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

A Persistent Software Database with Version Control

R.L. Cooper and M. P. Atkinson

Persistent Programming
Research Report 55
December 1987

A. Deale.

Preface

This paper has been submitted to the ACM International Conference on the Management of Data to be held in Chicago in June 1988.

A Persistent Software Database with Version Control

Cooper R.L. and Atkinson, M.P.

Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ

Abstract.

In constructing large scale applications programs, problems arise of maintaining a complex modular structure and few systems offer good tools for managing the structure. Using persistent programming languages to construct such tools transforms the problem into a database maintenance problem. The application construction environment can be considered to be a database of software libraries and standard database techniques can be used to handle the modules. There needs to be appropriate storage, retrieval, display and version control facilities. and thus software re-use is encouraged and the production of documentation can be automated. The persistent programming language, PS-algol, provides a good basis for such libraries as it has appropriate graphics facilities, first-class procedures and the ability to choose the time at which program modules are bound together. This latter feature means that it is possible to create utilities which permit different ways of binding modules together.

Introduction.

Producing an application program consists of a number of steps -

- systems analysis and design, from which arises a set of modules which will comprise the final program;
- searching the environment within which the program is to be created for versions of any of these modules which already exist - clearly this search may influence the first step;
- modifying any of these which only approximate to the required module;
- producing any new modules; and
- 'glueing' the modules together.

A persistent store with first-class procedures can hold program fragments in the same space as 'data'. Therefore the same operations are available on modules of programs as on any other data item. This should provide an ideal environment within which to build tools which support the activities above, by using the same kind of operations traditionally used on other kinds of data.

The diagram in Figure 1 represents a multi-application environment. In the centre is a 'system' library of modules of general applicability. Surrounding it are application programs which contain modules specific to the application as well as calls to system library modules. Each application has a single root module from which it is started and consists of a Directed Acyclic Graph of modules drawn from its own library and the system library.

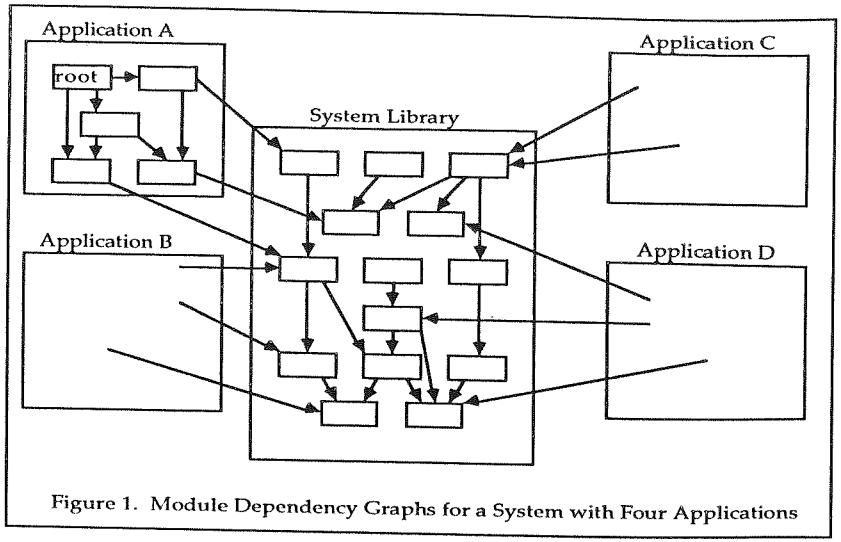

Figure 1. Module Dependency Graphs for a System with Four Applications

Given this view of an application, we may list some facilities which we wish to provide in developing an application:

- a display mechanism for the DAG, with facilities for scrolling it and zooming in to get details of individual modules;
- methods for discovering what modules exist in the system library;
- support for versions of modules;
- the ability to establish links between modules and determine the nature of these links;
- assistance in propagating any changes to low-level modules painlessly through the DAG;
- aid in the production of good quality on-line and off-line program documentation.

In this paper, we show how to provide these facilities, in particular concentrating on version control and the way in which modules are bound to their caller. We wish to support the variety of reasons for which new versions of modules are produced. In particular, we note that a new version may be: the removal of a program error; the extension of the power of the module; or the provision of an alternative to the already existing versions (for instance an implementation of a new sorting algorithm, applicable in different circumstances than those already existing).

We also wish to support a variety of binding styles. We may envisage that this may include once-and-for-all static binding, dynamic binding to get the latest version of the lower-level module or menu-driven selection between parallel versions. In the latter case, the application programmer may make the selection at the time of building the program or pass the decision to the user, who may, for instance, choose an interface style when starting the program. These different styles can be provided in a language like PS-algol, which permits a range of times at which programs are bound to data [MAD87, ABM88].

The rest of the paper describes some of the relevant facilities of PS-algol and then a set of programs which provide some of the facilities above.

PS-algol.

PS-algol is a block-structured, strongly typed language with orthogonal persistence [Psal87]. Every object in the language consists of a quadruple of attributes: its name; its type; its value; and its constancy. Of these, only the value may change and then only if it has not been declared as a constant object. The type system is simple, but rich enough to include types to represent graphical data [MDB86] and to include procedures as first-class objects (the language is data-type complete)[AM85]. Thus numerical, textual and graphical data are all handled in the same way as each other and in the same way as "programs" (i.e. procedures). This will enable us to store our modules in a database and to provide a good quality interface to the tools we will provide. The provision of first-class procedures also means that the

compiler could be made available as a library procedure. Therefore, programs can construct other programs as strings and compile and run them at run-time.

