators. When setnames appear in such contexts, they are viewed as denoting their scalar content, which treatment may be interpreted as a form of coercion. Thus if A and B are single-position, single-member sets ultimately deriving from the set of real numbers, the expression "A + B" is acceptable and represents the real sum of the two numbers.

Any setnames may appear as operands of set operators (if they are compatible in ultimate source sets). Thus if C and D are arbitrary, compatible sets, the expression "C **union** D" is acceptable and has the expected value.

### 3.2 Functional Reference Forms

These forms, which look like function invocations, allow references such as those shown in Figure 2 (to the sets shown in Figure 1). The structure and then the meaning of these forms is explained below.

In general, a functional reference has the form

s(a1,....,an)

where s is a set name, and the aj's are generally expressions or ? symbols. For example, the set P_Name, with source sets <Person, String>, might be referenced by

P_Name('444-222-111', ?).

The value of this expression is the set of second components of tuples of P_Name whose first component is '444-222-111'. Since we assume that P_Name is defined as "n-to-1", the result is a single member set.

More generally, the value of a functional reference s(a1,...,an) is, in relational terms, the result of:

- selecting those tuples <ti1,.....,tin> from s where, for all j such that aj contains an expression ej, tij = the value of ej.

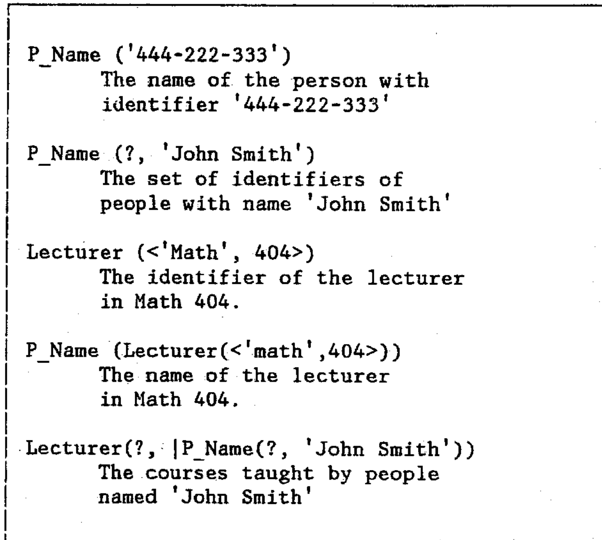

Figure 2. Functional Reference Examples

- projecting the results on those positions j such that aj contains a "?".

The aj's may also include the combination "?:expression", indicating that the position is used for both selection and projection. Also, the aj's may include the symbol "*", which is an expression whose value matches anything.

By convention, a reference s(a1,...,am) to a set of degree m+n is equivalent to a reference s(a1,...,am, ?...?), with n trailing ?. Thus

P_Name('444-222-111')

is equivalent to P_Name('444-222-111',?).

Multi-valued (set) arguments are permitted within the functional reference form. For example, the names of all students enrolled in a course c might be found by

P_Name(|Enrolled(?, c)).

The result of the inner (Enrolled) reference is a set e of all students enrolled in course c. "|" indicates that the P_Name function is to be applied once for each member of e. The result of the full expression is union of the results of each application of P_name.

In general, a functional reference containing one or more arguments preceded by "|" is evaluated by considering the arguments as a sequence of domains. The result is obtained by taking the union of the results of applying the function once to each argument list in the cross product of those domains.

It should be noted that the functional reference form can be used to access IDE tuples because their components are ordered (rather than labelled), and non-directional (i.e., the question marks can be placed anywhere). It is true that the form is con-

* The discussion of PL/IDE in this paper is informal, and limited to data accessing aspects relevant to the discussion to follow. The full language will be described in a forthcoming paper.

venient only for referencing short tuples, as positional referencing of long tuples is very error prone. But the use of short tuples is consistent with the most expressive use of the model, in which independent associations are represented as separate sets. This point will be addressed again in the context of comparisons with the relational model.

A closely related shorthand form for testing set membership is also defined. The form

```
s:<a1,...,an>
```

where the ai's may be expressions or "*" (don't care), tests whether <a1,....,an> is a member of s. It is equivalent to the expression

```
COUNT(s(?:a1, ?:a2, ..., ?:an)) ne 0
```

### 3.3 Query Forms

Another important set manipulation form is the query form. It is adapted from the "set-former" construct of SETL. A motivating example is

```
{<?s> <?d> where Major:<?s,?d>
           and DeptFaculty:<?d, 'Science'>}
```

meaning "the students majoring in departments in the science faculty". The bound variables of the query are ?s and ?d. The separate bracketing of ?d indicates that the values found for ?d are not used in the result.

