This paper should be referenced as:
Morrison, R., Dearle, A., Bailey, P.J., Brown, A.L. & Atkinson, M.P. “The Persistent Store as an Enabling Technology for Integrated Project Support Environments”. In Proc. 8th IEEE International Conference on Software Engineering, London (1985) pp 166-172.

The Persistent Store as an Enabling Technology for Integrated Project Support Environments

Ronald Morrison, Alan Dearle, Peter J. Bailey,
Alfred L. Brown & Malcolm P. Atkinson*

Department of Computational Science,
University of St Andrews,
North Haugh,
St Andrews KY16 9SX,
Scotland

*Department of Computing Science,
University of Glasgow,
Lilybank Gdns,
Glasgow G12 8QQ,
Scotland

Abstract

The software engineering community has recognised the need for integrated project support environments (IPSEs) for some time. With such a system the user is provided with an integrated set of software tools with which to operate. Given this set of integrated software tools rather than a set of ad hoc tools the cost of software and project support throughout its life cycle is reduced.

The technique of integration as a method of cost saving, applies to all levels in the hierarchy of problem solving, both hardware and software. This paper discusses one such level, that in which the IPSE is implemented and in particular the use of a persistent store as an enabling technology for IPSEs.

The facilities of the language PS-algol necessary to support an IPSE are illustrated by example and it is demonstrated how an IPSE's base may be provided by a persistent store that supports first class procedures as data objects. The need for a type secure object system which allows static and dynamic binding is demonstrated and finally the secure transactional base of PS-algol is shown to be a necessary and sufficient condition to provide secure version control and concurrent access to both programs and data.

Keywords IPSE Persistence PS-algol transactions

Introduction

The savings gained in building a project on top of an integrated project support environment are also available to the implementor of an IPSE who builds the IPSE out of an integrated system. At present many experiments in building IPSEs are based on ad hoc collections of software tools. This is not cost effective. However the difficulty is in identifying an appropriate technology in which to base the IPSE.

In this paper we describe some of the more novel features of the programming language PS-algol [1] which we feel are of use in constructing IPSEs. These features include a type secure persistent object store in which the user may conduct transactions, procedures which are higher order as first class data objects and a type mechanism that allows both static and dynamic binding.

We demonstrate how these features support transactions and concurrent access to the information base, separate compilation, delayed binding and version control. By use of examples we show how each of these features is appropriate in building an IPSE.

PS-algol

PS-algol was developed from the programming language S-algol [9] as an experiment in integrating a long term or persistent store with a programming language. The base types are integer, real, boolean, string, pixel, image and picture. These are augmented by the recursive application of the following three rules. Firstly given any data type T, *T is the data type of a

vector whose elements are of type T. Secondly the data type pointer comprises a structure with any number of fields, and any data type in each field. Finally given a series of data types T1,..,Tn and a data type T, proc(T1, .., Tn → T) is the type of a procedure of n parameters with types T1, .., Tn that returns a result of type T.

The range of PS-algol data types includes several unusual features. Firstly string is a simple data type [10] resulting in a very powerful string handling capability. Secondly the data type pointer may point to any structure class but, when a field of a structure is referenced a check is made that the structure present is of the appropriate class. Using this polymorphism over the data type pointer, arbitrarily complex data types can be manipulated without reference to their actual structure and this is the basis of the persistent store implementation [4].

The third unusual feature of PS-algol is that procedures are full first class data objects. That is, procedures are allowed the same civil rights as any other data object in the language such as being assignable, the result of expressions or other procedures, elements of structures or vectors etc.

The supporters of abstract data types [7] argue that it is essential for powerful languages to have an abstraction mechanism over data objects. However, as has been pointed out by Horning [5], the advantages and aims of procedural and data abstraction are similar. Indeed if procedures are data objects the mechanism for both abstractions can be the same that is the procedure. This, of course, is not a new idea and was present in the work of Strachey [11] and Zilles [12]. The higher order procedures of PS-algol can be this mechanism, and their use as such is discussed in [2].

PS-algol Persistence

The persistence of a data object is the length of time that the object exists. In traditional programming languages data cannot last longer than the activation of the program without the explicit use of some storage agency such as a file system or a database management system. In persistent programming, data can outlive the program. The method of accessing the data is uniform whether it be long or short term data. This concept has been discussed fully elsewhere [3]. Persistence must be provided as an orthogonal property of data; all data objects, whatever their type, have the same rights to long and short term persistence.