Although the language is strongly typed, with all objects being bound to a type at compile-time, a mechanism has been provided which allows a program to retain flexibility over the nature of such bindings - the extensible union type. To illustrate its value, consider the PS-algol command:

```
structure address( int house; string street, city )
```

which introduces a new class of objects, named *address*, with one integer field and two string fields. The name, *address*, is introduced as a constructor for instances of the class, so:

```
let CS = address( "17", "Lilybank Gdns", "Glasgow" )
```

creates a new object which is in the class, *address*. The type of this object is **pntr** and the data type completeness of PS-algol means that the fields of a structure may be of any type. In particular they may be of type, **pntr**. This provides a powerful modelling tool since complex structures may be represented in PS-algol.

However, a further feature is that all objects created as instances of any of these class constructors are of the same type: **pntr**. This means that programs can be written which manipulate objects without knowing which class they are in. Thus the binding of program to data can be deferred for as long as necessary. In particular, if a procedure is stored in a structure in a database, it can be unpackaged at the beginning of the calling program, which constitutes a static bind, or every time it is used, thus constituting a dynamic bind.

In particular, the mechanism which is used to provide persistence relies on this. Objects in the program can be made to persist between program runs by the simple mechanism of making it reachable from a root object and executing a **commit** command. The root objects of PS-algol are called 'databases' and consist of tables which associate strings and PS-algol objects of type **pntr**. The command:


```
let DB = open.database( "X", "Y", "write" )
```

creates *DB* as a reference to the table of the database whose name is "X" and password is "Y" and allows the program to update it. The command:

```
s.enter( "keyforO", DB, O )
```

then enters complex object, *O*, into the database, keyed by the string "keyforO", while the command:

```
let O = s.lookup( "keyforO", DB )
```

retrieves a reference to the complex object.

A table of this kind is a natural repository for a simple library of procedures. Each procedure is packaged into a PS-algol structure containing just one field, whose type is the appropriate procedure type. This may then be entered into the library with **s.enter** and retrieved for binding to a calling procedure using **s.lookup**.

Modular Program Construction in PS-algol

The provision of first-class procedures in PS-algol encourages the modular design and construction of programs. Essentially, the program is analysed into units of functionality and each is implemented as a PS-algol procedure. The procedure is then stored in the persistent store for later retrieval by calling modules. For instance, a minimum function might be implemented and stored in th e"program"/"Library" database by the following program text:

```
let min = proc( int a,b -> int )
        if a>b then a else b
structure minpack( proc(int,int -> int) minproc )
let library = open.database( "Procedures", "Library", "write" )
s.enter( "min", minpack(min), library )
if commit() = nil     then write "Procedure min stored'n"`
                      else write "Commit fails'n"
```

Then a module which calls *min*, a procedure which provides the minimum of a vector of integers, for instance, can use it as follows:

```
let library = open.database( "Procedures", "Library", "write" )
structure minpack( proc(int,int -> int)
let min = s.lookup( "min", library )( minproc )

let minvec = proc( *int V -> int )
  begin
       let smallest := V( lwb(V) )
       for i = lwb(V) to upb(V) do smallest := min( smallest, V(i) )
       smallest
  end
structure minvecpack( proc( *int -> int ) minvecproc )
s.enter( "minvec", minvecpack(minvec), library )
if commit() = nil     then write "Procedure min stored'n"
                      else write "Commit fails'n"
```

One advantage of this approach is that the *min* procedure, once stored in the database, is accessible for use by any other program that knows of its existence in a database called "Procedures", password "Library". However, the mechanism is deficient in a number of ways:

- no record of the dependency of one module on another is kept;
- there is no support for the creation of new versions;
- there is no support for documentation;
- the code to store and retrieve modules is unnecessarily complex.

We will take these points in order. As shown in Figure 1, there is an underlying graph of module dependencies which this simple mechanism totally obscures. After the two programs above have been run, there is no way of discovering the fact that *minvec* calls *min*. The link between the two exists only within the closure of *minvec* and this is inaccessible. Therefore, we cannot write a program which is given a pointer to the "root" module of an application and manipulates the module dependency graph (MDG), for instance a program to display it.

The provision of version control is even more important and is totally lacking here. Imagine the effect of making a change to *min*, removing the error that exists

in it - yes the error was deliberate! If the source file is edited, re-compiled and re-run, a correct version of *min* will be stored in the database as required. However, no effect will be felt by *minvec*, as this is still bound to the faulty version of *min*. The module containing *minvec* will also have to be re-run (although not re-compiled), in order to bind it to the new version. We can modify the *minvec* module by moving the lookup of *min* inside the body of *minvec* and this will have the effect of dynamically binding *min* into *minvec* every time the latter is run, thus ensuring that it always uses the latest version. We would like, however, to provide more sophisticated control over this binding, as we will outline below, to reflect the variety of reasons for providing new versions. Furthermore, as we do not have access to the MDG, when we mend *min* , we don't know which modules call it, nor which programs need to be re-run to bind these modules to the new version. Finally, we might like to retain a history of versions, as done in the system described by Davison & Zdonik[DZ86].

