EFDM: Extended Functional Data Model

K. G. KULKARNI* AND M. P. ATKINSON†

Department of Computer Science, University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ

This paper reports on an implementation of the functional data model and the associated data language. The notable features of the data model underlying the implementation include its object orientation, use of the extended function concept to model both attributes and relationships, hierarchically organised entity types with property inheritance, and the specification of derived functions which are used to infer automatically the data implied from that held explicitly in the database. The implementation provides an interactive user interface that supports a high-level set-based query and update language. Facilities for defining and using named views are provided. Meta data is made part of the database so that it can be examined using the data language. The implementation is done using the persistent algorithmic language PS-Algol.

Received February 1984

1. INTRODUCTION

The **Extended Functional Data Model** (EFDM) system is a prototype implementation of a database management system supporting the functional data model. The system provides an interactive user interface based on the DAPLEX language, proposed earlier by Shipman.[1] EFDM extends the DAPLEX language to provide limited facilities to perform general-purpose computation, to specify integrity constraints, and to define and use named views of databases. Meta data is made part of databases and is automatically updated whenever the database schema is modified. The users can examine the meta data at any time using the query language constructs.

The data model underlying EFDM is entity based. An entity is some form of token identifying a unique object in the database and usually represents a unique object in the real world. All inter-object associations are modelled by relating the corresponding entities and not their external references or *keys*. As a consequence, the referential integrity constraints[2,3] are part of the data model itself and need no explicit enforcement. It also accommodates the well-known data abstraction mechanisms like classification, generalisation and aggregation.[4] All the information associated with entities is modelled as functions mapping entities to entities. Entity attributes and inter-entity relationships are modelled as extensionally defined functions, i.e. as tables of arguments and results. Because such functions correspond to *semantically irreducible relations*,[5] there is no necessity for *normalisation*.[6] This simplifies the database design considerably. Functions can also be specified by means of algorithms to compute the result, given the arguments. Such functions help to incorporate procedural information as part of the schema, thus capturing the large part of the information that is normally spread out in application programs.

Functional approach is attractive from the point of view of database use also. First, because queries can be formulated as function applications, functional data languages resemble the familiar programming languages. Secondly, functional data languages can provide full computational power; there is no necessity for embedding them in conventional programming languages for this purpose.

We present a brief discussion of the data model underlying EFDM in Section 2 and discuss various aspects of the user interface of EFDM in Section 3. Section 4 covers the implementation details.

2. DATA MODEL

A good exposition of the **Functional Data Model** (FDM) is given by Shipman[1] and Kulkarni.[7] Briefly, the information corresponding to a real-world application is modelled as a set of entities and of functions relating the entities. Entities are created by an explicit *create* action signifying the start of interest in the corresponding real-world object for which it serves as the representative. When the interest in a certain object ceases, the corresponding entity is removed from the database by an explicit *delete* action. Scalars like integers and strings are also considered as entities, but they have the additional property of having a pre-defined representation and these can neither be created nor deleted by user actions.

Functions with arguments model properties of their arguments, by defining a result entity which is the value of that property. A function may be single-valued (the result of the function is an entity) or multi-valued (the result of the function is a set of entities). The model allows multi-argument functions, and these provide a convenient means to establish n-ary relationships without introducing artificial entities. A function specified by explicitly providing a table of arguments and results is called a *base function*. A function specified by providing an algorithm to compute its result is called a *derived function*.

The entities within a database are organised by a type system which is used to define the domain and range of functions. These types are arranged in a hierarchy so that (i) an instance of an entity type is an instance of all of its supertypes and (ii) a function which applies to entities of a given type also applies to entities which belong to subtypes of that type. This mechanism provides property inheritance. The extensions of entity types are allowed to overlap, thus making it possible to model facts like 'person X is both an employee and a customer'.

* To whom correspondence should be addressed. Current address: Digital Equipment Corporation, 301 Rockrimmon Boulevard, Colorado Springs, Colorado 80919, U.S.A.

† Current address: Department of Computing Science, University of Glasgow, Glasgow, U.K.

```
declare person() - > > entity
declare student() - > > person
declare staff() - > > person
declare course() - > > entity
declare tutorial() - > > entity
declare cname(person) - > string
declare sname(person) - > string
declare sex(person) - > string
declare course(student) - > > course
declare tutorial(student) - > tutorial
declare grade(student,course) - > string
declare title(course) - > string
declare staff(course) - > staff
declare group(tutorial) - > integer
declare staff(tutorial) - > staff
define course(staff) - > > inverse of staff(course)
define lecturer(student) - > > staff(course(student))
define tutor(student) - > staff(tutorial(student))
```

Figure 2-1. Functional schema for a student database.

```
constraint c1 on sex(person) - > total
constraint c2 on sex(person) - > fixed
constraint c3 on cname(person),sname(person) - > unique
constraint c4 on student,staff - > disjoint
constraint c5 on grade(student,course) - > some c in course(student) has
c = course
```

Figure 3-1. Examples of constraint specifications.

