University of Glasgow
Department of Computing Science
Lilybank Gardens
Glasgow G12 8QQ

University of St. Andrews
Department of Computational Science
North Haugh
St Andrews KY16 9SS

EFDM - User Manual

Second Edition

Krishna G. Kulkarni

Persistent Programming
Research Report 7
September 1983

Alan Devlin

PREFACE

This second edition of the EFDM User Manual varies only slightly from the first in that a number of small errors in the examples have been rectified.

EXTENDED FUNCTIONAL DATA MODEL

USER MANUAL

K. G. Kulkarni

Department of Computer Science,

University of Edinburgh,

Mayfield Rd.

Edinburgh EH9 3JZ

PERSISTENT PROGRAMMING

RESEARCH REPORT 7

SEPTEMBER 1983

Table of Contents

1. Introduction

This manual describes the use of the Extended Functional Data Model (EFDM) system. EFDM is a descendent of the functional data model (FDM) and the associated data language, DAPLEX, proposed by Shipman (SHIP81) with some significant extensions and modifications.

FDM models the real-world information as sets of entities and of functions mapping entities to entities. An entity is some form of token identifying a unique object in the database and usually representing a unique object in the real world. For example, FDM models a student in the real world by a unique *student* entity in the database. Entities are classified into a number of *entity types* so that every entity belongs to at least one entity type.

A function maps a given entity into a set of target entities. For example, a function *courseof* may map a particular *student* entity to a set of *course* entities. FDM treats values like integers and strings also as entities, except that such entities have a pre-defined lexicographic representation. Because of this, properties (or attributes) and relationships can both be modelled uniformly as functions mapping entities to entities.

Entity orientation is what distinguishes this data model from other name-based models like relational or record-based data models. Unlike these models, FDM makes an explicit distinction between entities and their external name(s), i.e. more than one entity may have the same set of external names and still be internally distinguishable. For example, if the Christian name and surname are the only information stored about each person, FDM can distinguish between two persons who have the same Christian name and surname while relational or record-based models cannot. This is possible in FDM because it creates two distinct entities corresponding to the two different persons in the real world.

Thus, each entity in the database is created by an explicit 'create' action, signifying the start of interest in the corresponding real-world object for which it serves as the representative. After an entity is created, the values for the functions applicable to it are

assigned by an explicit action, and the changes in properties of the corresponding real-world object with time are reflected by modifying the function values. Finally, when the interest in a certain real-world object ceases, the corresponding entity is removed from the database by an explicit 'delete' operation. Thus entities by themselves cannot be printed, only functions yielding lexical entities like strings or integers can be printed.

Figure 1-1 lists the set of functions used to model a simple student database expressed using the data definition statements of EFDM. The single arrow used in these function definitions indicates that the function is single-valued, i.e. the function application always returns a single entity and the double arrow indicates that the function is multi-valued, i.e. the function application returns a set of entities. The argument part of a function definition indicates the type(s) of argument entities and the result part indicates the type of result entities.

```
declare person ( )                  ->> entity
declare student ( )                 ->> person
declare staff ( )                   ->> person
declare course ( )                  ->> entity
declare event ( )                   ->> entity
declare tutorial ( )                ->> event
declare lecture ( )                 ->> event

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 course (staff)              ->> course
declare room (staff)                ->  string
declare phone (staff)               ->  integer

declare title (course)              ->  string
declare lecture (course)            ->> lecture

declare day (event)                 ->  string
declare time (event)                ->  integer
declare site (event)                ->  string
declare room (event)                ->  string

declare staff (tutorial)            ->  staff

define tutor (student)              ->  staff ( tutorial (student) )
```

Figure 1-1: Functional Schema for a Student Database.

By convention, zero-argument functions define entity types. For example, consider the *person( )* function shown above. Since there are no arguments, this function has only one possible result set. We use this fact to say that all members of this set have a distinct type, i.e. it defines the *person* entity type. Further, the set of *person* entitles returned by the *person( )* function above is a subset of the set of **entity** entities. We use this fact to say that each entity type in this model acts as a *subtype* of its result type. This leads to the hierarchical organisation of entity types so that they are all subtypes of the type **entity.**

An entity type can have any number of subtypes. For example, both *staff* and *student* are subtypes of *person* type. An important consequence of this type hierarchy is that a subtype inherits all the functions defined over all of its supertypes. This follows from the fact that a *student* entity is necessarily a *person* entity as well and hence all the functions applicable to *person* entities are applicable to *student* entities. Also, extensions of different entity types can overlap. For example, a *student* entity can simultaneously be a *staff* entity as well. The notion of *role* (see section 4) is used to determine the type of an entity during evaluation.