In general, a query representation of a set is given by

```
{<result list> <drop list> where boolean exp}
```

where the result and drop lists both have the form <t1,...,tn>. The result list specifies the result tuples. Each term ti may be either a bound variable of the query (e.g., ?v), a constrained bound variable (e.g., "?v **from** setname"), or an expression in some of the other bound variables in the result and drop lists. The drop list specifies those existentially quantified, bound variables of the query not to be used in the result.

The form "?v **from** setname" is required only if the "where clause" is not sufficient to establish a domain for the query variable, directly or indirectly. In the motivating example, it is unnecessary. The boolean expression can only be true for values of ?s and ?d appearing in tuples of the Major set.

The query form is very powerful. While universally quantified variables are not used explicitly, the ability to use set operations and functional references within "where clauses" provides similar capability. For example, the set of instructors each of whom teach all math majors can be found by:

```
{?i from Instructor where
    Major(?,'Math') in Enrolled(?,|Lecturer(?,?i))}
```

(i.e., all instructors ?i for whom the set of math majors Major(?,'Math') is a subset of (**in**) the set of students enrolled in courses given by ?i)

The query form is also the core of the iteration construct. For example, the statement

```
For ?s where Major:<?s,'Math'>
       and Level:<?s,'Senior'>
     ........
     End
```

would iterate over all senior math majors. The result variables of the embedded query serve as the iteration variables.

### 3.4 Assignment

Assignment to objects of the model is done in programming language rather than data manipulation language style. Instead of forms such as "UPDATE xxx WHERE", assignment statement forms are used. However, as befits a data base oriented language, there is not a single assignment operator, but rather three, corresponding to "insert", "delete", and "replace".

"A += B" adds to A the members of B not already in A. "A -= B" deletes from A those members also in B. "A = B" is defined as the result of (a) deleting from A all elements not also in B, and then (b) adding to A all elements of B not in A.*

(Shorthand forms encapsulating arithmetic operations and assignment were introduced in ALGOL68.[20] Schmidt[21] adapted them to relational assignment.)

Assignments can also use functional reference forms "left of equals", e.g.,

```
Major('333-222-111') = 'Math'.
```

The effect of this statement is to modify the set Major so that the value of the expression "Major('333-222-111') becomes 'Math'. More precisely, the statement is equivalent to the sequence

```
Major -= <?x, ?y> where Major:<?x, ?y> and
          ?x eq '333-222-111' and ?y neq 'Math'

Major += <'333-222-111', 'Math'>
```

### 3.5 Data Base Reference

In this paper we will assume that PL/IDE gives special handling to the referencing of external data bases. The actual formulation is more general and object-oriented. However, the syntax and results of the data base references as described here are correct, as far as they go.

* Note that "A=B" is not equivalent to the result of (a) setting A to null and then (b) adding the members of B, because step (a) would cause the erroneous deletion of tuples of other sets having A as a source set.

An external data base reference, according to these assumptions, is obtained by prefixing the name of the data base to the set of interest. For example, if "Chged_Majors" is a local set relating students to their new majors, the statement

```
For <?x,?y> where Chged_Majors:<?x,?y>
Acad.Major(?x) = ?y
End
```

is acceptable and modifies the data base "Acad" based on the content of "Chged_Majors".

Thus local and global data referencing is fully unified. The only difference between the referencing of local and of persistent data is the prefixing of a data base name in the latter case.

## 4. DATA GROUPS

One other aspect of the IDE data model, **data groups**, is addressed here, to complete the presentation of the reference forms developed for the model. Data groups are non-homogeneous aggregations of named sets. They are useful in applications requiring the handling of subcollections of data as a unit, both small subcollections (such as all associations involving a particular item), and large subcollections involved in bulk load, update, and extraction. We discuss below operations which produce and store data group expressions, and then comment further on their uses.

### 4.1 Data Group Expressions and Assignment

Data groups are obtained as the values of **data group expressions**. (Expressions discussed up to this point have been **set expressions**; expressions having individual sets as values.)