An example to represent a class of students and associated entities is given in Fig. 2-1. Note that **declare** or **define** introduces a new function, which if it has no argument is a new entity type with the result type as its supertype. **entity**, **string**, **integer**, **real**, and **boolean** are pre-defined entity types provided by the system. **entity** is the ultimate supertype of all other entity types. **declare** statements introduce base functions (e.g. *cname(person)* function relating a person to his Christian name), and **define** statements introduce derived functions (e.g., *tutor(student)* function relating a student to his tutor). Single-headed arrow $->$ implies the function is single-valued, and double-headed arrow $->>$ implies it is multi-valued. Overloading function names is allowed, and resolved by the type and number of arguments.

3. USER INTERFACE

EFDM provides an interactive language interface which allows for creation, retrieval, and modification of both structure and contents of databases. EFDM allows shared concurrent use of databases with either multiple readers or one writer at any time. The system provides a primitive transaction mechanism, in that the whole session between opening and closing a database is treated as a transaction. The database opened under *write* mode gets locked during the transaction. The system provides automatic crash recovery for *write* transactions.

ADAPLEX DBMS implementation, currently under development at Computer Corporation of America (CCA),[8] is also based on Shipman's proposals. It supports a composite language which is an embedding of a DAPLEX subset into the language ADA.[9] In choosing ADA as the host language, ADAPLEX hopes to exploit the *modules*, *tasks*, and *generics* etc. of ADA to provide some of the encapsulation needed for supporting a number of concurrent users with different views of the database. However, the need to adhere to the syntax and the semantics of ADA has forced the ADAPLEX designers to compromise on the power of the functional data model. In particular, the language supports no multi-argument (base) functions, does not allow the derived functions (or the procedures of ADA) to be made part of the entity type definitions, and does not permit new functions to be defined over a type or deleted, once the database is implemented.

The formal syntax of EFDM is given in Appendix A using the syntax specification language proposed by Wirth.[10] Detailed description of the various facilities can be found in the user manual.[11] The principal features are briefly discussed in the following sections.

3.1. Data definition

As described earlier, **declare** and **define** statements are used to add new functions to the schema. Constraints are specified using the **constraint** statement (see Fig. 3-1). Each constraint is identified by a unique name. Prior to accepting a constraint specification, a check is made to ensure that the existing data satisfies it. If this check fails, the request is aborted and the user informed of the cause of this action.

Currently one can constrain the functions to be *total* (constraint c1) or non-updateable (constraint c2), a group of functions to act as entity designators (constraint c3), or a group of entity types to be non-overlapping (constraint c4). One can also specify constraints on arguments or results of functions by means of predicates (constraint c5).

Functions and constraints can be removed using **drop** command. Removing a function definition removes all the values associated with that function as well as all the other function definitions which depend on it. However, before carrying out this operation, the system displays a list of functions that will be dropped by the action, and seeks confirmation from the user whether to go ahead or not.

EFDM allows the individual function and constraint declaration or deletion statements to occur at any time so that the structure of a database may be refined when it is already populated.

3.2 Data retrieval

As in DAPLEX, **for each** statements and expressions involving function applications are used to formulate queries. **for each** statement is used to iterate through the members of a set. Such **for**-loop statements can be nested to any depth. Expressions consist of names, literals, and operators. Every expression has a value and a type. An expression which evaluates to a set of entities is called *set expression* while that which evaluates a single entity is called *singleton expression*. The expression type corresponds to the entity type under which these entities are to be interpreted.

Every set expression has to be associated with a reference variable using the **in** operator. An example query formulation is shown below:

(Q 1) Find the Christian and surname of all students.

```
for each s in student print cname(s),sname(s);
```

Here, the variable *s* is successively bound to the instances

of *student* entity type. The **print** statement is used to output the results.

A set expression can involve a Boolean predicate. For example, consider the following query:

(Q 2) Find the Christian and surname of all female students.

```
for each s in student such that sex(s) = 'f'
print cname(s),sname(s);
```

Here only those *student* entities for which the boolean expression following **such that** evaluates to **true** are included in the result of the set expression.

The comparison operators supported by the system consist of: < (less than), < = (less than or equal to), > (greater than), > = (greater than or equal to), = (equal to), and ~ = (not equal to). = and ~ = are defined on all entity types. <, < =, >, and > = are defined on **integer**, **real**, and **string** types only. There is one unary operator **not**, and two binary operators **and** and **or** defined on entities of type **boolean**. They have the usual meaning. In addition to the comparison and Boolean operators, existential and universal quantifiers can be made use of in formulating predicates. For example, consider the following query:

(Q 3) Find the Christian and surname of all the students taking the IS1 course.

```
for each s in student such that
some c in course(s) has title(c) = 'IS1'
print cname(s),sname(s);
```

In this query, the expression following **such that** evaluates to **true** if at least one *c* in *course(s)* meets the *title(c)* = '*IS1*' test. Other such quantifiers are **all, no, at least, at most,** or **exactly**.