As described earlier, functions with arguments model both attributes and relationships. For example, while the *cname*(*person*) function models an attribute of *person* entities, the *course(student)* function models the relationship between *student* and *course* entities. Functions in this model can have more than one argument. For example, the *grade* function above establishes a mapping from each *student- course* pair to a **string** entity.

string and **integer** are pre-defined entity types provided by the system. They are special in that they have an established method of lexicographically representing the instances of these types. The actual set of such built-in types is analogous to the base types in a programming language. Other such useful types are **boolean, real, time, date** etc.

A function introduced by **declare** is called a *base function* and is represented by physically storing a table of arguments and results. For example, *cname(person)*

function relating a person to his Christian name. A function introduced by **define** is called a *derived function* and is represented by an algorithm to compute its result. For example, the result of *tutor(student)* function above is calculated by evaluating the expression *staff(tutorial(student))* for the given *student* entity.

The model allows overloading of function names. For example, though the functions *course( ), course(student)* and *course(staff)* all have the same name, they are distinguished by their internal names. The internal name of a function is generated by enclosing in square brackets the external function name and the argument types over which it is specified.

2. How to Get Started

EFDM provides an interactive user interface. (See Appendix A for the complete syntax.) On initiating the system, EFDM prompts the user for a database name. If a database with the specified name does not exist, the system creates an empty database for the user. After the database is opened successfully, EFDM responds with the prompt:

View:

If the user wants to enter the global view, the response should be

global

If, on the other hand, the user wants to enter a named view of the database (see section 7), he should type the name of the view in response. (If the view corresponding to the user-specified name does not exist, an error message is displayed and the session is terminated.) After the view is established successfully, the system responds with the prompt

command:

If the user is in the global view, he can then type in a FDM command starting with

- **declare** or **define** to add a new function or entity type.
- **for each** to retrieve and display data from the database.
- **for a new** to create a new entity.
- **let, include, exclude** to assign/modify function values.
- **delete** to delete an existing entity.
- **program** to define names queries.
- **print** to output results on to screen.
- **output** to output results of named queries on to files.
- **constraint** to specify a constraint.
- **view** to define a view.
- **drop** to drop an exisiting function, an entity type,
  a named query, a constraint, or a view definition.
- **load** to create a database from data stored in files.

If the user is in any other view, he is restricted to a subset of the above commands. For example, he cannot make any database updates nor declare any base functions (see section 7). The following system conventions must be noted:

1. All statements must be terminated by a semi-colon (;).
2. Each statement is parsed as it is input and if a syntax error is encountered at any stage, all the input is abandoned and the system returns to command mode after displaying the relevant error message.
3. If the statement is syntactically correct, it is executed immediately and if it contains the **print** instruction, the result values, if any, are displayed on the terminal. The system returns to command mode after executing the statement.
4. The interactive session can be terminated by inputting a full-stop in response to **command:**. The whole session is treated as one transaction and all updates done during the session get stored in the database only if the users reply affirmatively to the prompt : **commit transaction?**

The following language notations must also be noted:

1. **Data types**: The language provides a built-in entity type **entity** and supports primitive types of **integer, string, boolean.**
2. **Names**: A name (identifier) consists of a letter followed by any number of dots, letters or digits.
3. **Literals**: Three forms of literals are provided, i.e. integer literals, string literals, and boolean literals. An integer literal is one or more digits. A string literal is any sequence of characters enclosed by double quotes. **true** and **false** are the only two boolean literals allowed.
4. **Tokens** may be separated by an arbitrary number of spaces, new lines, or comments. Comments are arbitrary text enclosed in square brackets [ ]. Upper and lower cases are equivalent.

3. Data Definition

3.1 Function Specification

declare (**define**) statement is used to enter a new base (derived) function or entity type definition. A new function can be declared at any time during an interactive session. The arguments, if any, of the function declaration must correspond to entity type identifiers. For base functions, the result should also correspond to an entity type identifier. (The different mechanisms for defining derived functions are discussed in section 5.)

No forward or recursive function definition is allowed for base functions, i.e. functions (and hence entity types) must be declared before they are used and a function cannot be declared using itself.