Perhaps the lack of any encouragement for documentation may be considered less serious, but any attempt to help programmers document as they produce an application must improve productivity for two reasons. Firstly, it is quicker to document each module as it is written, rather than at the "end" of the production of the whole application. Secondly, it is easier to debug later parts of the program, if earlier parts are already documented. The approach above permits, if not encourages, the programmer who requires a facility to hack in a quick procedure together, dump it in the database, make a call from the module which is under consideration and then forget about it. Allied to this point is the lack of any method of helping the programmer find modules as they are needed. It might be guessed that the minimum procedure is called *min*, but what type are its parameters? At another point in the program, a sorting procedure might be needed. What versions are there and how are they accessed?

The final point is that it is hard to justify the amount of code required to package and unpackage procedures used in PS-algol. The language is designed to simplify the code and reduce the programming overheads. The overheads seen in the above programs are a consequence of the provision of features which provide

immense benefits in other areas, but there is no reason why we shouldn't provide some automatic methods to save us some programming effort. The overheads are made worse by the requirement that every name, including the structure and field names of the procedure packaging, must be unique. We therefore seek to provide a less cumbersome syntax for introducing the simple notions: "this module uses version V of module M in the program library" and "store this module in the program library - this is a new module or a new version of M introduced for such-and-such a reason".

To achieve all of these aims, we introduce a systematic intermediate packaging structure with which to represent the nodes of the MDG. This will include pointers to represent dependencies and versions; text fields for documentation and program source; a time field to capture module creation time; other fields for author name; and of course the procedure itself.

Version Control

We note first that version control in a software library contains two components - the storage of a new version and the retrieval of a specified version. We seek to provide the kind of facilities required for carrying out both of these activities, firstly listing the requirements and then showing how o satisfy them.

When a new version of a software module is created this can reflect a number of different intentions on the part of the programmer:

- a bug-fix - the programmer intends a complete replacement of the old version by the new because the old version is faulty;
- an "extension" - the new version is more powerful or of wider applicability than the old, although the old may still be adequate for some users;

- a new parallel version - this is not intended to replace the old version, but to be an alternative (for instance a new editor).

The programmer who wishes to use a module in a library has a separate set of possible requirements. The programmer may want:

- to retain his original version at all costs;
- any changes at all to permeate through to his application;
- to accept only bug-fix updates, but otherwise retain his original version;
- to specify a particular version when writing the code;
- to choose the version at the time the calling module is stored; or
- to pass the choice of version to use onto the user.

We next present an outline of a system which provides all of these alternatives. To start with we will consider a two-dimensional taxonomy of versions in which each module in our system to exist as a number of parallel "versions", each of which may exist in a number of sequential "generations". We then adopt the following insertion strategy, in which one of three conditions can hold:

- a) it is a new module - a new entry in the library is created;
- b) it is a new version - a new version node is created, which is inserted into a list of versions - if it is marked as preferred, the library entry for this node is made to point to this version;
- c) it is a new generation of an already existing version - the new generation is placed at the head of the list of generations of this version - the new generation may be marked as the preferred one.

The simplest PS-algol code to do each of these is given in Figure 2 for the case in which a string editor is to be inserted.

```
let editor = proc( string Xin -> string )
    .....
structure edPack( proc(string -> string) anEditor)
structure modVersion( string Vname, Gnumb;
            pntr thisVers, lastVers, lastGen )
let library = open.database(
            "Procedures", "Library", "write" )
s.enter( "editor", modVersion("first", "1",
            edPack(editor), nil, nil ), library )
if commit() = nil
        then write "Procedure edit stored'n"
        else write "Commit fails'n"
```

a) Inserting the first version of a module

```
let editor = proc( string Xin -> string )
    .....
structure edPack( proc(string -> string) anEditor)
structure modVersion( string Vname, Gnumb;
            pntr thisVers, lastVers, lastGen )
let library = open.database(
            "Procedures", "Library", "write" )
let oldVersion = s.lookup( "editor", library )
s.enter( "editor", modVersion("second", "1",
        edPack(editor), oldVersion, nil ), library )
if commit() = nil
        then write "Procedure edit stored'n"
        else write "Commit fails'n"
```

b) Inserting the second version of a module

```
let editor = proc( string Xin -> string )
    .....
structure edPack( proc(string -> string) anEditor)
structure modVersion( string Vname, Gnumb;
            pntr thisVers, lastVers, lastGen )
let library = open.database(
            "Procedures", "Library", "write" )
let oldGen:= s.lookup( "editor", library )
while oldGen( Vname ) ~= "second" do
            oldGen := oldGen( lastVers )
let newGen = modVersion( "second",
            succ( oldGen( Gnumb) ),
            edPack(editor),
            oldGen ( lastVers ),
            oldGen  )
s.enter( "editor", newGen, library )
if commit() = nil
        then write "Procedure edit stored'n"
        else write "Commit fails'n"
```

c) Inserting a new generation of the second version of a module

Figure 2. Three Different Types Of Module Instance Creation.

Turning to the problem of binding in called procedures, Figure 3 indicates 6 ways to do this, in the case of linking a call to a string editor into a procedure called *caller*.

Holding on to the original version is straightforward in a persistent system. Code fragment a) of Figure 3 shows how to do this. The latest editor will be bound into *caller* and the dependency created can never be broken by anyone other than the application programmer by re-running this program. In particular, the creation of new versions of editors will not affect this binding.