In general, each argument for a function application can be either a set expression or a singleton expression. When the argument is a set expression, the result of the function application is obtained by iteratively applying the function to each member of the argument set and taking the union of results. For example, consider the following query:

(Q 4) List all the courses taken by female students.

```
for each c in course(s in student such that sex(s) = 'f')
print title(c);
```

Here, the result of the set expression is calculated by taking the union of sets of *course* entities returned by applying the function *course(student)* to each member of the argument set.

Arithmetic may be performed on entities of type **integer** or **real**. The operators are: + for addition, − for subtraction, * for multiplication, / for real division, **div** for integer division throwing away the remainder, and **rem** for remainder after integer division. Currently, only one operator + + is defined on entities of type **string**. It concatenates the two operand strings to form a new string. The set operators **union**, **intersection**, and **difference** may be used to combine set expressions.

In addition, two special operators, **the** and **as**, are provided by the system. The **the** operator applied to a set expression returns a single entity if the result set has the cardinality of 1, otherwise an error condition is raised. For example, consider the following query:

(Q 5) Find the courses taught by Hamish Dewar.

```
for the s in staff such that
cname(s) = 'Hamish' and sname(s) = 'Dewar'
for each c in course(s)
print title(c);
```

The above formulation expresses the fact that only one *staff* entity in the database is expected to have the Christian name 'Hamish' and surname 'Dewar'.

The **as** operator is useful to specify the type of an entity explicitly during an expression evaluation. For example, consider the following query:

(Q 6) Find the names of those students who are taking a course which they teach.

```
for each s in student such that
some c in course(s) has
some c1 in course(s as staff) has c = c1
print cname(s);
```

3.3 Derived functions

Derived functions are specified by means of expressions. For example, the function *tutor(student)* in Fig. 2-1 is specified by the expression *staff(tutorial(student))*. Note that *student* acts both as the formal parameter and as an indication of the parameter's type. If more than one argument has the same type, explicit variables can be introduced using **in**. For example, consider the following boolean function definition,

```
define greater(i in integer, j in integer) − > i>j;
```

EFDM allows recursion for defining derived functions. For example, consider the following familiar **bill of materials** example:

```
declare part()                          − > >  entity
declare subpart(part)                   − > >  part
declare quant(p1 in part, p2 in part)   − >    integer
declare incremental.cost(part)          − >    integer
```

The derived function to provide the total cost of manufacture for a part is defined using recursion as follows:

```
define total.cost(part) − > incremental.cost(part)
+total (over p in subpart(part) quant(p,part)
*total.cost (p));
```

Additionally, EFDM provides three special operators, **inverse of**, **transitive of**, and **compound of**, for defining derived functions. The following derived function defined using the **inverse of** operator

```
define students(course) − > > inverse of
                course(student)
```

is equivalent to (and a short-form for) the following function definition

```
define students(course) − > > s in student such that
        some c in course(s) has c = course
```

Corresponding to the bill of materials example mentioned above, consider the following derived function

```
define subparts(part) − > > transitive of subpart(part)
```

This function returns the set containing the subparts of a given part, the subparts of subparts, subparts of subparts of subparts, etc.

The **compound of** operator is used to define new entity types only. This operator creates derived entities corresponding to the elements of the cartesian product of its operands. For example, the following derived entity type,

```
define enrolment() − > > compound of
            student,course(student)
```

returns entities of *enrolment* type. The new type being

defined will be a subtype of **entity** and will include one entity for each *student–course* tuple. In addition, the system implicitly defines the two functions

student(enrolment) − > student
course(enrolment) − > course

which return the *student* and *course* entities for each *enrolment* entity.

3.4 Aggregation functions

Aggregation functions provided by the system include **count**, **maximum**, **minimum**, **average**, and **total**.

The **count** function applied to a set expression returns the cardinality of the result set in integer form. For example, consider the following:

(Q 7) Find the number of courses taken by Angela Pearson.

for the s **in** student **such that**
cname(s) = 'Angela' **and** sname(s) = 'Pearson'
print count (c **in** course(s));

The **maximum** and **minimum** functions applied to an integer-valued set expression return the maximum and minimum values, respectively, of the result set. The **average** and **total** functions applied to an integer-valued multiset expression return the average and total, respectively, of the result multiset. (A *multiset* or a *bag* is a set which may contain duplicate elements.) A multiset expression is formed using the **over** operator, which takes a set specification and an expression defined over members of that set. For example, consider the following request:

(Q 8) Find the average tutorial group size.

print average(over t **in** tutorial size(t))

where the *size(tutorial)* function yielding the number of students belonging to a tutorial group is derived as follows:

define students(tutorial) − > > **inverse of** tutorial(student)
define size(tutorial) − > **count** (s **in** students(tutorial))

3.5 Database updating

Update operations in EFDM correspond to creating a new entity, assigning or modifying function values for an existing entity, extending or reducing the set of types for an existing entity, or deleting an existing entity. Updates on base functions result in updating the corresponding stored data structures. As in the relational context,[12] it may not be possible to translate updates on derived functions to updates on stored data structures in all the cases. Along with Shipman,[1] we envisage allowing updates on derived functions only if the designer of the derived function has specified a procedure describing how it should behave for each applicable update. Since this is still the subject of implementation experiments, it is not described here.