To avoid inconsistencies, the users are helped to ensure that the same real world fact is not described by more than one base function. For example, assume that there exists a base function

student (course) ->> student

establishing a relationship between *student* and *course* entity types. Suppose we now want to add another function

course (student) ->> course

which establishes another relationship between *student* and *course* entity types. If, in reality, this new function is the inverse of the old function, adding it as a base function will mean that the same fact is represented by two independently updatable functions, and this will surely lead to inconsistencies in the database. However, it can be added safely as a derived function. On the other hand, the new function may be corresponding to a new fact, say, relating to the majors taken by a student, in which case, it is to be added as a base function. As there is no way for the system to infer what is intended, whenever addition of a new one-argument base function is requested, the system will display all the existing base functions between the argument and result types and some that can be derived using function inversion and composition. The user is consulted to check if the function addition should proceed.

3.2 Constraint Specification

Constraints are specified using the constraint command. Each constraint must be 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 action.

Currently, EFDM supports the following kinds of constraints:

1) The constraint that the *sex (person)* function must be defined for every *person* entity can be specified as follows:

constraint c1 **on** sex (person) -> **total;**

2) The constraint that the value for the *sex (person)* function for a given entity can be assigned only at the time it is created and this value cannot be changed throughout the life time of that entity can be specified as follows:

constraint c2 **on** sex (person) -> **fixed;**

3) The constraint that the values for *cname (person)* and *sname (person)* functions for a given entity together act as unique designators of that entity can be specified as follows:

constraint c3 **on** cname (person), sname (person) -> **unique;**

4) The constraint that no *student* entity can be a *staff* entity as well can be specified as follows:

constraint c4 **on** student, staff -> **disjoint;**

5) The constraint that the *grade (student, course)* function is defined for only those *student-course* pairs such that the *course* entity is a member of the set returned by the application of *course (student)* function to the given *student* entity can be specified as follows:

constraint c5 **on the** grade (student, course) ->
some c in course (student) **has** c = course;

(For the explanation of **some** quantifier, see section 4.)

3.3 Removal of Functions and Constraints

Functions can be removed using **drop** command. For example,

drop course (student);

is a request for dropping the *course(student)* function.

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.

Constraints can also be dropped with the **drop** command. For example, the following command

drop c1;

removes the constraint c1.

3.4 Metadata

The metadata of the schema corresponding to an application is held in a set of EFDM functions shown in figure 3-1. These functions are automatically populated and modified when **declare, define** 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. For example, to find out the textual description of all existing function declarations, the following statement can be made use of:

for each f **in** function
print text (f)

So a user may use such queries to discover the form of a database. To facilitate this the functions given in figure 3-2 are defined.

```
function ( )          ->>  entity    -- contains a set of entities corresponding to the functions
name (function)       ->   string    -- the name of the function
nargs (function)      ->   integer   -- the number of arguments the function has
arguments (function)  ->>  function  -- list of those functions
result (function)     ->   function  -- and the result function
type (function)       ->   string    -- whether the function is single-valued or multi-valued
status (function)     ->   string    -- whether the function is base or derived or system-
                                        defined
text (function)       ->   string    -- the text of the function definition
document (function)   ->   string    -- documentation associated with that function
constraints ( )       ->>  entity    -- contains a set of entities corresponding to the
                                        constraints
name (constraint)     ->   string    -- name of the constraint
text (constraint)     ->   string    -- text of the constraint declaration
```

Figure 3-1: The functions to hold the meta data of a schema

```
entitytype( )              ->>  f in function such that nargs (f) = 0
supertype(entitytype)      ->   result (entitytype)
supertypes(entitytype)     ->>  transitive of supertype (entitytype)
subtype(entitytype)        ->>  e in entitytype such that result (e) = entitytype
subtypes(entitytype)       ->>  transitive of subtype (entitytype)
fnsover(entitytype)        ->>  f in function such that nargs(f) = 0 and some e
                                    in arguments(f) has (e = entitytype or some
                                    e1 in supertypes(entitytype) has e = e1)
fnsyielding (entitytype)   ->>  f in function such that nargs (f) " = 0 and
                                    result(f) = entitytype
```

Figure 3-2: The derived functions for querying meta data

4. Data Retrieval

for-loop statements and expressions are the basic constructs for data retrieval. For example, consider the following query:

(Q1.) Find the Christian and surname of all students.

This is formulated as follows:

```
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.

Expressions consist of names, literals, and operators. Every expression has a *value* and a *role*. The expression value is the set of entities returned by evaluating the expression. An expression which evaluates to a set is called *set expression* while that which evaluates to a single entity is called *singleton expression.* The expression role is 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. This can be any meaningful name chosen by the user. Variables, once declared, remain in scope throughout the statement.

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

(Q2.) Find the Christian and surname of all female students.