PS-algol's persistent store consists entirely of legal PS-algol data objects. The store is partitioned into databases to allow concurrency control and protection when sharing persistent data. Each database has a root data structure which, via pointers, allows access to the other data objects in the database. Therefore the interface to the persistent store need only provide a method of accessing the root data structure of a database. Since the pointer data type may point to any structure class and any data type can be a field of a structure there is no restriction on the data types that can be held in a database.

The Persistent Store Interface

The interface to the PS-algol persistent store is implemented by two procedures. These are:

let open.database = **proc**(**string** database.name,password,mode → **pntr**)

This procedure attempts to open the database with the name 'database.name' in the mode ("read" or "write") given by 'mode'. Passwords are associated with each database to provide some security when sharing databases. The result of this procedure is a pointer to the root data structure of the database or if unsuccessful a pointer to an error.record. An error.record is a data structure containing information relating to why the open failed.

This is sufficient to provide access to any object in the persistent store. Automatic transfer of data from the long term persistent store is performed by the persistent object management system [4] when the data is accessed. The access of the data in the persistent store is performed in exactly the same manner as in the main store, the object manager knowing the difference so as to leave the transfer transparent to the user.

It is often desirable to ensure that updates to persistent data occur in total or not at all. For example in a banking system the transfer of funds between two accounts would need to be such an update. A mechanism that implements atomic transactions is therefore provided by the following procedure.

let commit = **proc**(→ **pntr**)

When the first database is opened a transaction is started. Ordinarily data objects are copied from the persistent store when they are first used and changes to them are made locally. If any of these data objects have been changed a commit will copy them back to their databases. Any newly created objects reachable from these changed objects will also be copied into the persistent store. They have space allocated for them in the database of the object pointing at them. If data objects from databases that were not opened in write mode have been changed a commit will fail. This ensures that the persistent store is always in a consistent state.

If for any reason a commit should fail then its effects will be removed before any other use is made of the databases it was updating. In this way PS-algol provides a secure transaction mechanism on its persistent store.

If the commit fails the pointer which is returned is a pointer to an error.record and nil otherwise. The error record contains information about why the commit failed so that the program can do something sensible which may be to try to commit again or give the user an error message. The Persistent Store Mechanism

We will now present an example of how the persistent store may be used, by giving a program to paginate text stored in a database. Let us assume for the moment that text has already been added to some database. We shall see shortly that it is easy to construct libraries of long term data using a persistent system and write programs to maintain them. In this example we assume that the database root is a pointer to a data structure for associative store and lookup, supported by PS-algol, called a table. By convention a successful 'open.database' yields one of these tables. Entries are placed in the table using the procedure 's.enter' which takes the associative key, the table, and the value to be stored. The procedure 's.lookup' retrieves from the given table using the given key. Let us now look at the example,

```
structure text.file( *string the.chars )
! this structure is used to store text; recall that *string denotes a vector of strings

let more = proc( cint no.lines ; cpntr the.text )
if the.text is text.file then
begin
        let Line = the.text( the.chars )
        ! extract vector from structure
        let still.text := true
        let count := lwb( Line )
        while still.text do
        begin
                ! write out lines no.lines at a time
                repeat
                        if count > upb( Line ) then still.text := false else
                        begin
                                write Line( count )
                                count := count + 1
                        end
                while count rem no.lines ~= 0 and still.text


                ! if still.text wait for user to type a return
                if still.text do      prompt user and wait for reply
                begin
                        write "---MORE---"
                        let wait.for = read.a.line()
                end
        end
end else write "specified item is not text'n"

! ***** Main Program *****

! find out where text to be paginated is
write "database name : " ; let which.db = read.a.line()
write "password : " ; let pass = read.a.line()
let db = open.database( which.db,pass,"read" )
! if db points at an error.record the open failed
if db is error.record then
        write "Cannot open ",which.db," because ", db( error.fault ),"'n"
else
begin
        write "lookup name : " ; let index = read.a.line()
        let the.text = s.lookup( index,db ) ! lookup index in table
        more( 23,the.text )  ! paginate text
end
```

Figure 1: A program to paginate text obtained from a database

This program paginates text contained in a database into pages 23 lines long. The main program interrogates the user to find out where the text to be paginated is held. The user must specify the database name and its password. The 'open.database' call opens the specified database and provided no error has occurred returns a pointer to a table. The user is then asked the key of the text that is to be paginated. The key is then looked up in the table using 's.lookup' and the result passed to a procedure called more which paginates the text. The procedure also takes a parameter specifying how many lines are to be in a page.