To make changes to the implementation of the editor permeate through, it is necessary to move the *lookup* of the editor into the body of *caller*, as shown in code

fragment b). Now the editor is bound into *caller* each time it is called. This has the effect of always using the latest version.

To make the change be dependent on the last editor change being a bug-fix, slightly more complex code is required as shown in code fragment c). Here, an initial instantiation of the editor is selected, but is altered if a bug-fix to this version has been made - note that this uses an extension to our original *modVersion* structure, a string field called *Reason* to hold the reason for the update.

To specify the version required, a utility *findNamedVers* could be provided, which given the module name and version name could look up the latest generation of that version. Code fragment d) shows how to this utility could be used to specify the version once and for all. The call to *findNamedVers* could be moved inside *caller* to dynamically bind to the latest generation of the named version.

To leave the choice as to which version of the editor to choose until the procedure *caller* is stored, we could envisage a generic utility, *paraChoice*, which would be called instead of *lookup*, for instance by:

```
let myEditor = paraChoice( "editor" )
```

which, if there was only one version, would return the latest generation of it, but if there were more than one version would build a menu of the *Version* names and get the programmer to choose one. This is shown in fragment e).

The same utility could also be used to pass the choice onto the user by moving the call inside the *caller* procedure. This is shown in fragment f), which shows the version which gives a menu every time a module is called. Another version is called which presented the menu only for the first call after the application is started up.

```
structure edPack( proc(string -> string) anEditor)
let myEdPack = s.lookup( "editor" )
let myEditor = myEdPack( thisVers)( anEditor )
let caller = proc( ... )
    begin
        .......
        newString := myEditor( oldString )
        .......
    end
enter( "caller", caller )
```

a) Always use original version.

```
structure edPack( proc(string -> string) anEditor)
let caller = proc( ... )
    begin
        let myEdPack = s.lookup( "editor" )
        let myEditor=myEdPack( thisVers)(anEditor)
        .......
        newString := myEditor( oldString )
        .......
    end
enter( "caller", caller )
```

b) Always use latest version.

```
structure edPack( proc(string -> string) anEditor)
let myEdPack := lookup( "editor" )
let oldName = myEdPack( Vname )
let caller = proc( ... )
    begin
        let newEdPack =
            findNamedVers( "editor", oldName )
        if newEdPack ~= myEdPack and
            newEdPack(Reason) = "bugfix" do
                myEdPack := newEdPack
        let myEditor =
                myEdPack(thisVers )(anEditor )
        .......
        newString := myEditor( oldString )
        .......
    end
enter( "caller", caller )
```

c) Using bug-fix of original version.

```
structure edPack( proc(string -> string) anEditor)
let myEdPack = findNamedVers( "editor", "myver")
let myEditor = myEdPack( thisVers )( anEditor )
let caller = proc( ... )
    begin
        .......
        newString := myEditor( oldString )
        .......
    end
enter( "caller", caller )
```

d) Using version, "myver".

```
structure edPack( proc(string -> string) anEditor)
let myEdPack = paraChoice( "editor" )
let myEditor = myEdPack( thisVers )( anEditor )
let caller = proc( ... )
    begin
        .......
        newString := myEditor( oldString )
        .......
    end
enter( "caller", caller )
```

e) Choose version at commit time.

```
structure edPack( proc(string -> string) anEditor)
let caller = proc( ... )
    begin
        let myEdPack = paraChoice( "editor" )
        let myEditor=myEdPack( thisVers)(anEditor)
        .......
        newString := myEditor( oldString )
        .......
    end
enter( "caller", caller )
```

f) User chooses the version.

Figure 3. Six Different Strategies For Calling a System Module.

These constitute some of the ways a called module may be bound into a caller in PS-algol. The flexibility we are provided with in PS-algol is, however, somewhat

offset by the nature of the code we have to write to generate the different binding styles. It is somewhat complex, but worse than that obscures the programmer's attention. We propose here to make the programmers requirements explicit and allow the program to create module source files which have clear instructions to the system on what binding is required. We will take that source file and translate it into pure PS-algol of the forms shown in Figures 2 and 3 and submit that to the compiler.

In the case of storing a module, the programmer will be allowed the syntax

```
"save"  ( "systemlibrary" | applicationname ) modulename
        versionname ("new" | "newversion" | "newgeneration" )
```

A line of this form will be placed at the end of the module and this will be transformed into the code shown in Figure 2.

For retrieval, a line of the form

```
"retrieve" ( "systemlibrary" | applicationname ) modulename
    ( "fixed" | " latest" | "bugfix" | "preferred" |
        "Ichoose" | "userchoose" ( "firsttime" | "everytime" ) |
        ( "/" versionname ( "fixed" | " latest" | "bugfix" | "preferred" |
            ( "/" generationnumber ) ) ) )
```

at the start of the source file will be transformed into the code shown in Figure 3. For instance, the 6 parts of Figure 3 could be achieved by:

```
retrieve systemlibrary editor fixed
retrieve systemlibrary editor latest
retrieve systemlibrary editor bugfix
retrieve systemlibrary editor/myver fixed
retrieve systemlibrary editor Ichoose
retrieve systemlibrary editor userchoose everytime
```

Having described the version control system we are now in a position to incorporate these ideas into set of programs to give general support to the applications programmer. We continue by giving a semi-formal descriptions of the objects and operations we will be providing and then outline the method of implementation.

