University of Edinburgh
Department of Computer Science

James Clerk Maxwell Building
The King's Buildings, Edinburgh

University of St Andrews
Department of Computational Science

North Haugh
St Andrews, Fife

The Persistent Object Management System

M.P. Atkinson
P.J. Bailey
K.J. Chisholm
W.P. Cockshott
R. Morrison

8-1-84.

POMS MANUAL

DATA CURATOR
DOCUMENTATION

22 March 1983

Table of Contents

List of Figures

ABSTRACT

This is a description of the integrated Persistent Object Management System (POMS) for the language PS-algol. The objective of POMS is to provide an implementation of the PS-algol persistent heap[1] entirely by means of procedures written in PS-algol.

1. INTRODUCTION

High level languages provide programmers with abstractions which allow them to organise the store of a computer without having to worry about how to allocate data items to storage locations. In languages with dynamic storage allocation, it is not even necessary to know, when writing an algorithm, just how much data it will use, instead, the algorithm itself decides this as it runs. As anyone who has used a language like Algol-68[2] or S-algol[3] which have fully dynamic storage allocation will realise, this adds greatly to the productivity of programmers.

The catch is that high level languages only organise one type of store: RAM. They ignore disk store, leaving that to the operating system to organise. But programmers cannot ignore disks. RAM is volatile whereas one of the main uses of computers is to store information. As a result we have to spend a lot of time and energy organising the preservation of data structures on disk. In RAM our favourite language allows us to easily build up complex logical structures which we are then forced to output to the essentially serial media of files. Direct access files are some help, but it is hard to map a RAM data structure containing pointers onto even a direct access file. One difficulty, at least with most operating systems, is ensuring

that most transactions commit or leave the file unchanged.

It would be nice if we did not have to go to all the trouble of thinking how we were going to store our information on disk. It would be nice if the data structures that we were using in our program could somehow be saved on disk without our having to make any special effort. If a high level language can organise RAM it should be able to do the same for disk. There is no reason why we as programmers should have to treat the two types of store differently.

The Persistent Object Management System (POMS for short) is a package that allows programmers using PS-algol[1] to write programs without being aware of the distinction between RAM and disk. Any data structure that they build up on the heap can be automatically transfered to disk at the end of a program and then brought back onto the heap the next time the data is used by a program. This enables us to abstract from the physical properties of disk store just as we already abstract from the physical properties of RAM, allowing us to view both with a uniform set of conceptual abstractions. Thus we have a uniform view of all the store avaliable to the programmer.

The POMS is part of a research program to provide a persistent Von Neumann store for high level languages. Earlier implementations of such a store are documented in[45]. Language design issues for such an environment are discussed in [6].

2. WAYS OF IMPLEMENTING PERSISTENT STORE

The mechanisms used by high level languages to organise RAM are quite complex, and have provided material for whole books[7]. Doing the same for disk store in a high level language is, as we shall see, perhaps even more complicated. This explanation of the POMS is therefore presented using the method of repeated elaboration. We start off by giving a simple and rather incomplete picture of the implementation and then present successively more complex pictures. This enables us to present the new concepts involved one at a time.

2.1. WHAT TO KEEP

Languages with a heap are already faced with the problem of deciding what to keep. When the heap fills up, they either behave irresponsibly or they face up to their responsibilities and carry out a garbage collection. In the latter case, they have to decide which objects on the heap to keep and which can have their space reclaimed. The guiding principle of garbage collection is that no object that is reachable from identifiers currently in scope may be reclaimed.

The same principle can be used for the run time system of a language that organises both volatile and persistent store. If we assume that at least one identifier has a scope that is global to all runs of a program, or even to all programs, then we can say that at the end of a program nothing reachable from this identifier must be allowed to evaporate. Conversely, everything reachable from it must be saved by being transfered to disk. How can this be done?

2.2. COPY THE HEAP TO A FILE

The simplest method of providing a language with a persistent store is to just copy the whole heap out to a file. This often done in Lisp[8]. The fact that Lisp programmers can so easily store their data may be one reason for the popularity of the language. This method of providing persistent storage is simple. Simplicity is a virtue, but it carries a price, with this technique for intstance:

- The heap must always be loaded onto the same location, or pointers held on the heap become invalid. If the heap starts above the program, then any editing of the program which changes its size, will invlidate all of the pointers. In Lisp this is no problem as program and data both sit in the heap but this means giving up the freedom that you get with an ordinary filing system of running several programs against one file or one program against several files.

- It is impossible to store more data than you can load onto the heap at any one time. Your maximum database is limited by your machines physical or virtual memory. Although modern computers often have very large virtual address spaces, you may find that only a tiny fraction of this can actually be used due the limited size of the page swapping files.

- Programs which make only a few small changes to a large body of stored data pay a high i/o penalty because they have to load the whole heap at the start and save it again at the end of a session.

In an attempt to get over these problems POMS uses an incremental loading and storing strategy.

2.3. INCREMENTAL LOADING AND STORING

In an incremental strategy, the run time system maintains two separate heaps, one on disk and one in RAM. We will refer to a disk based heap as a database. Along with two types of heap come two types of address. The existence of and interaction between these two types of address are central to the operation of the POMS.

The first type of address we term a Persistent Identifier (abrieviated to PID). A PID describes where an object is to be found on disk. In its simplest form it is just an offset in words or bytes from the start of a file.

The second type of address is whatever the underlying machine supports, either a virtual or a physical address. We term these Local Addresses because their validity is localised in time and space to the particular machine and process on which the program runs.

In the incremental system, when a program has finished, or in database terms when it commits, the run time system copies objects from the local heap in RAM to the persistent heap on disk. As each object is copied from RAM to disk all the pointers that it contains that are in the form of local addresses are converted to pointers in the form of PIDs.

When the same, (or some other) program is run against this database on disk, a distinguished object is copied from the database to the local heap and assigned to a global pointer variable. This object, termed the *Root* of the database contains pointers to other items in the database. It might for example be the top of some sort of tree or the head of a list. If the program tries to dereference a pointer field in the Root what it picks up is a PID, which points onto the disk. It is thus necessary that PIDs be distinguishable from Local Addresses.

Several ways of distinguishing the two are possible. You could make all local addresses word aligned and all PIDs odd byte aligned. In POMS we have chosen to make all PIDs negative and to locate our heap in the bottom half of our machine's address space ensuring that all local addresses will be positive as shown in fig .