The procedure 'more' actually does the pagination. It first checks that it has been passed a pointer pointing at the expected structure class (a text.file). Then the vector of strings

containing the text is extracted from the structure and the text is written out 'no.lines' at a time.
Partial Application

Having functions as first class functions gives the added bonus of being able to partially apply functions. For example in the previous example we could partially apply the pagination procedure to yield a less general function as follows,

```
let more = proc( cint no.lines ; cpntr the.text )
! defined as above

let paginate.by.n = proc( cint n → proc( cpntr ) )
begin
        proc( cpntr the.text ) ; more( n,the.text )
end

let paginate.by.10 = paginate.by.n( 10 )

! this procedure when called will paginate the given text in 10 line intervals
! note that paginate.by.10 has type cproc( cpntr )
```

Figure 2: Partial application of text pagination procedure

The partial application of the more procedure gives us a paginate by n procedure which is a generator function for procedures which will paginate text into pages of any length. This language feature proves to be very useful when building software tools since it is possible to write general code which may be partially applied and used in a great variety of places. We will make use of this ability in a later example. We already have the basis of a good tool in the pagination program which will be of great utility.

Separate Compilation

With any large and complex system it is desirable and often necessary to construct the system from separately compiled pieces of code. It is therefore necessary to have a mechanism for compiling separate pieces of code and linking them together. Often these units of separate compilation are called modules.

Many languages have introduced the module concept as part of the programming language. Languages which have done this include Ada [6] and ML [8]. It has been shown in another paper [2] that the use of procedures as first class functions gives the same power as modules with no loss of freedom of expression. Figure 3 shows how the pagination program shown above may be entered into the persistent store.

```
structure more.pak( proc( cint,cpntr ) more.proc )
! this structure is used to hold the more procedure

let more = proc( cint no.lines ; cpntr the.text )
! procedure body defined as above

let more.prog = more.pak( more )
! put procedure into a structure

let utils = open.database( "utilities","friend","write" )
if utils is error.record
        then write "Error opening database : ",utils( error.fault ),"'n"
        else s.enter( "More",utils,more.prog )

! put the more procedure in table
let done = commit()
! commit procedure to persistent store
if done is error.record do write "Sorry - commit failed because ",done( error.fault
),"'n"
```

Figure 3 : A program to add the more utility to a database

We can now see that it is a simple matter to enter any data into the persistent store including procedures. To use this procedure it is a simple matter to write a small driving program which

uses it, the following complete PS-algol program allows the user to browse through text which is contained in the database.

```
structure more.pak( proc( cint,cpntr ) more.proc )
let utils = open.database( "utilities","friend","read" )
if utils is error.record
        then write      "Error opening database : ", utils( error.fault ),'"n"
else
begin
        let more.prog = s.lookup( "More",utils )
        if more.prog is more.pak then
        begin
                write "database name : " ; let which.db = read.a.line()
                write "password : " ; let pass = read.a.line()
                let the.db = open.database( which.db,pass,"read" )
                if the.db is error.record
                        then write      "Cannot open ",which.db," because ", the.db(
error.fault )
                        else
                        begin
                                write "lookup name : " ; let lookup = read.a.line()
                                let the.text = s.lookup( lookup,the.db )
                                more.prog( more.proc )( 23,the.text )
                        end
        end
        else write "Error retrieving program from database'n"
end
```

Figure 4: A complete PS-algol program which uses the more utility

Notice that we have satisfied one of the aims of having modules, that is separate compilation, without the need to introduce new language concepts. Another goal of modules is achieved here, that is the desire to have a unit of system construction. We have constructed a basic but useful program which may be used as part of a large system. Although not illustrated in this example we shall see shortly that the third goal of modules that of data encapsulation is also satisfied by having a programming system which treats persistence orthogonally. New Version Installation

With all large systems, constructed out of separate modules, there is a problem of managing the installation of new versions. It is necessary to modify the implementation of modules and then arrange for their subsequent use. Often this can only be done when no part of the system is running, then the new modules are installed by a complete system rebuild this may take considerable resources. Sometimes this is unacceptable as some systems are required to run twenty four hours a day. The alternative of replacing a module in situ has to be carefully managed, as it certainly could not be done safely when the module is in use if that execution were affected.

In PS-algol the transaction mechanism makes the concurrent revision and installation of modules safe. The effects of a transaction are not visible to other transactions until the transaction has committed. Programs starting after it will use the new one for the whole program execution if they are written in the style shown in Figure 4.