The following examples illustrate the various operations:

(U 1) Create a new student entity and assign Christian name as Moyana and surname as Johns.

for a new s **in** student
let cname(s) = 'Moyana'
let sname(s) = 'Johns';

When a new entity of a specified entity type is created, all the supertypes of that entity type get populated simultaneously with that new entity. All functions over the entity for each type are undefined unless explicitly assigned. Whenever a new entity is created, all functions applicable to it and constrained to be **total** must be assigned values and all groups of functions constrained to be **unique** must be assigned values which uniquely identify that entity. Otherwise, the whole entity creation will not be permitted.

(U 2) Assign CS 1 and CS 2 courses to Moyana Johns.

for the s **in** student **such that**
cname(s) = 'Moyana' **and** sname(s) = 'Johns'
let course(s) = {**the** c1 **in** course **such that** title(c1) = 'CS1', **the** c2 **in** course **such that** title(c2) = 'CS2'};

(U 3) Add IS1 course to the current course assignment of Moyana Johns.

for the s **in** student **such that**
cname(s) = 'Moyana' **and** sname(s) = 'Johns'
include course(student) = {**the** c1 **in** course **such that** title(c1) = 'IS1'};

(U 4) Drop CS1 course from the current course assignment of Moyana Johns.

for the s **in** student **such that**
cname(s) = 'Moyana' **and** sname(s) = 'Johns'
exclude course(student) = {**the** c1 **in** course **such that** title(c1) = 'CS1'};

The **the** operator in all the above statements ensures that the entities corresponding to the student named Moyana Johns and the courses named CS1, CS2, and IS1 actually exist in the database. If any of these entities are not present, the update statement will not be executed and the user is informed about the missing entities. This is an example of EFDM's treatment of *referential integrity*. Updates on functions constrained to be **fixed** are not permitted. If there are any constraints on the argument or the result values of the function to be updated, the update is allowed only if there are no violations.

Since any entity can potentially belong to more than one entity type, EFDM supports update actions that extend or reduce the set of entity types for an existing entity. For example, an entity already a *student* may be made a *staff* and subsequently cease to be a *student*. Following examples illustrate these operations:

(U 5) Include Moyana Johns into *staff* type.

include staff = {**the** s1 **in** student **such that** cname(s1) = 'Moyana' **and** sname(s1) = 'Johns'}

This will work only if there is no constraint which specifies the *student* and *staff* entity types are disjoint.

(U 6) Exclude Moyana Johns from *student* type.

exclude student = {**the** s1 **in** student **such that** cname(s1) = 'Moyana' **and** sname(s1) = 'Johns'}

Excluding an entity from the extension of a type results in removing its reference from the extensions of all subtypes of that type, if any, and from all functions defined on those types in which it is participating either as an argument or result. Before carrying out the operation, a list of these implicit updates is displayed and the user is asked to confirm the request.

The above operation removes an entity from the

extension of specified entity type and its subtypes, but the entity itself continues to exist in the database as a part of the extension of other entity types, if any. **delete** operation, illustrated below, removes an entity from the database altogether:

(U 7) Delete Moyana Johns from the database.
delete the s **in** student **such that** cname(s) = 'Moyana' **and** sname(s) = 'Johns';

The semantics of this operation is that the specified entity is removed from the extension of all the entity types it belongs to and from all the functions in which it is referenced. As in the previous case, the system displays the list of entity types and functions from which it will be removed and the user has the option either to abort or to proceed with the request.

3.6 User views

EFDM provides a view mechanism which, while providing a different perspective of the global information, also acts as an authorisation mechanism. Using this mechanism a central database administrator who has access to the entire database can define different overlapping user views.

Views are defined using **view** command and removed using **drop** command. Functions in a view are introduced using **deduce** statements. For example, for the student database of Fig. 2-1 we can define a view called *male-students* within the global view as

view malestudents is
deduce male() − > > **entity using** s **in** student **such that** sex(s) = 'm'
deduce name(male) − > **string using** name(student)
end

All functions introduced by **deduce** are treated as derived functions. Notice that **deduce** is used to define view functions instead of **define**. This is because view function definitions involve change of name space; names before the **using** keyword define the namespace of the view being defined, whereas names after the **using** keyword refer to names in the namespace in which the view is being defined.

To use a view, the user responds to a request for a view name at the start of the session. The user is restricted to the names available in the namespace of that view; he has no access to either the global namespace or the namespaces of other views. The user can query the database through his view but he cannot, at present, perform updates. Each query statement issued from a user view is translated into a corresponding query on the global namespace by recursively applying the view definition mapping. Thus it is not necessary to materialise the entire view for every query execution. The user can also define new views of his view.