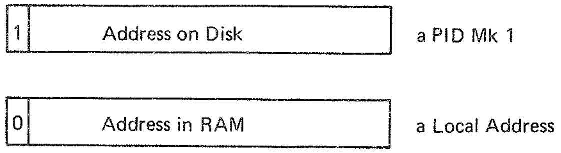

Figure 1:
The essential difference between the two types of address

A dereference instruction can therefore trap all negative addresses and escape into the POMS to deal with them. The POMS fetches the object from disk, loads it onto the heap and overwrites the negative address that had been dereferenced with the new local address of the object on the heap. The dereference instruction can then continue successfully. By this means objects are only brought in when needed and a program that only needs a few persistent objects will not pay a large I/O penalty.

You can also have a much bigger database than it is possible load into RAM as only parts of it need be loaded in any given session.

2.4. THE PIDLAM

The kernel of the POMS is the process of translating PIDs to local addresses when address faults occur, and back again when objects are sent to disk.

Two functions are used to do this translation:

loca.of.pid(**int** pid ->**int**)

pid.of.loca(**int** loca ->**int**)

which map from PIDs to local addresses and back again. These work by means of a data structure called the Persistent IDentifier to Local Address Map (PIDLAM for short). This is a two way index between the two types of addresses implemented by means of two hash tables from which intersecting linked lists are appended a shown in fig 2.

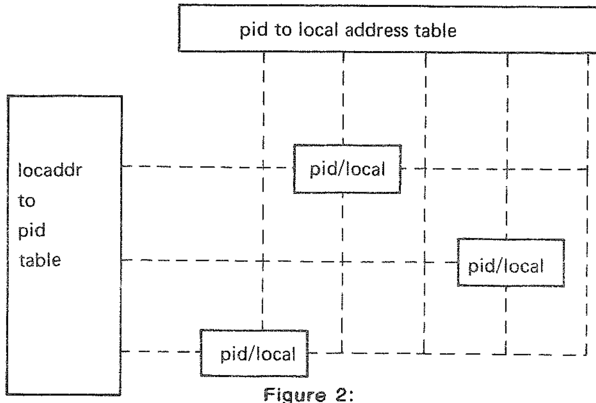

Figure 2:
The organisation of the PIDLAM

The cells on these links contain two fields, a PID and the equivalent local address. Given either of these, the other can be found by hashing into the appropriate table and following down a linked list until the matching entry is found.

When an illegal local address is found the algorithm in 3 is executed.

```
procedure illegal.loca(int pid->int)
begin
  let loca:=loca.of.pid(pid)
  if loca>0 then loca else begin
     loca:=fetch.object(pid)
     Def.pid(loca,pid)
     loca
  end
end
```

Figure 3: Dealing with illegal local addresses

Here it is assumed that *fetch.object* loads the object from disk onto the heap returning its local address and that *Def.pid* enters its parameters into the PIDLAM. Observe that we only need to fetch objects that we have not already fetched and thus have not entered into the PIDLAM.

The legal address returned by *illegal.loca* is used to overwrite the PID in the field that was being dereferenced thus ensuring that the next time this path through the data is followed there will be no delays.

2.5. TRANSACTIONS

The incremental strategy outlined above gets over the problems involved in the simple technique of just copying the heap to a file, but does not support atomic transactions.

An atomic transaction is a transformation of a database from one consistent state to another. Clearly in carrying this out it may be necessary to go through some inconsistent state or states but this must be hidden from the point of view of the external world. The external view of a transaction must be that it has either occurred or not occurred. This must remain true irrespective of possible hardware failures, or programs crashing.

In the model of persistence given above, what would happen if the program or machine failed whilst sending objects back to disk from the heap?

You could have a situation in which some of the data had been written back with new values, whilst other parts of the data still had their old values. The result would be an inconsistent state. If you were copying the whole heap back to a file it is possible to get round this by always copying to a new file with a different name. Once the copy is complete you can delete the old file and give the new file the old name. How are we to achieve the same effects with the incremental strategy?

The general approach that is adopted in POMS to provide secure atomic transactions is to use shadow pages[9]. When data is written back to disk during a transaction it is written to different sites on disk from where it came from. This means that no data that was valid at the start of the transaction is overwritten during the transaction. Thus if the transaction aborts, the previous consistent state is still available on disk. In doing this we exploit the fact that disk transfers occur in fixed sized units, the disk blocks. An address within a file can thus be thought of as having two components:

- the block number

- the offset within the block

We now distinguish between *logical* blocks and *relative* blocks. A relative block occupies a fixed position within the file and is identified by a ***Relative Block Number*** which is simply its offset from the start of the file. A logical block, characterised by a ***Logical Block Number*** may occupy different positions in the file. Suppose we have a table the ***LR.map***

which maps logical to relative block numbers such that

let L be a logical block number
then ***LR.map(L)*** is the corresponding relative block number

We now refine the definition of a persistent identifier so that it now looks like fig 4.

1	Offset	Logical Block Number

Figure 4: Persistent Identifier MK II

So we now address objects on disk in terms of their logical block and an offset from the start of that block.

In order to provide security, every transaction that modifies a Logical Block will cause it to be written back to a different Relative Block. The ***LR.map*** holds the currently valid mapping between Logical and Relative Blocks within a database. In order to securely commit a transaction three phases are executed:

1. Determine which items on the heap have been changed in the course of a transaction.

2. Assign a new relative block to each logical block containing the modified items and write the block containing the updated item out to the new relative block.

3. Update the ***LR.map*** and write that back to disk

Any failure in the first two phases will leave an unmodified ***LR.map*** on disk so that the transaction will not have occured. Any program that now opens the database will see it through the old map which will not reveal any of the changes made in phase 2. This technique was originally proposed in[10].

By implementing an ***LR.map*** we create two types of address space on disk: relative and logical. Of these, the second has the vital property of being transactionally secure. Changes in the state of the logical address space occur as atomic transactions, so long as changes in the state of the ***LR.map*** are themselves atomic transactions.

2.5.1. WHAT IS THE LR MAP

In principle the *LR.map* is just a large array acting as a lookup table. In practice this approach needs a slight modification. POMS allows for of the order of 2^{15} logical blocks in a database. If a single vector of this size was used to implement the map, small databases would carry an unnecessarily high mapping overhead. Moreover writing a large *LR.map* spread over several blocks back to disk would itself be an unsafe operation. If a failure occured whilst it was being written you could get an *LR.map* some blocks of which belonged to a new and others of which belonged to an old state of the database.