There are several kinds of data group expressions. The first is the **enumerated data group expression**, which lists the sets in the data group and their members. An example is:

```
[Student 'George', 'Mary'
 Status  <'George', 'soph'>
 Enrolled <'George', <'math', 131>>,
          <'George', <'compsci', 411>>,
          <'Mary',    <'math', 431>>,
          <'Mary,  <'compsci', 100>>]
```

which produces a data group containing the sets Student, Status, and Enrolled, with the indicated content. Note that the sets contained in a data group are not connected.

The above is an unfactored instance of a form which can be highly factored to limit coding and increase expressiveness. For example, the following enumerated data group expression has exactly the same meaning:

```
[Student
   'George'
     [Status 'soph'
      Enrolled <'math', 131>, <'compsci', 411>],
   'Mary'
     [Enrolled <'math', 431>, <'compsci', 100>]]
```

(There are other enumerated forms relying on templates rather than repeated relationship names, not shown here.)

A second type of data group expression, the **extraction expression**, extracts a data group from an existing data collection. This can be done by supplying a list of sets to be extracted. For example:

```
DB.[Student, Status, Enrolled]
```

which would obtain a data group containing the sets Student, Status, and Enrolled, having the same members as the identically named sets of DB.

Another form of extraction expression obtains data along a hierarchic path through a data collection. An example is

```
DB.[Student [Status  Enrolled ]](set-exp)
```

where "set-exp" is used as a constraint on the first set mentioned. This would obtain a data group containing (a) a set Student, with members 'DB.Student **intersect** set-exp', and (b) the sets Status and Enrolled containing tuples of DB.Status and DB.Enrolled having members of the new Student set as their first components.

(Note: a slightly longer form allows tracing hierarchic paths through other than last-position components of tuples.)

A final way of obtaining a data group is by defining a named path as part of the definition of a stored data collection. A data group can then be obtained by:

```
DB.pathname(set-exp).
```

where set-exp is, as above, a constraint to be used on the first set in the path.

Data groups are subject to the operations of data group union (**dunion**) and data group difference (**dminus**). They may also be used in **data group assignments**, and as arguments in inter-module communication.

A data group assignment statement modifies the content of the left-hand data collection by the right-hand data group. For example, the form

```
DB.[] += data group expression.
```

has the effect of a sequence of "+=" statements on each of the sets of DB having a set with a matching name in the "right of equals" data group. DB might be a large external data base, or a small temporary data collection.

### 4.2. Uses of Data Groups

The full set of uses of data groups in PL/IDE cannot be discussed in the absence of a discussion of the object-oriented aspects of the language. However, a few comments can be made.

First, data group assignment can be used to load or update a data base in bulk. For example, one might write a program consisting only of a data group assignment statement with a very long, enumerated data group expression "right of equals".

Next, data groups can be used as arguments (matched by parameters representing temporary data collections) to provide the equivalent of HLL record arguments.

Perhaps the most interesting use of data groups is to allow extraction and movement of non-homogeneous data to and from data bases. The importance of this type of capability for engineering design applications has been shown by Haskins and Lorie.[22] They have proposed a way of providing this capability in relational data bases by extending the model to add special path-creating fields. In IDE, the same effect is obtained by referencing predefined or dynamically structured paths through ordinary relationships. For example, if PartPath is a predefined path through data collection DB which includes all information about a part, then the statement

```
MYDB.[] += DB.PartPath ('gadget')
```

would copy all information about the part 'gadget' to data collection MYDB.

A final use of data groups is for the declarations of the PL/IDE language itself. The declarative portion of a module or data definition consists of a declaration block whose content is an enumerated data group expression. The latter contains sets designed to be used in schema definition. Other, related declarative uses of data groups are discussed in reference 23.

## 5. FEATURES AND ACCESS LANGUAGE

As demonstrated in the previous sections, value networks in general, and the IDE data model in particular, are extremely amenable to programming language access. Furthermore, this accessibility is achieved without loss of the representational expressiveness needed in a good data base model. This section analyzes the features of the IDE data model which contribute to its accessibility. The next major section (section 6) addresses the question of expressiveness.

The ability of the IDE data model to serve as a base for programming language rests on its ability to represent classic HLL data structures (variables, arrays, record structures), and to provide succinct access, in programming language style, to all data.

This ability is obtained, to a large extent, via the formulation of associations in the model, which are (a) positional, (b) atomic, (c) non-directional, and (d) associations of values. Positional associations are associations where participant roles are represented positionally rather than by labels. Atomic associations, as discussed by Imielinski and Lipski[24], are the basic facts represented in a data collection, which cannot be further decomposed without loss of information. (The use of atomic associations cannot be enforced, but the model is most effective when restricted to such associations.) Non-directional associations are ones which can be accessed in any direction. Associations of values are contrasted with associations of abstract, non-printing "entities" found in some models.