As every view definition automatically creates a different namespace, which is completely independent of the global namespace as well as the namespaces of other views, view mechanism acts as authorisation control mechanism also. It is only by explicitly including an item in the view definition that a user can gain access to it. Correspondingly, redefining a view to exclude an item from the view definition withdraws the right to access it.

```
metaitem() − > >  entity
name(metaitem) − >  string
text(metaitem) − >  string
document(metaitem) − >  string
function() − > >  metaitem
nargs(function) − >  integer
arguments(function) − > >  function
result(function) − >  function
type(function) − >  string
status(function) − >  string
constraints() − > >  metaitem
view() − > >  metaitem
outerview(view) − >  view
within(view) − > >  view
```

Figure 3-2. The functions to hold meta data of a schema

3.7 Meta data

The meta data of the schema corresponding to an application is held in a set of EFDM functions shown in Fig. 3-2. These functions are automatically populated and modified when **declare**, **define**, **constraint**, **view** or **drop** statements are processed. Only the document function may be explicitly updated by the user. The contents of these functions can be retrieved with the usual retrieval statements.

3.8 Discussion of EFDM user interface

As discussed above, EFDM user interface follows the DAPLEX language proposal quite closely. Additionally, EFDM provides facilities to extend or reduce the set of types of an entity, to delete an entity from the database, to specify integrity constraints and views, and to remove existing functions, constraints or views. EFDM also provides built-in special entities that provide access to meta data.

A major difference between the user interface of EFDM and the DAPLEX language concerns the issue of providing general-purpose computation facilities in data languages. DAPLEX does not provide a complete programming environment. It lacks the power of conventional programming languages to perform arbitrary computation. Shipman proposes that DAPLEX be embedded in another host language for this purpose.[1] This is also the approach adopted in some relational systems (e.g. SQL+PL1 in System R,[13] QUEL+C in INGRES).[14] In contrast, we have chosen to formulate a self-contained language. The motivation for this is twofold: (i) the embedding approach suffers from many shortcomings;[15–17] and (ii) we believe that the functional orientation of the model is ideal for developing an integrated language that caters for both data manipulation and general-purpose computation. In fact, in their query language, FQL,[18] Buneman *et al.* show the feasibility of this approach.

As a first step we have introduced arithmetic, string, Boolean, set and comparison operators as part of our implementation. We also allow recursion for defining derived functions. We are now working on the introduction of other features that make it a full programming language. The main issue here is whether to retain the predominantly functional style as in FQL or to choose an Algol-like style. FQL is purely a query language; it does not handle updates. Essentially the idea of explicitly controlling the implementation or propaga-

tion of updates is antithetical to pure functional programming. Yet organising the storage of data and hence updates is a dominant computing activity central to databases. We are therefore considering borrowing Algol-like features like variables, subprograms and other control structures to yield a convenient and consistent language with a form closer to familiar programming styles.

4. IMPLEMENTATION OF EFDM

EFDM is implemented using the persistent algorithmic language, PS-Algol.[19, 20] The major difference between PS-Algol and other algorithmic languages lies in the way they handle persistence of data objects. The notion of *persistence* is concerned with the length of time the objects in the language exist or are potentially accessible by some programmer or program operation. One can envisage this property as a continuum from the most transient values constructed within individual instruction executions in the CPU, to data held indefinitely as computer systems come and go. Previously, programming languages have treated this property of data consistently within the context of a program (either as values of local variables or as values on the heap) but have made radically different arrangements for data of longer term persistence (such as that in files or in a database). In contrast, PS-Algol accommodates the complete range of persistence and moreover treats the persistence as an orthogonal property of data, in that any data item may exist for any length of time.

Before we discuss the specific implementation details, we will briefly describe the PS-Algol language in the following section.

4.1 PS-Algol

PS-Algol is an Algol-like language derived from the strongly typed programming language S-Algol.[23] The base types of the language are integer, real, boolean, string, pointer and picture. In addition. PS-Algol treats procedures as first-class objects in the language. The type constructors are vector and structure. Multidimensional arrays may be formed by composing the vector construction operations. Structure classes are ordered cartesians of named fields belonging to one of the base types, or to a vector or procedure type. Pointers are access descriptors which can reference instances of any of the structure classes but which may not reference instances of base types or vectors. That is, a pointer is not bound to a structure class. However, when a pointer is dereferenced using a structure field name, a run-time check occurs to ensure the pointer is pointing at a structure with the appropriate field. All compound objects – strings, structures, vectors and procedures – are stored on the heap. The run-time system incorporates a garbage collector which preserves all objects reachable from identifiers currently in scope.

The concept of reachability for identifying limited data persistence during the run of a program serves as the means of identifying *persistent data*, i.e. the data that outlives a program execution. For instance, the guiding principle of garbage collection is that no object that is reachable from identifiers currently in scope may be reclaimed. PS-Algol extends this principle by introducing a new origin for the transitive closure of references, under explicit user control, which differentiates persistent data and transient data. When a transaction is committed, it is possible to identify a root object from which all persistent data are reachable. Hence, preservation of data is a consequence of arranging that there is a way of using that data.