A Semi-Formal Specification.

In the following, we present the objects and operations required in a semi-formal style, to clarify the requirements we have of our system. The formalism is influenced by [HJ87].

We start by defining a **module** to consist of the following:

a name by which it can be referenced;
a description of the purpose of the module
a reference to the first version.

We then define a **module version instance** to consist of:

a version name;
a reference to all other versions which exist;
a generation number, the range of which is left unspecified and may vary from implementation to implementation;
a reference to all other generations of this version which exist;
the reason for storing this version;
a set of references to other module instances which this one calls;
a set of references to other module instances called by this one;
a set of argument types for objects imported by the module;
a set of result types for objects exported by the module (we follow PS-algol in only permitting a single result for the purposes of this discussion - extending to the more general case introduces no new problems);
the program source which defines the module;
the procedure which implements the module; and
sundry documentation material.

A **module source** will be a text which includes specifications of which modules it calls and how the binding to these modules should be done; which objects it imports and exports; the body of the procedure which implements it; and a specification for how it is to be stored, whether as a new module, a new version or a new generation.

A **system library** is a set of modules for each of which all modules called by the module are also in the set.

An **application** is a set of modules and data held together such that:

for each module, all modules called by it are either in the application or the system library;
there is one distinguished **root** module which is not called by other modules but by directly invoking the application;
for each of the other modules, each module calling it is in the application.

We can then define a series of operations which we require.

Run an application. Invoke the root module.

Store a module. This will take in a module source and store it appropriately. The operation also requires an application name or "system library" to determine which module set to store it in.

Display an application/the system library. The MDG will be shown to an arbitrary degree of detail. Among the facilities this operation might be expected to provide are: scrolling about the MDG; traversing the versions of the modules zooming in on the imports/exports which flow along the dependencies or on the details of the modules as required.

Retrieve module details. Given a search string, any modules having that string as part of its name, version name or some subset of the documentation fields will be returned for examination. The search will be confined either to the system library or to an application together with the system library.

Produce program documentation. The application MDG will be processed in a systematic way and the documentation fields will be transformed into a documenting text. Note that the display operation above provides an on-line documentation tool similar to hypertext systems.

We conclude with an outline of an implementation of such a system.

System Implementation - the objects.

We need to specify concrete representations of our abstract objects above. Firstly a **module** will be represented by the following structure:

```
structure module(
        string moduleName;
        string description ;
        pntr dynamicCallers;        ! references from modules that call no specific
        pntr versions  )            !    instance
```

Then a structure to contain an instance of a module:

```
structure moduleVersion(
        string versionName;
        pntr lastVersion;           ! versions held in a doubly-linked list
        pntr nextVersion;
        string generationNumber;    ! probably better stored as a string
        pntr lastGeneration;        ! generations also held in a doubly-linked list
        pntr nextGeneration;
        pntr called;                ! to a lists of called modules and ones which call it
        pntr staticCallers;         !    the structure indicates the nature of the bind.
        string imports;             ! e.g. "int a,b"
        string export;              ! the type of the export
        string theSource;
        integer storeTime;          ! the time when the module was stored
        string author;
        string updateReason;        ! e.g. "bugfix"
        pntr packagedProc )         ! points to a packaged procedure.
```

The last pointer will point to a structure which contains a single procedure field to hold the procedure. The names of the structure and the field can be generated automatically from the imports and export of the procedure, for instance *minvec* would be stored in the following structure:


```
structure VintTintprocpack( proc( *int -> int ) VintTintproc )
```

A **module source** will be a string, which in the case of *minvec*, would be like

```
retrieve systemlibrary  min latest
let minvec = proc( *int V -> int )
    begin
        let smallest := V( lwb(V) )
        for i = lwb(V) to upb(V) do smallest := min( smallest, V(i) )
        smallest
    end
save systemlibrary  minvec firstgo new
Richard Cooper
Returns the smallest of a vector of integers.
```

More formally, the module source consists of:

one or more lines retrieving versions of modules from the system library or from the application;

the procedure body implementing the module version;

a line saving the module version in the system library or application;

a line with the author's name; and

one or more lines of description.

The **system library** is a PS-algol database with name "Procedures" and password "Library". New modules will be entered in the top-level table of the database. Updates will be linked via the various fields of the module structures.

An **application** is also a PS-algol database. One entry in the top-level table will point to a table containing the application library, which will be organised in the same way as the system library. The root module is contained in this table along with the others, and is keyed in the table with the name "root". This contrasts with the usual PS-algol approach in which the root module is a compiled PS-algol program which initiates the application. All the data connected with the application will be stored in structures reachable from the other entries in the top-level table.

System Implementation - the operations.

We are now in a position to outline the implementation of the operations in PS-algol.

To **run an application** we will issue the command

runapp *applicationName*

and this runs the following simple program:

```
let AppDB = open.database( applicationName, "friend", "write" )
let library = s.lookup( "library", AppDB )
structure voidprocpack( proc() voidproc )
let root = s.lookup( "root", library )( voidproc )
root()
```

To **store a module,** we will use the following command

store *moduleSource applicationName*

where *applicationName* can be "systemlibrary". This will take in the module source and translate it into a PS-algol program similar to those shown for storing *min* and *minvec*. The program that implements this lies at the heart of our system and deserves the closest attention.