The problem is solved by dividing the *LR.map* into two parts, a fixed part and an extensible part. The fixed part is small enough to fit into a single disk block. Then there is an extensible part. If the fixed part holds the first *n* blocks' L-R mapping, then all logical blocks in the range n to **max.blocks** will be mapped via the extensible part.

PS-algol implements arrays as Illife vectors. We can therefore represent the extensible part as a partially populated three dimensional array of integers.

```
let LR.map=vector 1..8 of
             vector 0..lr.lev.size of
             vector 0..lr.lev.size of 0
```

Where the *lr.lev.size* is chosen so as to enable two of the sub vectors of the array to fit into a disk block.

2.5.2. LR MAP IN LOGICAL SPACE

An important implication follows from the decision to implement the *LR.map* in two parts: the extensible part must exist in *logical* disk address space. In part this follows ineluctably from the decision to represent the *LR.map* as a PS.algol array. On disk PS.algol objects have their pointers as PIDs. PIDs define points in logical disk address space. Thus the *LR.map* extension must exist in logical address space. But this is also a necessity if transactions are to be safe.

Recall that all state changes of logical address space are atomic transactions. If we place the *LR.map* extension in logical rather than relative address space then changes in its state too will become atomic transactions.

But hold on! This argument seems recursive: logical space is transactionally secure if the *LR.map* is and the *LR.map* is made secure by being placed in logical space. Where is our fixed point?

It is the fixed portion of the *LR.map*. This all fits within one disk block. If we assume that writes of a single block to disk are transactions then changes in the state of the fixed portion of the *LR.map* will be atomic. If the transactional properties of the extensible part of the *LR.map* can be made to depend upon those of the fixed part we are safe.

To do this the following condition must hold:

If an object A is itself part of the mapping system, the lowest logical block number in whose mapping it participates must be higher than the logical block number of A itself.

This ensures the transactional security of the *LR.map* and thus of the whole logical disk address space.

In the POMS we chose to place the first *n* entries in the *LR.map* into the database Root mentioned earlier. Given the depth of the *LR.map* tree and the size of its nodes, it turns out that an *n* of 3 will be enough to meet the security condition, because the *LR.map* is of depth 3.

3. TYPES

The persistent storage of program data creates problems for the typechecking mechanism of a language that supports it. In conventional programming languages the compiler writer can assume that any data encountered in RAM will belong to some type declared in the program under compilation or in some module to which it is going to be linked. It is thus possible to implement strong typing on the premise that the compiler is omniscient. Once you start storing data on disk, potentially for a long time, this assumption breaks down. Data may persist on disk long after the program which operated on it has been deleted. The normal relationship between the durability of data and the durability of programs is inverted.

Unless we are just going to give up on type checking, we need to find new methods of implementing it. We have chosen to implement a more sophisticated run time type checker than is normal, so that the run time system takes over some of the responsibilities previously carried by the compiler.

3.1. SALGOL TYPE CHECKING

PS-algol is based upon S-algol and its type checking system derives from that of S-algol. The S-algol type checking scheme is based upon a mixture of compile time and run time type checks. S-algol supports both what are termed types, and what are termed structure classes. It has an infinite set of types obtained from the base types integer, pointer, boolean, real, string, file under the operations of proceduring and vectoring.

The values assumed by expressions of type pointer are members of structure classes, which are tuples of named fields. A value of any structure class can be assigned to any pointer. It can not be determined at compile time what class the value of an expression of type pointer will belong to. There are however, operations in the language which require that the class membership of a pointer be known at compile time: the operations of field subscriptions, and subscripted assignment.

The task of the type checking system in S-algol is to ensure that no identifier ever has a value assigned to it that is different from the type that it was declared to have. When a pointer is subsripted or has a field assigned to, it is necessary to be sure that the field has the type required in this context or else an assignment might occur that would violate the type rules.

In S-algol no two structure classes in the same scope can share a field name. The compiler can thus deduce from the field name used in a subscription what class the subscripted pointer will belong to, and plant code to carry out a run time type check.

In the course of processing a program the compiler assigns to every structure class that it encounters a unique number termed a *Trade Mark*. In non persistent implementations of the language, this trade mark is an adequate identifier for the class. The headers of instances of the class, in these implementations, incorporate a copy of the trade mark. When the compiler needs to check what class a pointer belongs to, it plants code to compare the header of the object pointed to with a known Trade Mark. In persistent implementations, this technique is not feasible, because it is a requirement that different programs, compiled independently, should be able to share data created at run time. Since the S-algol compiler assigns Trade Marks in ascending order of the textual occurences of structure class declarations, the only method of ensuring that the same classes were declared in the same order in the two programs. This approach, used in early implementations of PS-algol was judged to be unsafe for persistent data.

3.2. THE DATABASE APPROACH

An alternative approach, in line with conventional database practice, would be to maintain a schema database that the compiler would access and in which all past declarations of classes would be held. If the compiler found that a class in the program under compilation had already been inserted in the schema, it would obtain

the trademark previously associated with the class from the database, otherwise it would insert the class description into the database, increment a counter in the schema to obtain a new trademark and use this in the program. This approach was rejected because:

- The PS-algol compiler is a slightly modified version of the S-algol compiler. It is itself implemented in S-algol not PS-algol and is thus incapable of accessing a PS-algol database.

- The use of such an approach might lead to undue contention in accessing the schema database.

- It would make PS-algol program object code and databases non-portable because the meaning of a trade mark would be specific to the schema on the machine on which the program was compiled.

3.3. THE POMS APPROACH

Instead it was decided that both program object code and databases should carry complete descriptions of the classes declared or stored within them. POMS uses a tagged data architecture in which each object on the heap is either self describing or contains enough information to access a description of it. By tagging objects it is easier to:

- Implement an element of run time type checking.

- Implement a full garbage collector.

- Implement debugging aids.

Every instance of a structure class carries round with it a reference to its description.