```
for each s in student such that
sex(s) = "f"
print cname(s), snames(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.

Quantifiers can be made use of to specify predicates. For example, consider the following query:

(Q3.) 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** (the last three quantifiers must be followed by an integer-valued singleton expression).

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:

(Q4.) 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.

4.1 Operators

4.1.1 **Relational Operators**

An entity compared with another entity evaluates to a boolean result. The following comparison operators are allowed:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
=	equal to
"=	not equal to

= and "= are defined on all entity types. <, <=, > and >= are defined on **integer** and **string** types only. The entities to be compared must be of compatible type, i.e. types which are either the same or related by a subtype-supertype relationship.

4.1.2 **Boolean Operators**

There is one unary operator **not**, and two binary operators **and** and **or** defined on entities of type **boolean**. They have the usual meaning. The order of evaluation is from left to right and based on the precedence rule

```
not
and
or
```

4.1.3 **Arithmetic Operators**

Arithmetic may be performed on entities of type **integer**. The operators are

+	for addition
-	for subtraction
*	for multiplication
/	for division
rem	for remainder

The order of evaluation is from left to right and based on the precedence rule

```
* / rem
+ -
```

i.e. the operations x, /, **rem** are always evaluated before + and -. However, if the operators are of the same precedence then the expression is evaluated left to right. Brackets may be used to override the precedence of the operator.

4.1.4 **String Operators**

Currently, only one operator ++ is defined on entities of type **string**. It concatenates the two operand strings to form a new string.

4.1.5 **Set Operators**

The following binary operators are applicable to combine set expressions:

union for set union
intersection for set intersection
difference for set difference

The order of evaluation is from left to right. The type of expressions involved must be the same.

4.1.6 **the Operator**

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:

(Q5.) 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".

4.1.7 **as Operator**

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

(Q6.) 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);
```


4.2 Aggregation Functions

Aggregation functions include **average, total, count, maximum, minimum.**

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

(Q7.) 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 maxiumum and minimum values, respectively, of the result set. For example, consider the following query:

(Q8.) Find the maximum tutorial group size.

```
print maximum(i in students (t in tutorial) );
```

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:

(Q9.) Find the average tutorial group size.

```
print average (over t in tutorial count (s in students (t) )
```

4.3 Packaged Queries

A query can be named by preceding the query statement with

program *programid* **is**

For example, the following statement:

```
program females is
for each  p in person such that sex (p) = "f"
print cname (p), sname (p);
```

assigns the name *females* to the corresponding query. Such a named query can be executed any time by typing its name in response to the system prompt.

The information about currently defined queries is held in pre-defined EFDM structures. These are shown in figure 4-1.

```
query ( )          ->>      entity
name (query)       ->       string
text (query)       ->       string
```

Figure 4-1: The functions to hold meta data for queries.

These can be queried using the above data retrieval facilities. For example, to list all the existing query definitions, we can use

```
for each q in query
print text (q);
```

Any existing query can be removed using the **drop** statement. For example, the following statement,

drop females;

removes the above query definition.

4.4 Outputting the Results

The **print** statement is used to output the results. The results of both interactive and packaged queries normally appear on the screen.

There is also a facility to output the results of a packaged query to a named file. For example, the following statement

output females fem.dat;

executes the query named *females,* creates a file with the name *fem.dat* if it is not already created, and directs the output to that file instead of the screen.

5. Derived Functions

The **define** command is used to enter a derived function. Derived functions can be defined using expressions. For example, consider the function *tutor(student)* as defined in the schema of figure 1-1:

define tutor(student) -> staff(tutorial (student))

which returns a *staff* entity for a given *student* entity by evaluating the defining expression.

In the above example, the argument of the function implicitly declares a variable with the same name for use during evaluation. In case of ambiguity, variables can be associated with the arguments using **in.** For example, the following derived entity type

define female ( ) ->> p **in** person **such that** sex (p) = "f";

defines a subtype of type *person* , which returns those *person* entities meeting the qualification. It is to be noted that the use of set expressions involving set operators is currently not allowed for defining new entity types.

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 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) total.cost(p) );
```

Additionally, the following special operators are provided for defining derived functions:

Inverse of :

If we are interested in finding out the set of students taking a given course using the *course(student)* function, we can define a derived function as follows:

define students(course) ->> s **in** student **such that**
some c **in** course(s) **has**
c = course

EFDM provides a special operator **inverse of** to simplify the derivation of the above function. Using this operator, the above function can be defined as

define students(course) ->> **inverse of** course(student)

Inverse functions can be defined for one-argument functions only. A single valued function may have the inverse function which is single- or multi-valued. Similarly, a multi-valued function may have a single- or multi-valued function as its inverse.

transitive of :

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

define subparts(part) ->> **transitive of** p **in** 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 **transitive of** operator can be used to define one-argument functions only. The argument and the result of such a function must be of identical type.

compound of :

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** s **in** student, c **in** course (s)

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.