It relies an extension of the PS-algol parser, which analyses the source module in the form presented above. Using a system like that described in [BC87], we can create such an parser, which returns a tree of lexeme, token pairs. We can then use this as follows:

1) Divide the module source into six parts: the set of retrievals; the procedure definition line together with the first **begin**; the rest of the body of the procedure; the storage line; the author name; and the description. Note a **begin/end** pair may need to be inserted if the procedure body consists of just one clause.

2) Scan the retrievals and build **open.database** calls to the system library and application databases as required, as a string *ODB*, say.

3) For each retrieval, build the appropriate code to load the correct module. This code relies on:

   a packaging structure which can be built from the types found in the imports and export fields;

   calls to standard procedures, like *findNamedVers* and *paraChoice* - these will be already in the system library and so must be themselves retrieved;

   an examination of the final parameter to determine whether the retrieval should occur outside or inside the procedure - two strings *staticCalls* and *dynamicCalls* will be built up for these two cases.

4) The require PS-algol source to this point is *ODB* followed by *staticCalls* followed by the procedure definition followed by *dynamicCalls* and the procedure body.

5) The list of modules called by this one is created. The list contains a pointer to the called module if the bind is dynamic or to a module instance if it is static. The list is scanned to create backwards links - a general function, *makeBackwardLinks* can be created which does this systematically.

6) a) If this is a new module, build a module version object with **nil** fields for the version and generation lists and "new" for the *updateReason* field. Get the initial value of the generation number from a system function, *firstGen* - this allows different systems to have different generation numbering mechanisms. Create a new module object, make it point to the version object and put it into the table.

   b) If it is a new version, lookup the module object and create the new version object so that it is linked into the list of versions. Make the module point to this version if it is "preferred".

c) If it is a new generation, lookup the module and then find the version. Create a new module instance to represent the new generation and then link it into the list of generations - the generation number will be determined by using another system function, *succGen*, to generate it.

7) Fill in the other fields of the module version from the appropriate sources.

8) Put the **commit** line in and we're done. We have a complete PS-algol program to place the module appropriately. We can call the PS-algol compiler and then run the program.

As a final illustration, the module source given above for minvec would be translated into:

```
let sysDB = open.database( "Procedure", "Library", "write" )
structure intintTintprocpack( proc( int,int -> int ) intintTintproc )
let minvec = proc( *int V -> int )
    begin
        let min = s.lookup( "min", sysDB )(versions )( intintTintproc )
        let smallest := V( lwb(V) )
        for i = lwb(V) to upb(V) do smallest := min( smallest, V(i) )
        smallest
    end

structure callList( pntr call; string bindType ; pntr callNext )
let calledProcs = callList( s.lookup( "min", sysDB ), "latest", nil )
makeBackwardLinks( calledProcs )

structure VintTintprocpack( proc( *int -> int ) VintTintproc )
structure module( ....
structure moduleVersion(....
let version = moduleVersion( "firstgo", nil, nil, 1, nil, nil, calledProcs, nil, "*int V", "int",
                             "retrieve .....integers.",   ! i.e. the whole source
                             date(),                      ! system function
                             "Richard Cooper", "new",
                             VintTintprocpack( minvec ) )
let Module = module ( "minvec", " Returns the smallest of a vector of integers.", version )
s.enter( "minvec", sysDB, Module )
if commit() = nil     then write "New module minvec stored ok'n"
                      else write " Commit of new module minvec failed'n"
```


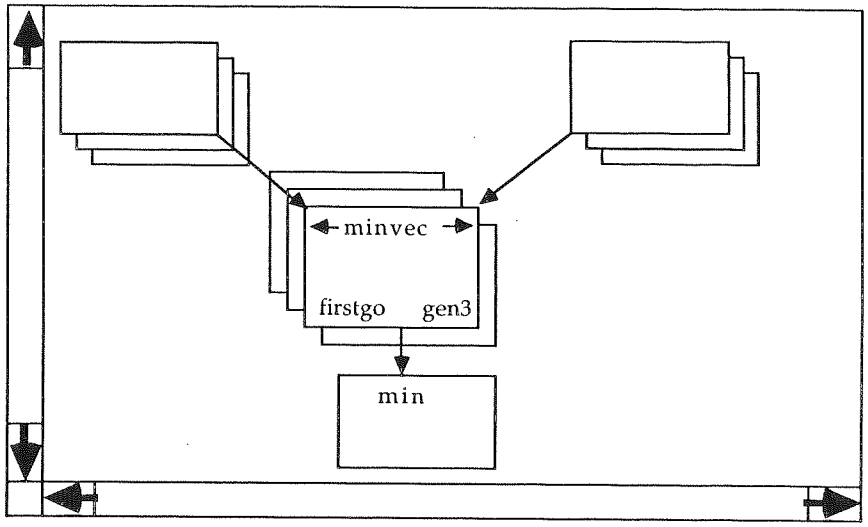

Fig. 4 Module Dependency Graph Display

To **display** an MDG, the command

dispapp *applicationName*

will call a program which shows the MDG from the root module in a scrollable window, putting as many nodes on the screen as it can. The modules are displayed as boxes with their name, version name and generation number, as shown in Figure 4. The existence of other generations is indicated by shadow boxes, to a maximum of five, while the existence of alternative versions is shown by arrows at the top of the box. Alternative versions or generations can thus be seen by clicking on the shadows or arrows. As specified above, clicking on the nodes will display the documentation material available, clicking over the shadows and version arrows will move to display the other versions, while clicking near the arcs will display import/export information. Figure 5 shows the effect of clicking on the *minvec* box and the *min/minvec* arc.