The importance of these characteristics can best be shown by observing the effects of their absence from closely related models, namely, the functional and relational models. We begin by outlining, for reference, the assumptions made about the latter models, and then discuss the problems they raise for the design of access syntax.

### 5.1 Assumptions

The version of the functional model to be considered here is that seen in DAPLEX[7] and somewhat modified in ADAPLEX.[4] While the two models differ somewhat, they can both be fairly described as incorporating the following constructs:

- Entities
- "Functions" mapping entities to other entities or to values (but not mapping values to values).
- Subsets of entities - the specialization mechanism.

The version of the relational model assumed considers a data collection to be a collection of relations, each of which is composed of some unordered, named columns. The columns are usually constrained to contain values from particular built-in value domains (e.g., integer ranges). Some columns may be identified as keys.

### 5.2 Functional Model Comparison

On the surface, the functional model appears very similar to the IDE data model. But, while the associations of the functional model are positional, they are not associations of values, and they are not accessible in any direction.

These differences have a number of effects on language. First, references to functional data must be circuitous. Consider, for example, a data base containing information about Parts, identified by

part numbers, and their prices. One cannot simply define a relationship "Price (Partno, Dollars)", and then ask "what is the price of part X123" or, in IDE syntax:

```
Price ('X123').
```

One reason is that values (such as part numbers and prices) may not be directly related to other values; only entities may be related to values. Thus one must define two functions, for example 'PartId(Part, Partno)', and 'Price(Part, Dollars)', both associating the entity 'Part' with a value. And both these functions must be referenced to find the price of a part, i.e., the query becomes "what is the price of the part entity whose partno is X123".

The fact that functions of the functional model are uni-directional further complicates the statement of this query. One cannot use an equivalent of

```
Price( PartId(?, 'X123'))
```

because the PartId function relates Parts to Partno, but not Partno to Parts. Instead, one must move to predicate forms, and use an equivalent of:

```
Price (?x where PartId(?x) = 'X123')
```

To avoid this circumlocution, and obtain the simplicity of "Price('X123')" within the context of the functional model, one can resort to mechanisms for the definition of derived functions and of procedures to update them. Such mechanisms can, in fact, be useful, but should not be required for such simple cases.

An equally serious limitation of the functional model is that it cannot model traditional language structures. Since values only appear in the range of functions, there is no way to represent individual values (constants and variables), or structures which directly relate values to values (such as arrays). Thus any use of the functional model within programming languages must be additive.

Finally, the entity orientation of the functional model makes the movement of data among data collections difficult, as entity identifiers are unique only within data collections. Provisions are made in ADAPLEX for addition of an entity to a data base together with its properties (all functions having that entity as their domain). However, if the range of a function is another entity, reference to the latter must be made via a predicate. This is awkward. Furthermore, it is not obviously extensible to bulk data movement, e.g., the copying of a group of entities, and their properties, from one data collection to another (as can be done using the data group operations described in the previous section).

### 5.3 Relational Model Comparison

We assume that some equivalent of functional reference forms are needed for smooth access to data bases from programming languages. In the introduction to this section four requirements for the formulation of associations were cited as key in supporting direct functional reference: that they be positional, atomic, non-directional, and value-oriented. The need for non-directional, value-oriented associations was demonstrated by a comparison with the functional model. The importance of positional, atomic associations is demonstrated by a consideration of the relational model.

By definition, the relational model uses labelled rather than numbered (positional) columns. By convention, relational schema contain wide relations representing the join of many atomic associations involving the same "key". These two aspects work together. Wide relations are difficult to reference positionally, so column labels must play a part in relational reference patterns.

It is very difficult to design satisfactory functional references in this context. One alternative is to consider the non-key columns of a relation to represent (individual) functions of the key, requiring the imposition of order on the components of compound keys. It leaves the problem, however, of access to the keys themselves.

To allow all components of a relational tuple to be accessed functionally, one can consider the columns to represent functions of "tuple id"s associated with each row. In effect, this provides the beginnings of a functional model (with entities approximated by tuple-ids), having the drawbacks of that model, but fewer of the advantages.

