The Use of a Persistent Language in the Implementation of a Process Support System

R. Mark Greenwood, Michael R. Guy and D. John K. Robinson

Process Support Centre, OPEN*framework* Division, ICL Kidsgrove

Published in the ICL Technical Journal May 1992.

The Use of a Persistent Language in the Implementation of a Process Support System

R. Mark Greenwood* Michael R. Guy D. John K. Robinson

Process Support Centre, OPEN*framework* Division, ICL Kidsgrove

May 5, 1994

Abstract

This paper describes how a persistent language, PS-algol, was exploited to implement a process support system. The concepts of persistence are explained, together with other attributes of PS-algol which add value to it. These include first class procedures, the ability for a PS-algol program to change itself by means of the callable compiler, and the universal pointer type which permits flexible binding.

The process support system PSS supports the enactment of process models by executing process programs written in the language PML. The central feature of PML is the role. This is an object which communicates with other roles via interactions, or messages. The central component of PSS is a process control engine which supports the compilation and execution of programs written in PML. Roles are persistent processes and are represented as (first class) PS-algol procedures. Interactions are persistent messages which are held in the working data of a persistent scheduler. The PML of a role may be changed at run time by compiling new PML and binding it into the system dynamically using the mechanisms of PS-algol.

The paper outlines the structure of PSS and gives examples of the way it has relied on PS-algol for its implementation.

1 The Position of PSS

The Process Support System (PSS) [Bruynooghe et al 1991] grew out of the IPSE 2.5 Project [Warboys 1989] [Snowdon 1989]. As its name suggests, its business is the support of process; nothing about it constrains the process supported to be that of software engineering, although much of the motivation behind the system is exactly that.

ICL, in developing its flagship mainframe operating system VME, recognised in the early 1970s that a significant change in culture was necessary in the control of a large and complex software

*now at the Department of Electronics and Computer Science, University of Southampton

project. Completion of a project of that scale is significant enough, but is dwarfed by the problems of maintaining integrity of design and control throughout continuous evolution spanning several decades. ICL developed and used CADES (Computer Assisted Design and Evaluation System) [Pearson 1973] [Warboys 1980] to tackle this problem. The computer assistance was a necessary component in terms of providing storage for design and code, and providing an environment for tools such as those to handle system construction and configuration management; however, the crucial aspect of the broader system vital to the longevity of the project was the *procedures* used to control what was allowed into the computer-based system. The *process* embodied in these evolving procedures was intended to ensure that checks on, for instance, the validity of the design, or the consistency of different parts of the system, had indeed been performed; the procedures were mainly paper-based, but some utilised other, smaller, databases independent of the main CADES one, and of course more tools.

Perry and Kaiser [Perry and Kaiser 1991] describe a model for software development environments SMP based on Structures, Mechanisms and Policies and an IFCS taxonomy, Individual, Family, City and State, which highlights the issue of scale. In terms of the Perry and Kaiser SMP model then, CADES had plenty of machine support for the Structures and the Mechanisms, but precious little for the Policies. Nevertheless, much experience was gained in handling the procedures manually, especially in dealing with the need to be able to change them while active.

It would have been over-ambitious in conceiving PSS to attempt initially to tackle an organisation as large and complex as the development of an operating system, going in effect straight for Perry and Kaiser's City model in their IFCS taxonomy. The intermediate step necessary was to gain experience in providing and using a system which supported process, allowed the process to change, and allowed users to make use of tools not necessarily in the original scheme of things.

It is these aspects of PSS that tend most to distinguish it from other software engineering environments (This is discussed further in [Warboys 1989]). It is also the case that most of them are built as layers on an existing database system, whereas the PSS has a computational model (the language PML) [PML 1990] for the environment it provides. PS-algol on VME was simply an available engineering tool appropriate to the implementation of PSS.

Early experience with using PSS suggests that its potential usefulness is not limited merely to what is generally thought of as software engineering, but that it can be applied to many different kinds of process. We have no reason to suppose that our architecture is deficient in any way that would prevent it from being used for a large and complex process, although there are issues of scale and performance that will have to be tackled.