The description chosen was a string containing a cannonical form of the class declaration. This is shown in fig 6. We term these strings *class ids*. Notice that in the cannonical form of of the class declaration the order of occurrence of fields does not correspond to that in the original declaration. In the example shown, the class declared as:

structure cons(**pntr** hd, tl; **int** val)

has its fields reordered in the cannonical description as *tl,hd, val*. This is a consequence of the rule that is used for evaluating the equivalence of class declarations. Classes are said to be equivalent if they have the same fieldnames and the fields have the same type, but the order in which the fields occur does not

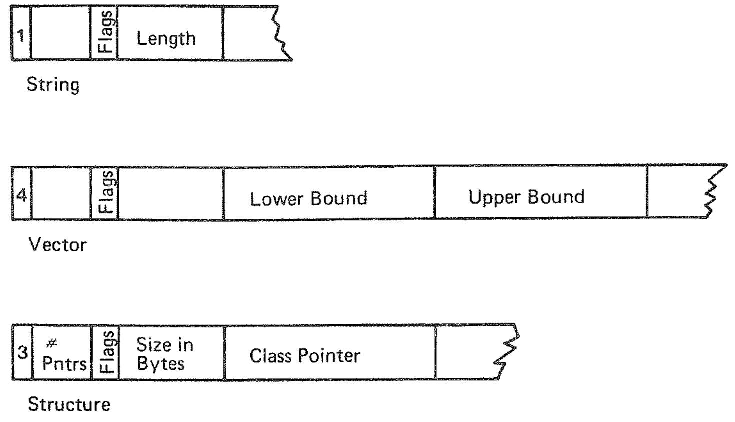

Figure 5: The format of objects on the PS-algol heap

matter. By sorting the fields into a cannonical order checks for class membership, whether by the is operator or the implicit check at dereference can be done by string equality. In order to minimise the cost of this certain optimisations have to be employed. The string equality function operates on the addresses of strings. When given two string addresses it first compares the addresses for equality before comparing the strings. The essential thing is to make sure that the great majority of class comparisons can be reduced to address comparison. The proposition behind this is that the great majority of class comparisons succeed. This can be ensured by appropriate organisation of what is termed the structure table.

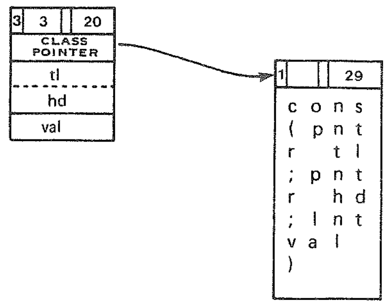

Figure 6: Each structure contains a pointer to a string containing its declaration

The compiler continues to allocate trade marks as before, and these are used as indices into a set of three tables embeded in the program code.

- **Structure table**: see fig 7. This is a sequence of structure headers organised as shown in figs 6 and 5 but with each header followed by a pointer into the permutations table. The class pointer fields of the headers point into the:

- **Class id. table**: this is a sequence of string literals organised as shown in fig 5 containing the cannonical declaraion of the classes.

- **Perms table**: consists of the mapping of fields from the order in which they have been declared to the order in which they are stored. This is necessary because in the syntactic form used to intialise new members of a structure class, the order of occurrence of initialising values must correspond to the order of occurence of the fields in the version of the class declaration currently in scope.

The class pointer as produced by the compiler is an offset from the start of the **class.id.table** and the perms pointer an offset from the start of the **perms.table**. At program startup the run time system converts the class pointers into actual store

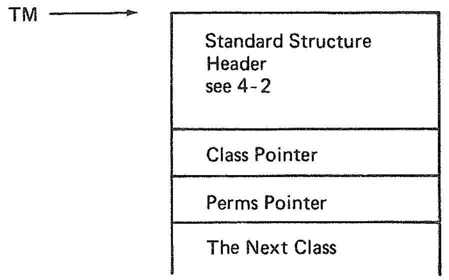

Figure 7: Structure Table Organisation

addresses of the entries in the **class. id. table**.

In order to ensure that class member ship checks can be reduced to address comparison when a structure is imported, the following algorithm is executed:

1. Check to see if a header for an equivalent class exists in the structure table. This is check is optimised by keeping a vector of the addresses of the strings in the **class id. table** sorted alphabetically. A binary search is performed on this using the class declaration in the imported structure.

2. If there is an equivalent class then the class declaration field in the header is overwritten with the address of the equivalent string in the **class id. table**.

3. 4. CLASS FRACTIONS

It was explained earlier that in the POMS we take advantage of the fact that disk store is organised in terms of fixed size blocks to implement a transactionally secure storage system. Another factor that has to be taken account of is that disks are much slower than RAM. If you are using disks, it is wise to try and minimise the number of disk transfers that take place. In RAM there is no corresponding concept. You can follow a linked list equally well whether the nodes are adjacent in memory or widely spaced. On disk, however, it is much faster to follow a list whose elements are next to each other because several nodes can then be brought in with each disk transfer.

Since linked lists are a common class of data structures on heaps it was thought wise to design our placement algorithms to take this into account. We hypothesise that linked lists tend to be made up of objects of the same structure class and have therefore follwed these rules in allocating PIDs to structures.

- items of a given class are gathered together into a set of blocks which

are termed the fractions of the class.

- each logical block contains members of only one class

- when allocating PIDs to cells on a linked list, try whenever possible to allocate adjacent cells to the same logical block i.e., put them into the same fraction of their class.

3. 5. CLASS DESCRIPTORS

For every class in a database there is a class descriptor. This holds information that is common to the class as a whole, including:

- **Freelist:** a list of free PIDs available for the storage of new members of the class.

- **Use list:** a list of the class fractions used to store this class.

The class descriptors are accessed via a table indexed on class identifiers and reachable from the database root. It is intended that the Use list will enable fast access to all fractions of a class and thus allow the programming of bulk operations on all members of a class. For instance it would provide the access paths required for a query language interpreter program which allowed operations of selection and projection over classes.

3. 6. NON STRUCTURE OBJECTS

In addition to structures, POMS supports the persistent storage of strings and vectors. Such objects do not fall into classes but they can still be grouped according to size so that objects of the same size can be clustered into single disk blocks just as with structures. However, in the case of strings and vectors the number of different sizes is likely to be greater than with structures. This could lead to the proliferation of disk blocks containing only one item - the only 43 character string in the database for example. So a compromise has been sought, objects for which the $\log_2$ of their size is the same are grouped together. Thus there are logical blocks containing objects of size 2: 3 words, 4: 7 words, 8: 15 words etc.

4. DATABASES

We are now in a position to summarise what a POMS database contains. A Database is a named collection of PS-algol objects: strings, structures and vectors. It is the unit upon which program transactions are carried out.

At an implementation level a database effects three mappings:

- from Relative Block Numbers (RBN) to Physical Block Numbers. This mapping is analogous to that normally supported by a direct access file system. Its function is to make a set of non-contiguous disk extents look like a single contiguous address space.

- from Logical Block Numbers (LBN) to Relative Block Numbers. As will be explained later, this mapping provides the mechanism for secure transactions.