6. Database Updating

Using the update operations of EFDM, the users can create or delete an entity, assign or modify function values, and extend or reduce the set of types for an existing entity. Currently, updating the database through derived functions is not supported. Also, as there is no notion of *transactions,* checking for constraint violations following an update action is also not supported.

6.1 Creating a new entity

A new entity is created using **a new** expression. For example,

a new p **in** person

creates a new *person* entity and associated it with the variable *p*. When a new entity belonging to a certain entity type is created, all the supertypes of that entity type get populated simultaneously with that new entity. For example,

a new s **in** student

creates a new entity which is included in the extension of both *student* and *person* entity types.

To assign values for the functions applicable to the newly created entity, the **for** statement along with the **let** statement (see section 6.2) may be made use of. It is not necessary to populate all the functions applicable to the newly created entity at the time of creation itself. This is because the functions in EFDM are assumed to be partial by default. All unassigned single-valued functions will be initialised with a special entity 'UNDEFINED' and all unassigned multi-valued functions will be initialised with the empty set. However, to be able to assign values at a later time, care should be taken to see that the newly created entity can be uniquely identified.

6.2 Assigning or modifying function values

let, include, and **exclude** statements are used for this purpose. The **let** statement replaces the existing function value, if any, by the new value. Otherwise, it assigns the specified value as the result for the given set of arguments. The **include**

statement extends the existing result set of a multi-valued function for the given arguments and the **exclude** statement does the opposite. For example, consider the following requests:

(U1.) 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";
```

creates a new *student* entity and assigns the specified values to the functions applicable to that entity.

(U2.) For the student with Christian name 'Moyana' and surname 'Johns', change her current course assignment to CS1 and CS2 courses.

```
for the s in student such that
cname(s) = "Moyana" and sname(s) = "Johns"
let course(s) = c in (the c1 in course such that
            title (c1) = "CS1", the c2 in course
            such that title (c2) = "CS2");
```

(U3.) For the student with Christian name 'Moyana' and surname 'Johns', add the course IS1 to her current course assignments

```
for the s in student such that
    cname(s) = "Moyana" and sname(s) = "Johns"
    include course (s) = c in (the c1 in course
        such that title (c1) = "CS1");
```

(U4.) For the student with Christian name 'Moyana' and surname 'Johns', drop the course CS1 from her current course assignment

```
for the s in student such that
   cname(s) = "Moyana" and sname(s) = "Johns"
     exclude course (s) = c in (the c1 in course
        such that title (c1) = "CS1");
```


The following points should be noted:

1. The type of the expression in the right-hand side of the = sign must be the same as the type of the left-hand side function.

2. The expression must evaluate to a singleton for single-valued functions and to a set for multi-valued functions.

3. The inclusion of an already existing element into a set has no net effect.

4. The exclusion of a non-existant element from a set has no net effect.

6.3 Extending the set of types of an entity.

The syntax for including an existing entity into the extension of a specified entity type is the same as that used to include entities into multi-valued function extensions. For example, to include a student with Christian name 'Moyana' and surname 'Johns' into *staff* type, we use

```
include staff = s1 in (the s) in student such that
    cname (s1) = "Moyana" and sname (s1) = "Johns")
```

Inclusion of an entity in the extension of its own type or any of its supertypes has no net effect.

6.4 Reducing the set of types of an entity.

The syntax for this case also is the same as that used to exclude entities from the multi-valued function extensions. For example, to exclude the staff with Christian name 'Moyana' and surname 'Johns' from the *staff* type, we use

```
exclude staff = s1 in (the s1 in staff 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 in which it is participating either as an argument or result. Hence, before carrying out the operation, a list of these implicit updates is displayed and the user is asked to confirm the request.

It is to be noted that reducing the set of types of an entity does not result in the removal of entity from the database as long as the set of types is non-empty. For example, the above operation only removes the specified entity from the *staff* type, while the entity itself continues to exist in the database as a *person* and possibly as a *student* entity.