2 Persistence and PS-algol

The IPSE 2.5 Project decided to implement the PSS in the persistent programming language PS-algol [Atkinson et al 1983]. Although the choice of language was the result of debate we will not pretend that all the advantages of it were recognized in advance. We will instead give a brief description of the more important language features relevant to PSS and in later sections show how these features were exploited.

Persistence itself is a new technology for easing the task of writing programs. Persistent languages work on the principle that data and code last as long as they are reachable. One extreme way of viewing it is that all data and code created by a program lasts for ever, but for the sake of efficiency language systems have garbage collectors which discard data which is no longer accessible. From this simple definition many advantages can be gained, and some of these will be explained in the subsection on PS-algol. First we give a description of a persistent store which is required to support a persistent language.

2.1 The Persistent Store

Persistent store can be viewed as the next logical step after virtual store [Guy 1987]. The introduction of virtual store enabled a programmer to concentrate on the task of programming without having to be too aware of the size of the program. Before the advent of virtual store a large program had to be divided into overlays which were brought in from disc when required. If the overlay structure was wrong either the program did not run or it ran very slowly. Virtual store introduced what is termed a one-level store.

It was, however, still necessary for the programmer to be aware of whether the data accessed by a program was held within the program's work space or in a file or database residing outside the program. Accessing and updating the content of a file meant explicit code to transfer a copy of the data in the file into the program, update its contents, and copy it back. Persistent store avoids the need for this. Data held permanently on disc is accessed and updated with the same ease as data held in main memory.

To achieve this certain extensions are needed to the virtual store model.

The first extension allows data to be written back to permanent memory under the control of the program. With virtual store, pages are written to secondary store on disc when the virtual store manager needs the space in main store. This results in a disc copy which is usually both inconsistent with itself and incomplete; this does not matter in the virtual store scenario, because the role of secondary storage is simply to provide the transient support of main store. With persistent store, although data and code are fetched when needed, they are written back in a controlled manner to ensure the consistency of the store's contents. There are several ways that this can be done. One is to have an explicit *commit* command in the language; this can be called, for example, just before the program completes. Another is to encapsulate an operation in the program within a transaction. Another is to hide the need for commitment from the programmer altogether. In this case it may be necessary for the data being written back to permanent store to include the state of the program being executed. The first two mechanisms are provided by PS-algol. The third mechanism is the one that comes with PML, the language which the PSS implements.

The next extension treats the disc as an object store for variable-length objects, rather than a page store containing fixed-length pages. An object will relate to some language feature, for example a record, a code body, a string, or the stack frame (or local name space) of a procedure. A persistent store will contain a large number of such objects. References are no longer implemented by virtual addresses, which can be fabricated in a program, but by explicit inter-object pointers. Provided the language for programming the store implements references only in terms of these pointers, this enables the store to perform garbage collection and at the same time maintain the referential integrity which is central to persistence.

The last extension allows concurrent programs to interact when executing in the same persistent store; it adds a mechanism for controlling the way they do it. One approach is to give them access to the same space and ask them to control their interaction via semaphores. Another approach, which we have adopted, is to lock implicitly any object which is fetched into the work space of a program and to unlock it at the end of the transaction in which it was fetched. Before unlocking objects the transaction will selectively commit all the changes made within that transaction. Thus a consistent set of data is available to another program.

PML is a persistent language, as will be explained later. So is PS-algol, the language chosen for the implementation of PML. Raleigh [Kay and Rivett 1991] is implemented in MegaLog, the persistent Prolog from the ECRC [Bocca 1991]. PS-algol and MegaLog form an interesting comparison. One of the benefits of persistence as applied to a procedural language such as one of the algol family is that objects which encapsulate data inside them and which are accessed by procedural interfaces can be stored permanently without having to perform any transformation on them. This sort of benefit is already in Prolog. However, with traditional Prologs the whole 'database' has to be read into working store in one go and updated in its entirety. Persistence adds the attribute that only those objects (e.g. Prolog predicates) which are required are read, and saving the database involves writing away only those objects which have changed. These changes permit the use of databases which are larger than the virtual store available to a program, and open the way to shared usage of the system.