- from Persistent Identifiers (PID) to LBN+offset. This mapping takes us from a disk block based address space to a high level language object address space.

As far as possible, the data structures necessary to carry out these mappings have been implemented in terms of S-algol data structures. This is intended to make code more maintainable and more portable.

4.1. ROOT BLOCK

In logical block 1 of a database is the **Root Block** ROOT. This is a valid S-algol structure belonging to the class **opened.db** whose declaration is given in fig 8.

```
structure opened.db(bool  write.mode;
                    ***int zLR.map;
                    *int owner.dir;
                    int  db.no;
                    pntr owner.toc;
                    int zone1,zone2,zone3,zone4,zone5,zone6,zone7,zone8,
                       zone9,zone10,zone11,zone12;
                    int LR1,LR2,LR3;
                    pntr persistence.root;
                    ***int BIT.map.tree;
                    int Next.RBN,Max.RBN,Next.LBN,Max.LBN;
                    pntr Class.table1, Class.table2,function.table;
                    *pntr Non.struct;
                    string path.db.name;
                    int old.D.old.B
                    )
```

Figure 8: The class of a Root Block

The **Root Block** contains pointers to all of the administrative data necessary to maintain the database, and a pointer to the root of whatever data particular applications programs have put in the database. This last pointer is the *persistence.root*. The first field of this **Root block** is the LR.map which translates Logical Block Numbers to Relative Block Numbers.

4.1.1. RELATIVE BLOCK NUMBERS

We have already said that the Relative Block Numbers to which the LR map maps are analogous to record numbers within a direct access file. It would obviously be possible to implement these directly in terms of the record numbers of the filing system that the POMS was running on top of. It was decided not do do this in order to make the POMS more operating system independent. It is intended that POMS should be able to run as a stand alone system on single user machines where there may not be a suitable disk operating system beneath it. It therefore has provision to do its i/o directly in terms of physical disk blocks and disk addresses. This makes it necessary for the POMS to carry out the mapping from a contiguous set of relative blocks to a possibly noncontiguous set of phsyical blocks.

Zones 1..12 are the data structure used to carry out the mapping from Relative Block Numbers to Physical Block Numbers. A Zone is a contiguous set of physical blocks. The number of physical blocks in a zone z: 1..12 is given by 2^{3+z}. A Zone field contains the PBN of the first block in the Zone. The mapping from RBN to PBN is shown in fig 9.

The zones are held in 12 integer locations in the root structure. It is necessary that they be part of the **Root Block** to avoid an indefinite recursion when one attempts to get them in.

4.2. BLOCK ALLOCATION

A pair of bitmaps are maintained in each database to keep track of the free relative blocks. *bitmap* 1 is the current bit map, a bit is set in it for each Relative Block present in the database that is not currently assigned to a logical block. *bitmap* 2 is the future bit map, a bit is set in it for each Relative Block that will not be assigned to a logical block at the end of the current transaction. Whenever a new relative block is required it is the first that corresponds to a set bit in bitmap 1 is used. If none are set, another zone is added to the database, and all bits corresponding to it are set in both bitmaps. The new zone will be twice the size of the last one allocated to this database. When the last zone has been added, a warning will be issued on commit that the database has now reached its maximum allocation. If the bitmap was not empty the first relative block found is allocated and the bits corresponding to it in both bitmaps are cleared.

The database root block holds a number of registers assiciated with the allocation of blocks:

- *Max.RBN* This holds the highest relative block number so far allocated in the database. It corresponds to the highest bit set in the Bitmap.

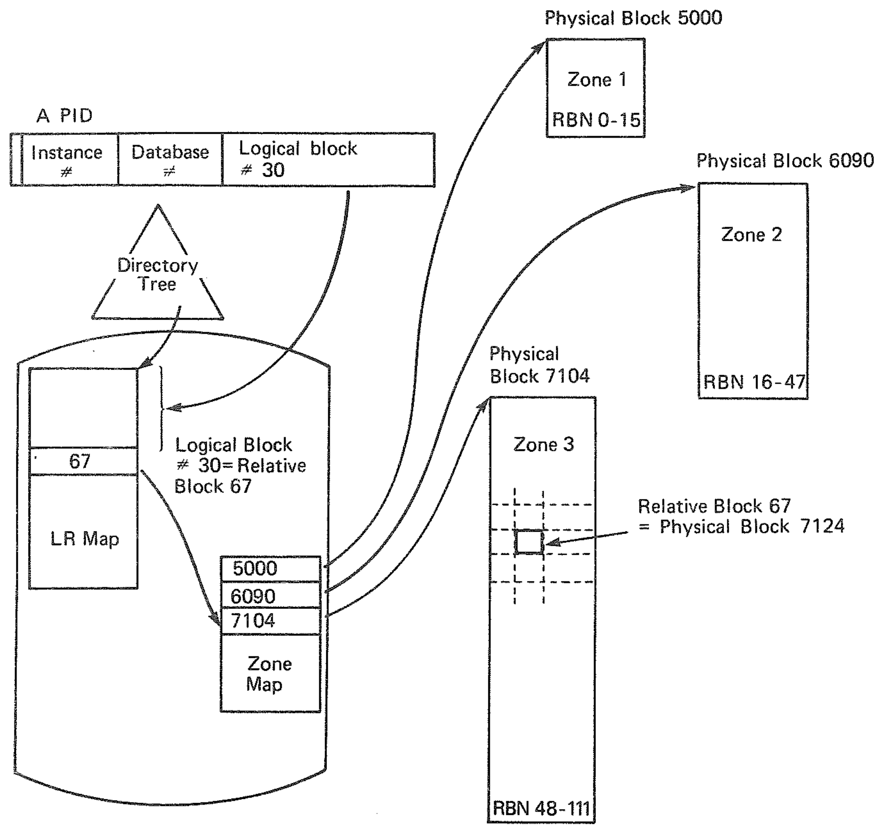

Figure 9: Interpreting a Logical Block Number

- ***Next.RBN*** This is one greater than the highest relative block so far assigned to a logical block. It nust not exceed ***Max.RBN***
- ***Next.LBN*** When a new class fraction is required this provides its LBN and is then incremented.
- ***Max.LBN*** This holds the highest LBN currently supported by the LR map. The LR map is itself stored in the database and is loaded in by the same

  mechanism as any other collection of objects. To prevent a recursive address fault two steps are taken:

* whenever Max.LBN-Next.LBN falls below lr.lev.size + 1 , then a new bottom level block is added to the LR map and Max.LBN is incremented by lr.lev.size.