The run-time system of PS-algol undertakes the responsibility for preserving the persistent data on non-volatile store (disk) and for organising the movement of persistent data from disk to active heap during the program run. On the first dereference of a pointer to a structure containing persistent data, that structure is copied to the heap from the secondary storage, possibly carrying out minor translations. Thereafter it is operated on by the same mechanism as for any other data on the heap. When a transaction is committed, all the data on the heap that are reachable from the persistent objects used during the transaction are transferred back to the disk.

Some language systems, notably Interlisp,[21] allow a user to save a snapshot of his environment and to restore that environment subsequently, thus providing the effect of persistence. PS-Algol differs from such language systems in the following respects.

PS-Algol moves only those persistent objects that are referenced by the program to the active heap. Consequently, databases handled by PS-Algol can be much larger than the maximum heap size allowed. In addition, the programs which make only a few small changes to a large body of stored data do not pay a high I/O penalty, as would be the case if the entire environment is to be loaded and saved.

PS-Algol allows sharing of the persistent data by a number of concurrent users.

PS-Algol provides a secure transaction mechanism.

4.2 Data structures

Traditionally, a large part of the data structure design task concerns the organisation of data on secondary store. As the PS-Algol run-time system handles this aspect for all persistent objects, we were spared this task in implementing EFDM. In order to implement EFDM primitives, it was enough to design efficient data structures using the structuring options provided by PS-algol. In addition to making the task much simpler, this allowed us to experiment with different high-level data structures.

In a simple-minded implementation, the entities can be represented as system-controlled unique integer numbers. All base functions can then be implemented as tables of argument and result numbers. Though this makes it easy to accommodate schema changes, it suffers from an excessive storage overhead. In addition, it results in an overly fragmented database, and since it is frequently the case that values for multiple functions applied to the same entity are needed together, this has an adverse effect on the performance.

In our current implementation, entities are represented by a structure shown in Fig. 4-1. For a given entity, the values of all one-argument non-inherited base functions are stored in a vector whose elements are of type **pntr**. The vector contains one element for each function, regardless of whether the function is single-valued or multi-valued.

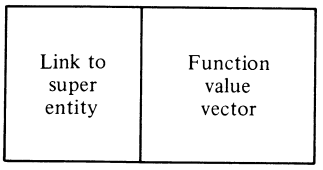

Figure 4-1. Entity data structure.

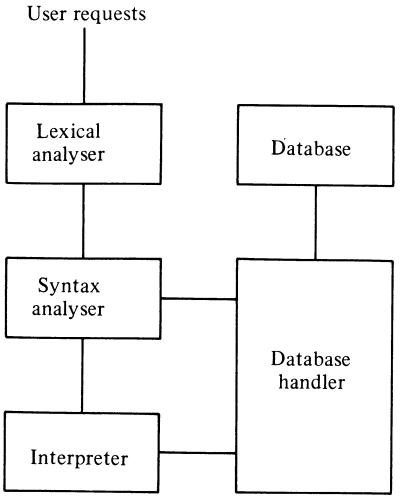

Figure 4-2. Block Diagram of EFDM.

For a single-valued function, the corresponding vector element points to either another entity structure or an instance of a dummy structure designed to yield a pointer to a scalar value. For a multi-valued function, the corresponding vector element points to the head of a linked list. (In the current version, sets are implemented as linked lists.) The elements of such a list may point to either the entity structures or the dummy structures that hold the scalar values. Because of the type hierarchy, functions applicable to a given entity include those defined on all the supertypes of the entity's type. Hence, each entity structure includes a pointer to another entity structure belonging to its immediate supertype.

As the language itself ensures the uniqueness of pointers, a pointer to an entity structure is taken as the unique identifier to the corresponding entity. (It is to be noted that because of the way PS-Algol implements persistence, pointers to the persistent objects are guaranteed to remain unchanged even though the actual data on disk are relocated from one program run to another.) Since entity structures can only accommodate one-argument base functions, a separate representation is required for multi-argument base functions. A multi-argument base function with n arguments, say, is implemented as a linked list of vectors of n+1 elements (of type **pntr**) where the first n elements point to the argument entities and the last element to the result which is either a single entity or a set.

The meta data corresponding to a schema is stored in a *schema table* which contains PS-Algol structures for all function and constraint definitions. (PS-Algol provides a library of routines to manipulate table structures.) The structure for each base function with no arguments (i.e. an entity type) contains a reference to the set of entity structures belonging to it. The structure for a multi-argument base function includes a reference to the corresponding linked list of argument-result value vectors. The structure for a derived function includes a reference to the syntax tree corresponding to its definition. The schema table is made persistent so that everything reachable from schema table, i.e. function definitions and the data associated with functions, persists.

4.3. Implementation details

The block diagram of EFDM implementation is shown in Fig. 4-2. User requests formulated using the EFDM syntax go through the lexical analysis and the syntax analysis phases. The syntax analyser produces a syntax tree for each successfully analysed statement. In addition, it handles all schema modification requests by issuing appropriate calls to the database handler. Other types of requests, i.e. data retrieval and update requests, are passed down to the interpreter. The interpreter traverses the syntax tree formed by the syntax analyser, issuing calls to the database handler whenever interaction with the database is required. The database handler provides storage and retrieval facilities for all the data stored in the system. This includes both user data and system data such as meta data.