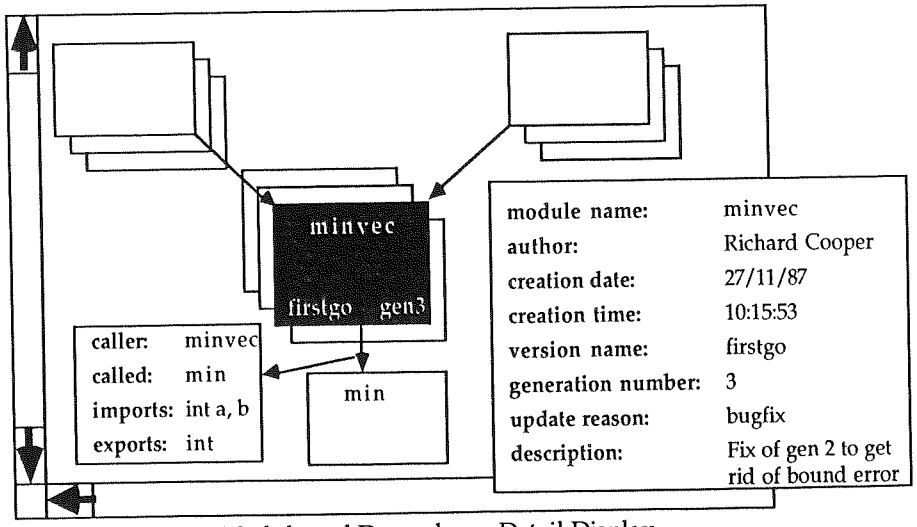

Fig. 5 Module and Dependency Detail Display

An outline of such a program is:

open the database and find the root module;

display at, say, the left-hand side of the screen;

find all the modules called from the root recursively to a fixed depth;

use a node placement algorithm to decide where to show these modules;

then display the nodes and arcs;

wait for the user to click in the scroll-bar or on one of the boxes or arcs;

if the selection is a scroll, recalculate and re-display the nodes;

if the selection is a node or arc, open a new window, with a go-away box.

Module retrieval is done by the command

retrieve *astring applicationName*

where applicationName may be "systemlibrary". The program called by this command scans the appropriate table and finds every module with the string in its name, then all those with the string in its version name and then with its string in the description field and finally in the author field. It then presents a menu of module name/version name pairs, clicking over each of which will display all the documentation information about the procedure.

The **production of documentation** is done by the command

document *applicationName*

this will traverse the MDG and write out the documentation for each module in a systematic way. The program provides a menu of all the modules in the MDG, with the user determining the order of description by clicking over the menu items one by one. There is also an item in the menu which allows the user to supply intervening text, for instance headings, directly from the keyboard.

Conclusions.

We have shown how to build a set of tools to manipulate a procedure database which assists in the construction of application programs. We have been able to do this because we have a persistent language in which the fine detail of storage has been removed from the programmer's concern. The language also provides good graphics facilities which encourages fast prototyping of interfaces. The provision of first-class procedures and the extensible union type **pntr** have been our major aids, however. We have been able to write programs which manipulate the procedures which instantiate our modules with the same ease as any other data item and we have had access to the compiler at run-time. Further, by making references to the stored procedures via **pntr** fields, we have been able to write generic programs which handle any kind of procedure, although any use of any of these will be fully type-checked.

We are not here concerned with the nature of the tools which we have constructed - it is the business of software engineers to design the kind of tools which they require. One tool that we haven't described would be a program which takes the program graph and recreates all of the module dependencies, under user control. What we have shown is that when the tools have been designed, they can be implemented in a language like PS-algol.

We have in fact made a number of restrictive assumptions in the system as described. For a start we have assumed acyclicity in our dependency graphs, as do [HJ87]. The extension to cyclic graphs introduces no new problems. We have also assumed that each module source produces exactly one procedure. There is no intrinsic reason why a module shouldn't store more than one procedure. Nor is there any reason not to use the same mechanism to store other kinds of object. The data-type completeness of PS-algol means that any type of object can be manipulated with the same ease as any other. Thus we envisage future work implementing systems like those described in [EEE87] and [KC87]. We have already referred to the restriction of procedure exports to one - this was merely to ease the transformation into PS-algol. All of these restrictions were made to simplify the work, by excluding extraneous issues, and not because they pose any new hard problems.

Work is already under way to implement this system. The implementation is taking two paths - a direct implementation in PS-algol and an indirect implementation using the requirements modelling language, PSRML [CA87]. A set of tools which ease the programmer's lot should soon be in place.

Acknowledgements.

The work reported here was funded partly by ICL URC grant and partly by Alvey/SERC grant GR/D43259. Many of the ideas in this paper arose those discussions with David Kerr. We would also like to thank our colleagues in the PISA group at Glasgow and St. Andrews and Kieran Clenaghan for many useful comments.

Bibliography.

[AM85] Atkinson MP and Morrison R, "Procedures as Persistent Data Objects", *ACM TOPLAS*, 7, 4, 539-559, October 1985.

[ABM88] Atkinson MP, Buneman OP and Morrison R, "Binding and Type Checking in Database Programming Languages", to appear in *Computer Journal*, 1988.