* the new block is assigned a pid with a PSN falling within an already allocated portion of the LR.map

5. DIRECTORY LEVEL

Up to now this explanation has assumed that a program will only open one database at a time. In practice this is likely to prove rather restrictive. It is often usefull to be able to partition data into compartments which although related to one another have enough autonomy to justify treating them distinctly. It would be usefull if one could devise a mechanism by which objects in one PS-algol database might reference objects in another. This would allow the sharing of common data without the overheads of taking copies into all the databases that needed it.

To do this we must partition or persistent address space. This can be done by adding another field to the PID.

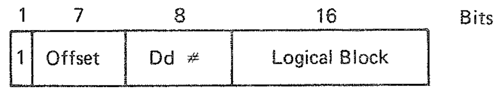

Figure 10: Persistent Identifier Mark III

The extra field the **Database Number** (shown in fig 9) allows the space of logical blocks to be divided into 256 databases.

To make this field meaningfull it is necessary to impose a numbering system on the databases, so that when we encounter a Database Number we know which database this corresponds to. Organising the interpretation of Database Numbers is the task of the POMS Directory system.

At an abstract level a Directory is an updateable funtion from strings to Directories or Databases. At an implementation level a directory serves further purposes.

- It acts as a means of finding directories and databases on disk.
- It acts as a means of distribution of disk resources.
- It is the basic support for atomic transactions.

In order for Directories to act as a route map they must start in a known position. The starting point of the hierarchy will be located at Physical Block Number 1.

5.1. TRANSACTIONS

Directories provide the basic support for atomic transactions. To commit a transaction it is necessary to have some mechanism to cause the newly written set of pages to be treated as the currently valid set. In the POMS this mechanism is provided by the directories which contain pointers to the currently valid instanciations on disk of databases. When a transaction on a database completes, it is written back to a new location and its directory entry is updated to point to the physical disk address of its Root Block.

It is still necessary to be secure against the possibility of a hardware failure occuring during the course of a directory write. It is obviously impermissible to write the directory back to the same site that it came from, for if one did, an error during write could cause the old directory to be lost. The solution adopted is to write it back to alternating sites.

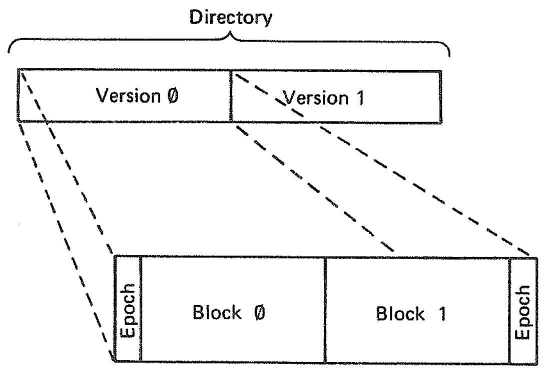

Figure 11: The format of a Directory

The physical implementation of a directory is as two contiguous Versions each of which occupies two contiguous disk blocks as shown in fig 11.

The address of a directory is defined to be the PBN of the lowest numbered block in these two versions. The first and last 16 bit integer fields of a Version are its

Epoch Number. Before a Version is written bach to disk these Epoch Numbers must be incremented. A Version is said to be Valid if both of its Epoch Numbers are the same. In the event of a failure occuring between the output of the first and the second disk blocks, then this should be detected by the inconsistencies between the Epoch Numbers at the start and finish of the Version. This method of consistency check is proposed in^{10}.

Whichever of the two Versions in a database has the higher Epoch Number is said to be the Current Version. If a Version is not Valid its Epoch Number is defined to be negative.

The operation of incrementing a Epoch Number is cyclical such that the number following 10000 is 0. Given two Epoch Numbers a, b then

a<b iff b=a+1 or a+1<b

If one Version has been corrupted it is always possible therefore to recover the previous version.

A Directory Transaction is comprised of the following phases:

1. Read both Versions.
2. Determine which is Current.
3. Make any necessary changes to the in core copy of the Current Version.
4. Write the updated Current Version, with incremented Epoch Number back to the site of the Non Current Version

5.2. STRUCTURE

The internal structure of a version occupies 510 16 bit integers disregarding the Epoch Numbers. These are divided into several fields as shown in fig 12.

5.2.1. PASSWORD

A Password is a string of up to 16 characters padded with blanks. No operations are allowed on a directory without the calling program supplying the correct password. The one exception to this is navigation. The system routines are allowed to navigate through the directory structure without knowing the passwords of intermediate levels.

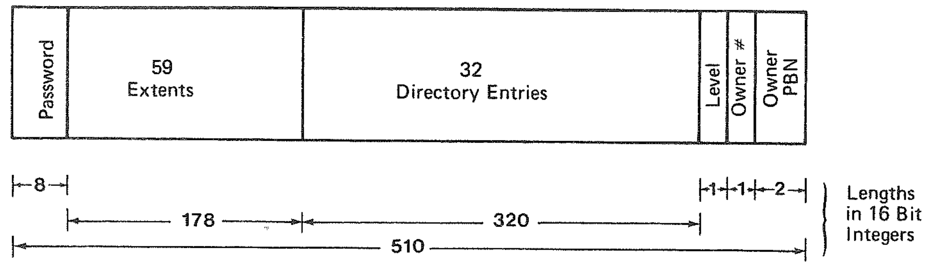

Figure 12: The format of a Version

5.2.2. EXTENT

An Extent is three numbers that describe a contiguous area of disk. The first is the disk number, the second the starting block number on the disk, the third is the number of blocks in the extent.

The storage allocation system used in POMS is hierarchically distributed. Each directory is responsible for providing space for its subordinate databases or directories. If a directory runs out of space it asks its owner to allocate it extra space. Each time a directory demands space from its owner it asks for an amount equal to a fixed proportion of the total ammount aquired so far. Currently this proportion is set to unity, so that the effect is to cause directories to grow by successive doublings of the amount of space allocated to them. The rationale behind this approach is twofold

1. It is hoped to encourage locality of access to data that is likely to be accessed within a given transaction thus reducing disk head movement.

2. By distributing the storage allocation it prevents over-locking of a single global block allocation structure.

5.2.3. ENTRY

A Directory Entry is a 20 byte field composed of the as shown in fig 13.