More sophisticated mechanisms can be implemented with these facilities. For example, a program may arrange to bind a particular version of one module to the package it constructs, by leaving it directly referenced, or leave it to be picked up when the package is run collecting the latest version. Software tools could be written, to build up systems where groups of modules could be installed, retained, replaced etc. using no more language concepts than the features illustrated here. In the following example we see how we may write a small section of code which installs a module in the database. Notice that the database locking mechanism ensures that the module is not installed until the old version is not being accessed.

```
structure module.pak( proc() the.proc )
! structure to hold a program

let install.module = proc( proc() module ; string module.name,db,db.pass )
begin
        let where := nil
        ! keep trying to open database in write mode
        repeat
                where := open.database( db,db.pass,"write" )
        while  where is error.record
        ! while it isnt opened wait

        ! here the program has got database to itself in write mode
        s.enter( module.name,where,module.pak( module ) ) ! install program
        let done = commit() ! commit transaction
        if done is error.record do write "error entering module'n"
end
```

Figure 5: A procedure to install a new module

Binding

In order to build a safe system it is important to have a secure type system. Many of today's programming languages have safe type systems in which the type checking is done statically, one such language is Pascal. Static type checking makes it impossible to write some programs in which the type of the objects being dealt with are not known until run time, an example of such a program is a loader.

It is possible to delay some of the binding until run time, this is known as dynamic binding. The PS-algol language uses a mixture of dynamic and static binding. Type checking is done as early as possible, usually this means at compile time, we call this eager type checking.

Dynamic binding does have costs. Programs which depend on it may contain errors which could have been detected in a static binding system. This is obviously not acceptable in some situations such as missile control systems. Programs may also run slower because of the checks required at run time. Many programs require to have run time checking though, even in languages with static binding. For example, array indices need checked, the lack of this kind of checking is a common defect in Pascal and C implementations.

Despite the small disadvantages of dynamic binding it does have the advantage that it gives data independence. With static systems the data is bound to the program at compile time. Therefore any changes made to the data require the programs using that data to be recompiled. This is not true of systems with dynamic binding and we believe that this advantage outweighs the disadvantages. Constructing an IPSE in a Persistent Environment

We have already seen how, using first class functions, separate compilation of programs may be achieved in a persistent environment. We have also seen how data may be stored in and retrieved from the persistent store. We will now investigate how an IPSE may be constructed using these features.

Suppose we wish to build a general purpose help utility which may be used either as a stand alone program or as a sub-system of a larger programming or general purpose system. How would we do this? We will use the dynamic binding of PS-algol to separate the data from the program. All we will assume is that text will be held in 'text.file' structures. Notice that, if we weren't trying to write a small example we could write another module which translated text held in any format to the format we require for this program. We will also wish to put the program into a database so that it may be used by other programs. This allows us to build a general program which may be used by many other programs.

The program shown in Figure 7 contains two main procedures help and helper. We would normally put each of these into the database separately allowing the more general help procedure to be used by other utilities and the helper procedure to be used for a stand alone general help system. Procedure help is now given.

```
let help = proc( cpntr help.on )
if helping do
begin
        repeat
        begin
                write "Help : " ! help system user prompt
                let response = read.a.line()
                if response = "quit" then helping := false
                else
                begin
                        ! first lookup topic in table
                        let topic := s.lookup( response,help.on )
                        case true of
                        topic is text.file   : more( topic ) ! text - show it to user
                        topic is table.pac          : help( topic ) ! subtopic - give help
                        default              : write "Sorry no help on ",response,"'n"
                end
        end
        while helping
end
```

Figure 6: A general procedure to give help

In the procedure help two different classes of data objects will cause help to be given, these structure classes are table and text.file. Notice that this program is not bound to these data types however and if another class is found a simple error message will be given. In the future this procedure may be made to respond to more structure classes as the system evolves. This may be done at low cost without the need for restructuring of existing data or the recompilation of programs using the help sub-system.

Notice that the procedure is passed a pointer to a table which itself may contain tables or text.files. In this way we can examine a hierarchy of help information. For example, we may have the general purpose help system access a database containing a table containing pointers to other tables each one containing information on different parts of the system. When we wanted to use help from a particular program we would not want to traverse all the help information available. We would therefore partially apply the helper procedure to yield a help system on a particular topic.

In this example we can see the data encapsulation of procedures taking place. When a procedure is placed in the database its environment (closure) is stored with it. When the procedure helper is committed to persistent store all the objects associated with identifiers declared before that procedure are committed. Therefore the free variable 'helping' and the procedures are automatically placed in the persistent store without any effort on the programmers part.