Schmidt[21] has described an approach which synthesizes some of the above directions. In that approach, order is imposed on attributes, so that tuples can be expressed as vectors. Column names are used as functions of tuple variables (expressed as tuple_var.colname) within queries and iteration constructs. Finally, the form "Rel(Sel:s1,..sn)," where Sel is defined as a collection of ordered columns of Rel, is used to identify subrelations having the values s1,..,sn in the specified columns. These variations in reference forms, together with many types of primitive and higher-level operators, produce an interesting data sublanguage, but one which is very complex.

## 6. FEATURES AND EXPRESSIVENESS

The features of the IDE model which facilitate its use as a programming language base do not detract from its suitability as a data base model. That the model provides good support for integrity via source-sets (which supply both value constraints and referential integrity) should be clear. Furthermore, in the area of representational expressiveness, which is primarily a question of the information content of schemas, the model is superior to the relational model and at least the equal of the functional model. This claim is justified below.

### 6.1 Relational Model Comparison

The expressiveness problems of the relational model have been dealt with at length by Kent[25] and others. In general, a data base schema is expressive if it makes clear: (a) what associations are maintained, (b) what kinds of things participate in those associations, and (c) in what roles. In the classic relational model there is no formal provision for expressing this information. Instead, column labels are used informally to convey various kinds of semantic information. For example, in the relation

Student (Ssno, Name, Major, Advisor, Bldg, Room)

Ssno and Name might identify domains, while Major and Advisor serve as both association names and role names; the domains for those columns (respectively department and instructor) are omitted. The labels Bldg and Room might identify domains which, when taken together, represent an office allocated to the student.

It is possible, of course, to establish installation conventions for the use of a relational data base requiring that each independent association be placed in a different relation. This would allow relation names to serve as association names, and thus allow attribute names to represent participant domains in most cases. But it is of no help when two participants in the same relation are drawn from the same domain. Nor is it of assistance in identifying compound domains.

Augmentations to the relational model to increase its expressiveness have been proposed by Codd[26] and others, but each such proposal adds features which can be expressed more economically by reconsidering the model as a whole. The IDE model can be thought of as the result of such a reconsideration.

### 6.2 Functional Model Comparison

It is claimed above that the IDE model is at least as expressive as the functional model. This claim may seem unwarranted, given the lack of "entities" in IDE.

However, it is not clear that the partitioning of data into "entities" and "values" aids in schema construction and comprehension. As discussed by Kent,[27] it is extremely difficult to construct firm criteria for deciding what things will be considered "entities". Thus while the partitioning has a distinct semantic ring, the semantic base is in fact flimsy, and not particularly conducive to increased understanding.

A major justification for the use of entities is that things may have many externally-assigned names, and those names may not be unique. Thus system-assigned names are required. This justification is valid in the context of "knowledge bases" intended to represent arbitrary real world information, where the objects represented generally do not have systematic associated naming systems. For example, in the conceptual graphs of Sowa[28] all nodes representing specific instances are entities.

However, this justification for entities disappears in the context of a commercial data base. There the things chosen as entities (such as parts, departments, employees) are usually precisely those things that do have unique external identifiers (and if they do not - should). Thus we see the use in ADAPLEX of declarative material distinguishing between functions of an entity which uniquely identify the entity (keys) and those which are other properties of the entity.*

It should be noted that what is being questioned here is the utility of **obligatory entities**, not of system-assigned identifiers. It may indeed be useful to supply identifiers for (the members of) some sets, and, furthermore, to disallow any but comparison operations on those identifiers. But this can be done without incurring entity-associated reference inconvenience for all objects. Rather, one can add the set "identifier" to the collection of built-in sets (integer, real, etc.), together with a function defined on dynamic sets returning an identifier not yet used in that set.

## 7. SUMMARY

The IDE data model, like other value networks, is closely related to both the relational and functional models. However, the particular combination of features used in the model render it especially suitable for its intended use - as a data model for both data bases and for the local data of programming languages.

It is rather more expressive, and has more built-in integrity than the relational model, yet permits more concise, programming language oriented reference patterns. Furthermore, since it is value rather than entity oriented, it does not require the circumlocutions, and entity/property decisions of the functional (and entity/relationship) models.

It should be emphasized that the IDE data model is a data base rather than a knowledge base model. The content and usage patterns of repositories serving traditional application needs are not the same as those of repositories used to support substantial natural language applications. These differences are an important subject for investigation as, in the near future, we will want to understand how to provide repositories susceptible to both types of usage in a convenient fashion.