If the top bit of the lock is set then entry is open for writing. Otherwise the lock

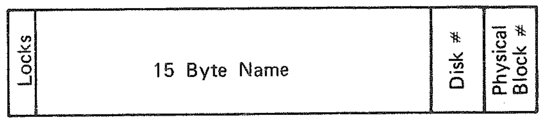

Figure 13: A Directory Entry

contains a count of the readers. If the lock is non-zero the entry can not be opened for writing. The protocol is thus one writer many readers.

An entry may point at either another directory or at a database. If it points at a database this is indicated by a disk number>=0 if it points at a directory the disk number is <0. In the latter case, the absolute value of the disk number must be used for navigation.

The convention adopted in naming directory entries is to specify a path name whose components are separated by "/", so one might have a name such as "poms/text/dictionary" to indicate a dictionary database that was in a directory "text" in a directory "poms".

5.3. DATABASE NUMBER

A Database Number identifies the Database holding an object. The interpretation of this number depends upon the frame of reference in which it is situated. An absolute identification of a database can be achieved, using the path through the directory hierarchy from TOP to the database in question. Given that each level of the hierarchy can branch at most 32 ways, a database can be identified uniquely by a list of 5 bit numbers

$(b[-[1], b_2, \ldots b_j) : j \rightarrow 1..n$

where n = max depth of hierarchy.

The absolute identification of a database requires 5n bits. If we impose a restriction on the set of objects that can be addressed from any one site, then we can in theory economise on the number of bits required. We have argued elsewhere [11] that it is desireable to restrict addressability in a persistent addressing system to the databases in nodes of the tree that form a path from the current node back up to the root of the tree. Thus, although for a tree of depth n and branching factor m there can be m^n databases, the number that can be addressed from within any one database would be m.n. In our example m=32 and n=8. A POMS system can contain 2^{40} databases. The maximum that can be addressed from any given position is 256. If we are in database $(b_1, b_2, \ldots, b_j) : j \le 8$, then this provides us with a frame of reference to interpret a database number D: 0..255:

D refers to database D rem 32 in bi

where i= D div 32

An illustration of the interpretation of a database number is given in 14. Our interpretation of a PID is now given in fig

5.3.1. OWNER

The owner field contains information necessary to locate the directory within the hierarchy. It points at the disk address of the owning directory.

5.4. OPERATIONS

A number of operations on the directories are provided as transactions.

- **Space Allocation**: transactions are provided to claim or release space.
- **Entries**: transactions are provided to delete, lock and unlock entries and to change the disk address associated with a directory entry.
- **Utilities**: various utility functions to search directories, create subdirectories and make enquiries about directories are available.

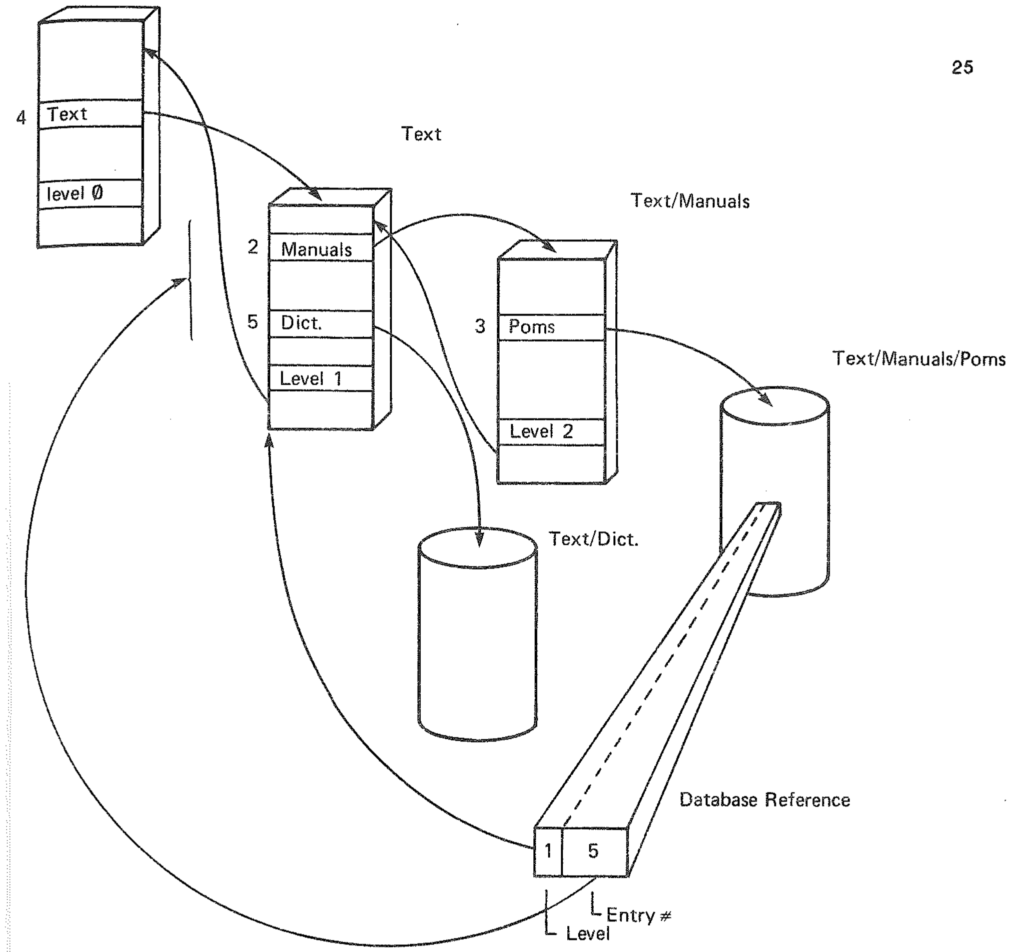

Figure 14: Interpretation of the Database Number field of a PID

1	7	3	5	16	Bits
1	Offset	Lev	Entry	Logical block	

Figure 15: **Persistent Identifier Mark IV**

6. CLOSEDOWN

Currently it is only possible to commit a transaction on a single database: the one opened in write mode. The Commit sequence is effected by the notional algorithm given in fig 16.

```
!


procedure commit
begin
 let the. db=Output. db
 let BIT. map=the. db(BIT. map. tree)
 let dn: =0  ;  let dc: =0 ; BIT. map(1) : =BIT. map(1)
 repeat {
        dc: =dn
        procedure keep(int i)
         if written(i) and marked(i) do {
            dc: =dc+1; clear. flag (i, written. bit )
            mark. class. fraction. of(pid. of. lon(i) ,  the. db)
          }
        app. to. dependents(Output. db, keep)
        }
 while dc>dn do dn: =dc
 purge. each. marked. class. fraction
 let disk. no=the. db(old. D)
 let block. no=PBN(RBN(PCN. of(pid. of. loca(pntr. to. loca(the. db)))))
 let root. loc=@1 of int[abs(disk. no) , block. no]
  add. entry(db. file, the. db(owner. toc) , root. loc, the. db(db. name))
end
!
```

Figure 16: The algorithm for commiting a transaction on a database

The procedures used in the algorithm are outlined below.