We have managed to construct a seamless system here. The help system may be used by another program and it, in turn, uses the 'more' procedure which we looked at earlier. We have a flow of control from one program to another in a controlled manner which is completely transparent to the user. Furthermore no complex inter module linking is required by the programmer in order to construct the system.

```
!***********  General Help System  ***********
structure text.file( *string the.chars )
structure more.pak( proc( cint,cpntr )more.proc )
structure help.pack( proc( cstring,cstring )help.proc )

!*********** DB HANDLING PROCEDURES *****************
let utilities = open.database( "utilities","friend","read" )
if utilities is error.record do
        { write "Cannot open utilities database" ; abort }
```

```
let more = proc( cpntr a.file )
begin
        let the.proc = s.lookup( "More",utilities )
        if the.proc = nil then write "Sorry can"t find more utility"
        else the.proc( more.proc )( 23,a.file ) ! apply more
end

!****************** MAIN PROGRAM **********************
let helping := true
let help = proc( cpntr help.on ) ! as defined above

let helper = proc( cstring help.on,passwd ) ! this is the general purpose help procedure
begin
        let help.db = open.database( help.on,passwd,"read" ) ! open help db
        if help.db is error.record
                then write "Cannot open help database"
        else
        begin
                write "Help Sub-system'nquit to quit'n"
                help( help.db ) ! call help procedure
        end
end

! next put help program into the database
let source = open.database( "help","friend","write" )
if source is error.record then write "Error opening database : ",source( error.fault ),"'n
else s.enter( "help",source,help.pack( helper ) )

! commit procedure to persistent store
let done = commit()
if done is error.record do
        write "Sorry - commit failed : ",done( error.fault ),"'n"
```

Figure 7: A program which puts the help procedure in a database

Conclusions

In this paper we have introduced some of the facilities that a programming system requires to be an appropriate base for an integrated project support environment. We have presented these facilities as an integrated package in the programming language PS-algol. The savings in such an approach to building IPSEs are the same as those for building a project out of an IPSE. That is lower life cycle costs.

The facilities of PS-algol that make it a suitable enabling technology for IPSEs are a transactionally safe and secure persistent object store. The transaction mechanism controls concurrent access and version installation to the object base, the type mechanism provides a mixture of static and dynamic checking facilitating delayed binding. By allowing procedures to be first class data objects we have the mechanism of separate compilation without a separate facility for it and data encapsulation.

We believe these features are necessary for supporting an IPSE and encourage language designers to incorporate them into new languages rather than have them as separate ad hoc support tools. Acknowledgements

This work is supported at the University of Glasgow by SERC grants GRC 21977 and GRC 21960 and at the University of St Andrews by SERC grant GRC 15907. The work is also supported at both Universities by grants from International Computers Ltd.

References

1. Atkinson, M.P., Bailey, P.J., Cockshott, W.P. & Morrison, R. PS-algol reference manual. Universities of Edinburgh and St Andrews PPR-8 (1984).
2. Atkinson, M.P. & Morrison, R. First class functions are enough. 4th Conference on the Foundations of Theoretical Computer Science and Software technology. Bangalore, India. (1984). In Lecture Notes in Computer Science, 181 (1984), pp 223-240. Springer-Verlag.
3. Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. An approach to persistent programming. Computer Journal 26, 4 (1983), pp 360-365.
4. Cockshott, W.P., Atkinson, M.P., Bailey, P.J., Chisholm, K.J. & Morrison, R. The persistent object management system. Software, Practice & Experience 14 (1984).
5. Horning, J.J. Some desirable properties of data abstraction facilities. ACM Sigplan Notices 11 (1976), pp 60-62.
6. Ichbiah et al., The Programming Language Ada Reference Manual. ANSI/MIL-STD-1815A-1983. (1983).
7. Liskov, B. & Zilles, S.N. Programming with abstract data types. ACM Sigplan Notices 9, 4 (1974), pp 50-59.
8. Milner, R A proposal for standard Ml. Technical Report CSR-157-83 University of Edinburgh. (1983).
9. Morrison, R. S-algol language reference manual. University of St Andrews CS/79/1 (1979).
10. Morrison, R. The string as a simple data type. Sigplan Notices 17, 3 (1982), 46-52.
11. Strachey, C. Fundamental concepts in programming languages. Oxford University Press, Oxford (1967).
12. Zilles, S.N. Procedural encapsulation : a linguistic protection technique. ACM Sigplan Notices 8, 9 (1973).