---

* Another justification occasionally given for the entity orientation is that it makes it easier to accommodate to changes in identifiers. For example, one can correct a "partno" error without replacing all associations involving that part. However, this problem can be handled in a value-oriented model by providing an operation which replaces all appearances of one member of a set by another.

## 8. ACKNOWLEDGMENTS

I thank Alex Hurwitz, Arvid Schmalz, and Rita Summers for their careful reviews and many helpful comments.

## 9. REFERENCES

1. P.S. Newman, "Towards an Integrated Development Environment" IBM Systems Journal, 21, 1 (1982) 81-107
2. H.M. Markowitz, A. Malhotra, D.P. Pazel, "The EAS-E Application Development Systems: Principles and Language Summary", Communications of the ACM, 27,8 (August 1984) 785-799
3. A.I. Wasserman, "The Data Management Facilities of PLAIN", ACM-SIGMOD 1979 Intn'l Conf. On Mgmt. of Data, (May 1979) 60-70
4. "ADAPLEX: Rationale and Reference Manual", CCA-83-03, Computer Corporation of America (1983)
5. E. Horowitz, A. Kemper, "AdaRel: A Relational Extension of ADA" Technical Report TR-83-1309, Computer Science Dept., University of Southern California (1983)
6. P.P.-S. Chen, "The Entity-Relationship Model - Toward a Unified View of Data", ACM Transactions on Database Systems 1,1 (March 1976) 9-36
7. D.W. Shipman, "The Functional Data Model and the Data Language Daplex", ACM Transactions on Database Systems, 6,1 (March 1981) 140-173
8. P. Buneman, R.E. Frankel, "FQL, A Functional Query Language" ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass. (May 1979) 52-58
9. M. 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)
10. R. L. Griffith, "Three Principles of Representation for Semantic Networks", ACM Transactions on Database Systems, 7,3 (September 1982) 417-442
11. 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)
12. M. Berthaud, M. Duponchel, "Toward a Formal Language for Functional Specifications", Proc. IFIP Working Conf. on Constructing Quality Software, North Holland (1978)
13. D.M. Choy, R.J. Bamford, F.C. Tung, "A Database Management System for Office Systems and Advanced Workstations", IBM Research Report RJ4318 (June 1984)
14. P. Lyngbaek, D. McLeod, "Object Management in Distributed Information Systems", ACM Transactions on Office Information Systems, 2,2 (April 1984) 96-122
15. G. M. Nijssen, "Current Issues in Conceptual Schema Concepts", in Architecture and Models in Data Base Management Systems, North Holland (1977) 31-66
16. J.T. Schwartz, "On Programming, An Interim Report on the SETL Project", Computer Science Department, Courant Inst. Math Sci., New York University (1973)
17. P.S. Newman, "An Atomic Network Data Model", IBM Scientific Center Report G320-2704, (June 1980)
18. C.J. Date, "Referential Integrity", Proc. Seventh Intn'l Conf. on Very Large Data Bases, (Sept. 1981) 2-12
19. J.M. Smith, D.C.P. Smith, "Database Abstractions: Aggregation and Generalization", ACM Transactions on Database Systems, 2,2 (June 1977) 105-133
20. F.G. Pagan, A Practical Guide to Algol 68, John Wiley & Sons, 1978, p.20
21. J. W. Schmidt, M. Mall, "Abstraction Mechanisms for Database Programming", SIGPLAN Notices 18, 6 (June 1983)83-93
22. R. L. Haskin, R.A. Lorie, "On Extending the Functions of a Relational Database System", Proc. ACM SIGMOD Conf. on Mgmt of Data, (June 1982) 207-212
23. P.S. Newman, "Module and Application Generation via Declarative Modules", IBM Scientific Center Report G320-2739, (October 1984)
24. T. Imielinski, W. Lipski, "A Systematic Approach to Relational Database Theory", Proc ACM SIGMOD Conference on Mgmt of Data, (June 1982) 8-14
25. W. Kent, Data and Reality, North Holland (1978)
26. E.F. Codd, "Extending the Database Relational Model to Capture More Meaning", ACM Transactions on Database Systems 4,4 (Dec. 1979) 397-434
27. W. Kent, "Limitations of Record-Based Information Models", ACM Transactions on Database Systems, 4,1, (March 1979) 107-131
28. J.F. Sowa, Conceptual Structures: Information Processing in Mind and Machine, Addison-Wesley, Reading, MA (1984)