6.1. MARK CLASS FRACTION

The procedure mark. class. fraction. of maintains a hashed table of class fractions. Each time it is called, it checks to see if the class fraction of its parameter has been entered in the table and if not enters it into the table and updates the LR map entry for that class fraction to point at a new Relative Block. The bit corresponding to the new Relative Block is cleared from the current bit map, and the bit corresponding to the old Relative Block is set in the future bit map.

6.2. PURGE

The procedure *purge. each. marked. class. fraction* builds a list of class fractions in this hashed table sorted on their Relative Block numbers. It then traverses the list , and executes the following algorithm

1. fetch the old relative block

2. find all PIDLAM entries whose PIDs belong to this logical block.

3. discard those that have not had their Dump bit set

4. copy the objects refered to by the rest into the logical block buffer

5. write out the buffer to disk

The initial sorting by relative block numbers is to minimise disk head movement. This could be thwarted by step i) of the above algorithm. It will therefore be advisable to prefetch several of the old relative blocks at a time, and to fetch these in relative block order.

6.3. BITMAP

At the start of the close down the current bit map is assigned to itself. This apparently senseless assignment is intended to make sure that the top level of the bit map tree is written back to disk. The procedure rotate. bit. maps is shown in 16.

```
!


procedure rotate. bit. maps(pntr db)
begin
 let bitmap=db(BIT. map. tree)
 let temp=bitmap(1)
 bitmap(1) : =bitmap(2)
 bitmap(2) : =bitmap(1)
end
!
```

Figure 17: The bit maps must be swapped at the start of a transaction

7. PORTABILITY

The POMS is written in PS-algol in order to make the porting of PS-algol to new environments simple. An objective of POMS is to make it easy to implement PS-algol on any machine for which S-algol is already available.

7.1. NEW BUILT IN PROCEDURES

In order for an S-algol system to be able to run the POMS several new procedures must be provided by the run time system to communicate with the POMS, or to carry out operations required by POMS that are inefficient if written in standard S-algol.

7.1.1. HEAP INTERFACE

- *HEAP* this returns a vector which is the heap indexed by LONs
- *illegal.lon* if the S-algol system finds that it has been instructed to dereference a PID it calls the procedure *illegal.lon* with the illegal local address (PID) as parameter.
- *pntr.to.lon* this coerces a pointer to a local address held in an integer
- *int.to.pntr* used to set up initial data structure, just coerces an int to a pntr
- *make.space* allocates a number of words on the heap and returns the local address
- *block.move* given a source, a destination and a count copies the requisite number of bytes from the source to the destination

7.1.2. BIT TWIDDLING

Some additional standard procedures to handle boolean operations have been added.

- *not(int->int)* this ones complements an integer
- *test.bit(int word,bitno->bool)* this returns true if bit bitno in word is set
- *clear.bit(int word,bitno-> int)* this returns the word with bit bitno cleared

7.1.3. DISK INTERFACE

POMS is designed to work on either a bare machine on which it has total control of the disk drives or on a machine with a disk operating system interposed between it and the drives. In the second case a direct access file is to be used to mimic each drive.

The basic operations on the disk or erzatz disk are carried out by the procedures:

read.blocks(file,int->*int) ! int parameter is block count

write.blocks(file,*int)

seek.blocks(file,int(block number),**int** (disk.number))

In the case of a system working at the level of the physical disk the filename will be ignored. Read.blocks and write.blocks will transfer to or from the address provided by the previous seek operation.

A Physical Block Number (PBN) is defined as being the identifier used by the procedure seek.blocks to identify which block transfers are to start at. Blocks are assumed to be 512 bytes. Seek.blocks and the transfer routines must convert PBNs into the records or blocks used by the underlying system. Within an S-algol program the blocks are represented as vectors of integers.

8. CONCLUSION

POMS has been implemented on the VAX/VMS system. Work is currently underway transporting it to the ICL Perq, ICL 2900, Vax/Unix and Motorola 68000. In converting an existing S-algol system to PS-algol, additions to the compiler and interpreter do not exceed 10 % of the original source.

9. ACKNOWLEDGEMENTS

This work has been carried out at the Univesities of Edinburgh and St Andrews. The work at Edinburgh was supported in part by SERC grant GRA 86541. It is now supported at Edinburgh by SERC grants GRC 21977 and GRC 21960 and at St Andrews by SERC grant GRC 15907. The work is also supported at both Universites by grants from ICL.

REFERENCES

1. Atkinson M.P. Chisholm K.J. Cockshott W.P. ``PS-algol: and algol with a persistent heap,'' *ACM Sigplan Notices*, July 1982, .

2. A. Van Wijngaarden et al. ``Revised report on Algorithmic Language Algol68,'' *Algol Bulletin*, No. 36, March 1974, .

3. Ron Morrison, ``S-algol reference manual,'' Tech. report cs 79/1, St Andrews University Dept of Computer Science, 1979.

4. Atkinson M.P. Chisholm K.J. Cockshott W.P. Marshall R. ``Algorithms for a persistent heap,'' *Software Practice and Experience*, No. Vol 13, 1983, .

5. Atkinson, M.P. Chisholm, K.J. Cockshott, W.P. ``CMS: a Chunk Management System,'' *Software Practice and Experience*, 1982, .

6. Atkinson, M.P., Chisholm, K.J., Cockshott, W.P., ``The New Edinburgh Persistent Algorithmic Language,'' Tech. report 8, Infotech, December 1981.

7. Peter Wegner, *Programming Languages Information Structures and Machine Organisation*, McGraw-Hill, Computer Science, 1968.

8. Warren Teitelman, *Interlisp Reference Manual*, Xerox, Palo Alto Research Center, 1974.

9. Traiger I.L. ``Virtual Memory Management for Database Systems,'' *ACM Operating Systems Review*, Oct 1982, .

10. Challis, M.P. *Data Consistency and Integrity in a multi-user environment*, Academic Press, Databases : improving usability and responsiveness, 1978.

11. Cockshott W.P. *Orthogonal Persistence*, PhD dissertation, University of Edinburgh, November 1982.

Index