6.5 Entity Deletion

The **delete** statement is used to remove a specified entity from the database. This also causes a cascade deletion of all functions which reference this entity, again consulting the user before the cascade proceeds. For example, to delete a student with Christian name 'Moyana' and surname 'Johns' from the database, we use

```
delete the s in student such that
    cname(s) = "Moyana" and sname(s) = "Johns";
```

The difference between this statement and the **exclude** statement above is that while exclude removes a specified entity from the specified type and its subtypes, **delete** removes a specified entity from the extensions of all its types.

7. 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. The underlying assumptions of this mechanisms are:

1) There exists a *global* view from which all user views are derived. That is, we assume the structure and contents of this global view are arrived at by integrating the different application views to one common community view. Hence, the global view encompasses all the information required by all the users. If certain information required by a user is not in his view, he must request a central authority (database administrator) to include it in his view, who will, in turn, examine whether the information requested by him is already available in the global view and if not, he will take steps to include it in the global view and then include it in the user's view.

2) The users of a certain view are restricted to the names available in the namespace of that view; they have no access to either the global name space or the namespaces of other views.

Views are defined using **deduce** statements. For example, for the student database of figure 1-1, we can define a view called *malestudents* as

```
view malestudents is

    deduce male ( )       ->> entity using s in student
                                  such that sex(s) = "m";
    deduce name(male) ->      string using cname/sname(s);
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 **using** keyword refer to names in the current namespace. Every view definition automatically creates a

different name space, which is completely independent of the global name space as well as the name spaces of other views.

Views can be dropped with the

drop *viewname*

command.

The information about currently defined views is held in pre-defined EFDM structures. These are shown in figure 7-1.

```
view ( )          ->> entity
name (view) ->    string
text (view)       ->> string
```

Figure 7-1: The functions to hold the meta data for views

7. 1 Operations from Views

Once within a view, users can pose the usual EFDM requests to carry out the database operations using the names available in their name space. Each query statement issued from a user view is translated into a corresponding query on the global name space by recursively applying the view definition mapping.

Since all the functions in a view are treated as derived functions, updating through view functions is currently not allowed. The only schema changes that are allowed from a view are either defining derived functions from the set of functions they are allowed to have access to or dropping the derived functions they have created. This means that the individual users are not allowed to introduce any base functions or stored data of their own.

8. Bulk Loading of a Functional Data Base

The **load** command is provided to load function definitions and function values stored in files. On entering this command, the user is prompted for the names of two files, one containing the function declaration statements (schema file), and the other containing the function values (data file).

A series of **declare, define** or **program** statements, each terminated with a semicolon constitute the schema file. Note, however, that for the purpose of using this command, it is necessary for each entity type to have a single-valued *key* function (with **'integer'** or **'string'** as its result) whose values uniquely distinguish the entities belonging to that type. We will assume that the following functions are defined for this purpose: *personno* for *person* type, *studentno* for *student* type, *staffno* for *staff* type, *courseno* for *course* type, each having **integer** type as result. The last statement of the schema file must be a full stop (.).

The data file should be organised as a set of tables. Each table must be qualified as either E-table or A-table. Each entry in an E-table results in creating a new entity and assigning values for some specified functions applicable to that entity. In contrast, each entry in an A-table either assigns or modifies the function value for an existing entity, identified by its key function value. End of all data is indicated by an *.

The name of an E-table must correspond to the name of an entity type specified in the schema. Each column header of an E-table must correspond to the name of a single-valued, one-argument function defined on that entity type or any of its supertypes which yields entities of primitive types like **string, integer** etc. Any number of such columns may be present in an E-table. The format for an E-Table is as follows: Its name is followed (after a space) by E to indicate the nature of the table. The different column headers must be separated by spaces, and followed by an * to indicate the end of the list of headers. There then follow rows of values (strings, integers etc.); each row conveys information about the values of the various functions of a new entity, identified by the value of the *key* function. After the rows, an * denotes the end of the table.

For example, E-tables corresponding to *student* and *staff* types of the database shown in Fig. 1-1 appear as shown below:

```
student E
studentno   cname        sname        sex  *
1            Angela      Pearson      f
-            -           -            -
*
staff E
staffno     cname        sname        sex  *
1            Malcolm     Atkinson     m
-           -            -            -
*
```

The name of an A-table must correspond to the name of a function with one or more arguments. The column headers for an A-table correspond to the key functions of each of the arguments of the function in the order of the argument position. The last column header corresponds to the key function of the result type if it is an user-defined type or the primitive type names like **'string'**, **'integer'** etc. otherwise. The format for an A-Table is as follows: Its name is followed (after a space) by A to indicate the nature of the table. The different column headers must be separated by spaces, and followed by an * to indicate the end of the list of headers. There then follow rows of values (strings, integers etc.); each row conveys the information about the association between two or more existing entities identified by the values for the corresponding key functions. After the rows of values, an * denotes the end of the table.

For example, A-Tables corresponding to *course(student)* and *grade(student, course)* functions appear as shown below:

```
course A
studentno (student)      courseno (course)   *
1                        1
1                        2
-                        -
*
grade A
studentno (student)      courseno (course)   string   *
1                        1                   A
1                        2                   B
-                        -                   -
*
```

Reference

SHIP81 Shipman D.W. "The Functional Model and the Data Language DAPLEX" ACM Transactions on Data Base Systems vol. 6 no. 1, March 1981.

Appendix A: Syntax Specification of EFDM Implementation

command = *imperative* |
declare *funspec* ("->" | "->>") *typeid* |
define *funspec* ("->" | "->>") *fundef* |
constraint *identifier* on *funlist* ->
(**total** | **fixed** | **unique** | **disjoint** | *singleton*) |
program *programid* **is** *imperative* |
output *programid fileid* |
view *viewid* **is**
(**deduce** *funcspec* ("->" | "->>") *typeid* **using** *fundef*) *end.*
drop (*funspec*|*programid*|*viewid*) |
load |
programid.

imperative = **for each** *set imperative* | **for** *singleton imperative* |
update | **print** *stuple*

set = *vblid* **in** *set1*[**such that** *predicate*] [**as** *typeid*]

set1 = *mvfuncall* | *typeid* | "(" *stuple* ")" |
"(" *set* ((**union** | **intersection** | **difference**) *set*) ")"

singleton = *exp1* [**or** *exp1*]

exp1 = *exp2* [**and** *exp2*]

exp2 = [**not**] *exp3*

exp3 = *exp4* [*compop exp4*]

exp4 = [*prefix*]*exp5* (*addop exp5*)

exp5 = *exp6* [*mulop exp6*]

exp6 = *exp7* [**as** *typeid*]

exp7 = *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* | **compund of** *tuple*)

funcspec = *funcid* "(" [*arglist*] ")"

arglist = *typeid* ["," *typeid*]

funlist = (*typeid* | *fumcid* "(" *arglist* ")") { "," (*typeid* | *funcid* "(" *arglist* ")") }

compop = ">" | "<" | "=" | ">=" | "<=" | "~"

quant = **some** | **all** | not | (**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*

programid = *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 italic words represent syntactic categories. Square brackets denote optionality. Grouping is expressed by parentheses, i.e. (a | b) c stands for ac | bc.

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

or

The Secretary,
Persistent Programming Research Group,
Department of Computational Science,
University of St. Andrews,
North Haugh,
St. Andrews KY16 9SS
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", Pergammon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

Cole, A.J. & Morrison, R.
"An introduction to programming with S-algol", Cambridge University Press, Cambridge, England, 1982.

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages". Software, Practice and Experience 7, 5 (July 1977), 535-537.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", Proc. of the DECUS conference, Bath (April 1978), 5-10.

Atkinson, M.P.
"A note on the application of differential files to computer aided design", ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

Atkinson, M.P.
"Progress in documentation: Database management systems in library automation and information retrieval", Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", Information Processing Letters 9, 1 (July 1979), 1-4.

Atkinson, M.P.
"Data management for interactive graphics", Proceedings of the Infotech State of the Art Conference, October 1979. Available as EUCS departmental report CSR-51-80.

Atkinson, M.P. (ed.)
"Data design", Infotech State of the Art Report, Series 7, No.4, May 1980.

Morrison, R.
"Low cost computer graphics for micro computers", Software Practice and Experience, 12, 1981, 767-776.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in Database, Pergammon Infotech State of the Art Report, Series 9, No.8, 299-318 (January 1982) - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", Computer Bulletin II/31 (March 1982).

Morrison, R.
"The string as a simple data type", Sigplan Notices, Vol.17,3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure", see PPRR-8-84.

Morrison, R.
"Towards simpler programming languages: S-algol", IUCC Bulletin 4, 3 (October 1982), 130-133.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University of Pennsylvania. October 1982. Circulated (revised) in the Workshop proceedings 1983, see PPRR-2-83.

Atkinson, M.P.
"Data management", in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3 (March 1983), 273-285. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983, Vol.26, No.4, 360-365 - see PPRR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983, 70-79 - see PPRR-2-83.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D.
Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu,
A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", proceedings of the third British National Conference on Databases, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, (July 1984).

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985) - see PPRR-9-84.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", 8th International Conference on Software Engineering, Imperial College, London (August 1985), 166-172 - see PPRR-15-85.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24 - see PPRR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 278-283 - see PPRR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117 - see PPRR-16-85.

Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303 - see PPRR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383 - see PPRR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 399-410 - see PPRR-16-85.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438 - see PPRR-16-85.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, Vol.14, No.3, (1986) - see PPRR-10-84.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", proceedings of the 19th Annual Hawaii International Conference on System Sciences, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA - see PPRR-19-85.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", proceedings of the 9th Australian Computing Science Conference, January, 1986 - see PPRR-21-85.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages"; proceedings ACM SIGMOD Conference 1986, Washington, USA May 1986 - see PPRR-22-86.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 147-157 - see PPRR-14-86.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", proceedings of Information Processing 1986, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics", University Computing, Vol. 8, N0. 2, (Summer 1986) - see PPRR-23-86.

Kulkarni, K.G. & Atkinson, M. P.
"Implementing an Extended Functional Data Model Using PS-algol", Software - Practise and Experience, Vol. 17(3), 171-185 (March 1987)

Cooper, R.L. & Atkinson, M.P.
"The Advantages of a Unified Treatment of Data", Software Tool 87: Improving Tools, Advance Computing Series, 8, 89-96, Online Publications, June 1987.

Internal Reports

Morrison, R.
"S-Algol language reference manual", University of St Andrews CS-79-1, 1979.

Bailey, P.J., Maritz, P. & Morrison, R.
"The S-algol abstract machine", University of St Andrews CS-80-2, 1980.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
"EDQUSE reference manual", Department of Computer Science, University of Edinburgh, September 1981.

Hepp, P.E. and Norrie, M.C.
"RAQUEL: User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

Theses

The following theses, for the degree of Ph. D. unless otherwise stated, have been produced by members of the group and are available from the address already given,

W.P. Cockshott
Orthogonal Persistence, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

J. Livingstone
Graphical Manipulation in Programming Languages: Some Experiments, M.Sc., University of Glasgow, 1987

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those which have been produced at 9th July 1986. Copies of documents in this list may be obtained by writing to the addresses already given.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., Chisholm, K.J. and Cockshott, W.P.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-4-83	The PS-algol reference manual - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R	Presently no longer available
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P.,Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00
PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-87	PS-algol Reference Manual - fourth edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.,	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - second edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for an Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00
PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-86	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00
PPRR-25-87	Applications Programming in PS-algol - Cooper, R.L.	£1.00
PPRR-26-86	Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P.	£1.00
PPRR-27-87	A Context Sensitive Addressing Model - Hurst, A.J.	£1.00
PPRR-28-86b	A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A.	£1.00
PPRR-29-86	A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P	£1.00
PPRR-30-86	Data Types for Data Base Programming - Buneman, O.P.	£1.00
PPRR-31-86	An Introduction to PS-algol Programming - Carrick, R., Cole, A.J. & Morrison, R.	£1.00
PPRR-32-87	Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment - Morrison, R, Brown, A, Connor, R and Dearle, A	£1.00
PPRR-33-87	Safe Browsing in a Strongly Typed Persistent Environment - Dearle, A and Brown, A.L.	£1.00

PPRR-34-87	Constructing Database Systems in a Persistent Environment - Cooper, R.L., Atkinson, M.P., Dearle, A. and Abderrahmane, D.	£1.00
PPRR-35-87	A Persistent Architecture Intermediate Language - Dearle, A.	£1.00
PPRR-36-87	Persistent Information Architectures - Atkinson, M.P., Morrison R. & Pratten, G.D.	£1.00
PPRR-37-87	PS-algol Machine Monitoring - Loboz, Z.	£1.00
PPRR-38-87	Flexible Incremental Bindings in a Persistent Object Store - Morrison, R., Atkinson, M.P. and Dearle, A.	£1.00
PPRR-39-87	Polymorphic Persistent Processes - Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R., Dearle, A., Hurst, A.J.and Livesey, M.J.	£1.00
PPRR-40-87	Andrew, Unix and Educational Computing - Hansen, W. J.	£1.00
PPRR-41-87	Factors that Affect Reading and Writing with Personal Computers and Workstations - Hansen, W. J. and Haas, C.	£1.00
PPRR-42-87	A Practical Algebra for Substring Expressions - Hansen, W. J.	£1.00
PPRR-43-87	The NESS Reference Manual - Hansen, W. J.	£1.00