[BC87] Blott SM and Campin J, "Lgen, Pgen and Sgen: Language Development Tools", *Persistent Programming Research Report*, 49, Universities of Glasgow and St. Andrews, 1987.

[CA87] Cooper, RL and Atkinson, MP, "Requirements Modelling in a Persistent Object Store", in *proc 2nd International Workshop on Persistent Object Systems, Appin*, August 1987.

[DZ86] Davison, JW and Zdonik, SB, "A Visual Interface for a Database with Version Management", *ACM TOOIS*, 4, 3, 226-256, July 1986.

[EEE87] Ecklund D, Ecklund E, Eifrig R and Tonge F, "DVSS: A Distributed Version Storage Server for CAD", in *proc 13th International Conference on Very Large Databases, Brighton, England*, 443-454, September 1987.

[HJ87]] Hull R and Jacobs D, "Towards a Formalism for Module Interconnection and System Evolution", in *proc International Workshop on Database Programming Languages, Roscoff, France*, 313-336, September 1987.

[KC87] Katz RH and Chang E, "Managing Change in a Computer-aided Design Database", in *proc 13th International Conference on Very Large Databases, Brighton, England*, 339-346, September 1987.

[MDB86] Morrison R, Dearle A, Brown AL and Atkinson MP, "An Integrated Graphics Programming Environment", *Computer Graphics Forum*, 5, 2, 147-157, June 1986.

[MAD87] Morrison R, Atkinson MP and Dearle A, "Flexible Incremental Bindings in a Persistent Object Store", *Persistent Programming Research Report*, 38, Universities of Glasgow and St Andrews, 1987.

[Psal87] "The Ps-algol Reference Manual - Fourth Edition", *Persistent Programming Research Report*, 12, Universities of Glasgow and St Andrews, 1987.

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. & Gray, 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.

Atkinson, M.P, Morrison, R. & Dearle, A.
"A strongly typed persistent object store", 1986 International Workshop on Object-Oriented Database Systems, Pacific Grove, California (September 1986).

Atkinson, M.P., Morrison, R. & Pratten G.D.
"PISA : A persistent information space architecture", ICL Technical Journal 5, 3 (May 1987),477-491.

Atkinson, M.P. & Morrison, R.
"Polymorphic Names, Types, Constancy and Magic in a Type Secure Persistent Object Store". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Cooper, R. & Atkinson, M.P.
"Requirements Modelling in a Persistent Object Store". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Wai, F.
"Distribution and Persistence". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Philbrow, P.
"Associative Storage and Retrieval: Some Language Design Issues". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Guy, M.R.
"Persistent Store - Successor to Virtual Store". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Dearle, A.
"Constructing Compilers in a Persistent Environment". Presented at the 2nd Internaional Workshop on Persistent Object Stores, Appin, August 1987.

Carrick, R. & Munro, D.
"Execution Strategies in Persistent Systems". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Brown, A.L.
"A Distributed Stable Store". Presented at the 2nd International Workshop on Persistent object Stores, Appin, August 1987.

Cooper, R.L., Atkinson, M.P., Dearle, A. & Abderrahmane, D.
"Constructing Database Systems in a Persistent Environment". Proceedings of the Thirteenth Internaional Conference on Very Large Databases, Brighton, September 1987.

Atkinson, M.P. & Morrison, M.
"Polymorphic Names and Iterations", presented at the Workshop on Database Programming Languages, Roscoff, September 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 17th September 1987. 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., Bailey, P., Chisholm, K.J., Cockshott, W.P. and Morrison, R.	£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-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. 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 - 2nd 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-31-87	An Introduction to PS-algol Programming (third edition) - 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
PPRR-44-87	Persistent Object Systems: their design, implementation and use. (proceedings of the Appin workshop August 1987) - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£20.00
PPRR-45-87	Delayed Binding and Type Checking in Database Programming Languages - Atkinson, M.P., Buneman, O.P. & Morrison, R.	£1.00
PPRR-46-87	Transactions and Concurrency - Krablin, G.L.	£1.00
PPRR-47-87	Persistent Information Space Architecture - PISA Club Rules - Atkinson, M.P., Lucking, J.R., Morrison, R. and Pratten, G.D.	£1.00
PPRR-48-87	An Event-Driven Software Architecture - Cutts, Q. and Kirby, G.	£1.00
PPRR-49-87	An Implementation of Multiple Inheritance in a Persistent Environment - Benson, P.J., D'Souza, E.B., Rennie, I.S., Waddell, S.J.	£1.00
PPRR-50-87	A Distributed Stable Store - Brown, A.L.	£1.00
PPRR-51-87	Constructing Compilers in a Persistent Environment - Dearle, A.	£1.00
PPRR-52-87	Lgen, Pgen and Sgen - Language Development Tools for a Persistent Programming Environment - Blott, S.M. and Campin, J.	£1.00
PPRR-53-87	Polymorphic Names and Iterations - Atkinson, M.P. and Morrison, R	£1.00
PPRR-54-87	A Requirements Modelling Tool Built in PS-algol - Cooper, R.L. and Atkinson, M.P.	£1.00

PPRR-55-87	A Persistent Software Database with Version Control - Cooper, R.L. and Atkinson, M.P.	£1.00
PPRR-56-87	User Interface Tools in PS-algol - Cooper, R.L., McFarlane, D.K. and Ahmed, S.	£1.00