The lexical analyser, syntax analyser, interpreter and database handler are all written in PS-Algol. Because of this, we were not required to write code for many of the tasks associated with the traditional DBMS implementations, namely translations between the program's form of data and the form used for the long-term storage medium (disk), organising the transfer of data to and from the disk and the transaction management mechanisms for controlling concurrent database access. (Details of how the PS-Algol run-time system handles these aspects are described in Ref. 22.) The implementation consists of about 3000 lines of PS-Algol code and represents approximately half a man-year's effort.

A user query is executed by systematically traversing the corresponding syntax tree. There are separate procedures for evaluating functions, depending on the number of arguments and the nature of function. For an one-argument base function, the value for a given argument entity is obtained by accessing the appropriate vector entry in the corresponding entity structure. For a multi-argument base function, the value is obtained by searching through the corresponding linked list. For a derived function, the value is obtained by systematically executing the parse tree corresponding to its definition.

All updates on base functions are implemented as changes in the corresponding stored data structures. Creating a new entity results in the creation of appropriate entity structure(s) and adding it(them) to the appropriate linked list(s). Deleting an existing entity results in the removal of corresponding entity structure as well as the removal of all references to it from the extensions of entity types and functions in which it is participating. All constraints concerning the functions participating in a update are checked for violation, and the update is allowed only if there are no violations. Updates on derived functions are currently not supported.

To handle the incremental schema changes, the implementation adopts different techniques depending on the number of arguments the function has and the

nature of the function. The addition or deletion of a one-argument base function results in creating new instances of function value vectors in the corresponding entity structures and copying corresponding values from the old to the new instances. (PS-Algol allows specifying the upper bound of a vector at the time of its creation.) The addition or deletion of other kinds of function has no effect on the stored entity structures.

EFDM has been used extensively for teaching purposes at the University of Edinburgh. A number of student projects have successfully been implemented using it. The emphasis so far has been on getting a reasonably efficient working system that clearly demonstrates, at least at a conceptual level, the power and utility of data model features like entity orientation, entity type hierarchy and the unified treatment of data and programs provided by the functional orientation. The current implementation is expected to act as a test bed for conducting experiments on the issues connected with query optimisation, derived data control, enforcement of integrity constraints, specification and use of views, accommodating schema evolution, etc.

5. CONCLUSIONS

In this paper, we have described a working implementation of the functional data model providing interactive user interface. The flexibility of the extendable entity types and the convenience of property inheritance have been demonstrated to be compatible with a small and understandable set of operations which are feasible to implement. Allowing users access to the meta data has been shown to be both beneficial and safe. The view and data derivation constructs are shown to combine powerfully. This work also demonstrates the fact that by choosing a persistent language like PS-Algol for implementation, it is possible to adhere to the spirit of uniformity that functional data models provide.

Acknowledgements

The authors gratefully acknowledge the efforts of Ken Chisholm, Paul Cockshott and George Ross in providing the persistent programming environment. We appreciate many useful discussions with them and with Pedro Hepp, Segun Owoso and Ron Morrison. The work at Edinburgh was supported by SERC grant 86541. It is now supported by SERC grants 21977 and 21960, and by a grant from ICL. K. G. Kulkarni was supported for the duration of this work by a scholarship from the Association of Commonwealth Universities. The authors also wish to acknowledge the anonymous referee for many useful comments on the earlier drafts of this paper.

REFERENCES

1. D. W. Shipman, The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems* **6** (1) 140–173 (March 1981).
2. E. F. Codd, Extending the relational model of data to capture more meaning. *ACM Transactions on Database Systems*, **4** (4) (December 1979).
3. C. J. Date, Referential integrity. *Proceedings of the 7th International Conference on Very Large Data Bases* (1981).
4. J. M. Smith and D. C. P. Smith, Database abstractions – aggregation and generalisation. *ACM Transactions on Database Systems* **2**, (2) (June 1977).
5. H. Biller, On the notion of irreducible relations. In *Data Base Architecture*, edited G. Brachhi and G. M. Nijssen, pp. 277–296 Amsterdam: North-Holland (1979).
6. E. F. Codd, Further normalisation of the data base relational model. In *Data Base Systems*, Courant Computer Science Symposia Series, vol. 6, Hemel Hempstead: Prentice-Hall (1972).
7. K. G. Kulkarni, Evaluation of functional data models for database design and use. *Ph.D. Dissertation*, University of Edinburgh (1983).
8. J. M. Smith, S. Fox and T. Landers, *Reference Manual for ADAPLEX*. Computer Corporation of America (January 1981).
9. Ichbiah *et al.* Rationale of the design of the programming language Ada. *ACM Sigplan Notices* **14** (6) (1979).
10. N. Wirth and C. A. R. Hoare, A contribution to the development of Algol. *Communications of ACM* **9** (6), 413–431 (1966).
11. K. G. Kulkarni, *Extended Functional Data Model – User Manual*, technical report PPR-7-83, University of Edinburgh (September 1983).
12. U. Dayal and P. A. Bernstein, On the updatability of relational views. *Proceedings of the 4th International Conference on Very Large Data Bases* (1978).
13. M. M. Astrahan *et al.* System R: relational approach to database management. *ACM Transactions on Database Systems* **1**, 97–137 (June 1976).
14. M. Stonebraker, E. Wong, P. Kreps and G. Held, The design and implementation of INGRES. *ACM Transactions on Database Systems* **1**(3), 189–222 (September 1976).
15. A. Pirotte and M. Lacroix, User Interfaces for database application programming. In *Proceedings of the Infotech State of the Art Conference on Database*. Infotech (1980).
16. M. Stonebraker, Retrospective on a database system. *ACM Transactions on Database Systems* **5**, 225–240 (June 1980).
17. M. P. Atkinson, P. Bailey, W. P. Cockshott, K. J. Chisholm and R. Morrison. Progress with persistent programming. In *Database-Role and Structure*, edited P. Stocker, *et al.* Cambridge: Cambridge University Press (1984).
18. P. Buneman and R. E. Frankel, FQL – Functional Query Language. *Proceedings of ACM-SIGMOD International Conference on Management of Data* (1979).
19. M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott and R. Morrison. An approach to persistent programming. *The Computer Journal* **26** (1983).
20. M. P. Atkinson, W. P. Cockshott, P. J. Bailey, K. J. Chisholm and R. Morrison, *PS-Algol Reference Manual*, Technical report PPR-4-83, University of Edinburgh (January 1983).
21. W. Teitelman, *Interlisp Reference Manual*. Xerox Palo Alto Research Center (1974).
22. W. P. Cockshott, M. P. Atkinson, K. J. Chisholm P. J. Bailey and R. Morrison, The persistent object management system. *Software Practice and Experience* **14**, (1983).
23. R. Morrison, *S-Algol Language Reference Manual*, Technical report CS/79/1, University of St Andrews (1979).

APPENDIX A: SYNTAX SPECIFICATION OF EFDM IMPLEMENTATION

command = imperative|
    declare funspec (' − > ' | ' − > > ') typeid|
    define funspec (' − > ' | ' − > > ') fundef|
    constraint consid **on** funlist − >
    (**total** | **fixed** | **unique** | **disjoint** | singleton) |
    view viewid **is**
    {**deduce** funcspec (' − > ' | ' − > > ') typeid **using**
    fundef} **end**.
    drop (funspec|consid|viewid).
imperative = **for each** set imperative|
    for singleton imperative|
    update|**print** stuple.
set − vblid **in** set1
    [**such that**] predicate [**as** typeid]
set 1 = mvfunccall|typeid|'{'stuple'}'|
    '(' set {(**union**|**intersection**|**difference**) set} ')'.
singleton = exp 1 {**or** exp 1}
exp 1 = exp 2 {**and** exp 2}
exp 2 = [**not**] exp 3
exp 3 = exp 4 [compop exp 4]
exp 4 = [prefix] exp 5 {addop exp 5}
exp 5 = exp 6 {mulop exp 6}
exp 6 = exp 7 [**as** typeid]
exp 7 = constant|vblid|svfuncall|aggcall
    the set|**a new** typeid|
    quant set (**has**|**have**) predicate|
    '(' singleton ')'.
svfuncall = funcid '(' stuple ')'.
mvfuncall = funcid '(' mtuple ' '.)
stuple = singleton {',' singleton}.
mtuple = expr { ',' expr}.
expr = set|singleton.
aggcall = (**count**|**max**|**min**) '(' set ')'.
    (**total**|**average**) '(' **over** mtuple singleton ')'.
update = **let** funcall ' = ' expr|
    include (funcall|typeid) ' = ' set|
    exclude (funcall|typeid) ' = ' set|
    delete singleton.
funcall = funcid '(' stuple ')'.
fundef =
        (expr |
        inverse of funcspec |
        transitive of expr |
        compound of tuple).
funcspec = funcid '(' [arglist] ')'.
arglist = typeid { ',' typeid}.
funlist = (typeid | funcid '(' arglist ')' { ','
(typeid | funcid '(' arglist ')') }
compop = ' > ' | ' < ' | ' = ' | ' > = ' | ' < = ' | ' ~ = '.
quant = **some**|**all**|**no**|(**at (least**|**most**)|**exactly**) integer.
integer = singleton.
predicate = singleton.
constant = int|str|bool.
int = digit {digit}.
str = ''' character {character}'''.
bool = **true**|**false**.
vblid = identifier.
typeid = identifier.
funcid = identifier.
consid = identifier.
viewid = identifier.
identifier = letter {(letter|digit|'.')}.
prefix = ' + '|'-'.
addop = ' + '|'-'|' + + '.
mulop = '*'|'/'|**rem**.

Note. Bold-face words, and non-alphanumeric symbols enclosed in quote marks represent terminals. Lower-case words represent syntactic categories. Curly brackets denote repetition. Square brackets denote optionality. Grouping is expressed by parentheses, i.e. (a|b)c stands for ac|bc.