2.2 Some aspects of PS-algol

PS-algol [Morrison 1988] has much in common with any conventional algorithmic language, and is not described in detail here. The slightly artificial example below will help to draw out some important points. It involves the generation of a package containing an array of flags together with procedures for raising and lowering them. The code is perhaps best read from the outside in. Lines 1 to 9 define a procedure `generate.flag.control`[1] which takes an integer parameter defining the number of flags required and returns a procedure for raising one of them. This procedure, `raise.flag`, is defined on lines 3 to 7 and it in turn returns a procedure, `lower.flag`, defined on line 5, for lowering that flag. Lines 10 to 12 show how these procedures may be called.

```
line  1:    let generate.flag.control = proc(int n -> proc(int -> proc()))
line  2:    begin let flags.raised = vector 1 :: n of false
line  3:            let raise.flag = proc(int i -> proc())
```

[1] Full stops may be included in PS-algol identifiers as an aid to readability; they form part of the identifier and have no other syntactic significance.

```
line  4:          begin flags.raised(i) := true
line  5:                let lower.flag = proc(); flags.raised(i) := false
line  6:                lower.flag
line  7:          end
line  8:          raise.flag
line  9:    end

line 10:    let raise.EFTA.flag = generate.flag.control(7)
line 11:    let lower.Liechtenstein = raise.EFTA.flag(4)

line 12:    lower.Liechtenstein()
```

This example demonstrates two important features of PS-algol.

First of all, *procedures are first class objects*. The procedure `generate.flag.control` takes a parameter defining how many flags are required and creates a vector of that size (`flags.raised`); it then creates a procedure, `raise.flag`, which when called raises a particular flag. The name of this procedure is given as the last statement before **end** on line 9, and is returned from the outermost procedure. The procedure returned from `generate.flag.control` is created dynamically when `generate.flag.control` is called. As many flag vectors as are required can be created in this fashion.

Another example of returning a procedure value is given in the `raise.flag` procedure itself. When an entry is set in the flag vector a procedure is returned which can lower the flag. This is a common programming technique in PS-algol and means, for example, that only the program that raised the flag can lower it, unless of course it chooses to pass the lowering procedure to another program.

The example also illustrates *persistence*. The vector `flags.raised` is declared inside the procedure `generate.flag.control`. In conventional stack-based languages the vector would vanish at the end of the block in which it was created. In PS-algol, it persists because it is referred to by the procedure which is being returned from the block. The `raise.flag` procedure itself persists because it is reachable by use of the identifier `raise.EFTA.flag`.

Persistence and first class procedures are both powerful programming tools in their own right. Used together, they give the programmer the opportunity to create Abstract Data Types (ADTs) [Atkinson and Morrison 1985] as in the example above.

There are three other attributes of PS-algol which we single out as being of great value for the implementation of PSS.

The first is the *table*. This is a language feature which stores associations between a name and a structure or record. Entries are created by `s.enter` and looked up by `s.lookup`, as shown in the following example.

```
let tab = table()
structure flag.control.pack(proc(int -> proc()) fc.proc)
```

```
let stored.flag.control = flag.control.pack(raise.EFTA.flag)
s.enter("EFTA flag control",tab,stored.flag.control)
...
s.lookup("EFTA flag control",tab)
```

The first line creates a table. The second declares a record type which contains a single entry which is of type `proc(int -> proc())`. The next line creates a record of this type which holds the procedure `raise.EFTA.flag`. The line after this 'enters' the record into the table, associating it with the key "EFTA flag control". The last line shows how the record can be retrieved.

The table bears a strong resemblance to an indexed sequential file. It must be noted, however, that entering a record into a file creates a copy of it. `s.enter` does not copy, but creates a new pointer to the same object. Even if PS-algol did not have tables as a language feature it would still be possible to code them in PS-algol. However, having them as a defined part of the language has given us the opportunity to optimise their design in low level code as they are used extensively.

The next attribute of PS-algol is the *universal pointer type*. PS-algol is strongly typed which means, for example, that it is not possible to store a string in an integer. This enables the compiler to make more effective checks on the correctness of a program. However, in a persistent system some procedures will be written after the data on which they operate, and some data will be created after the procedures to which they are submitted. The universal pointer type means that a variable of type `pntr` may point at a PS-algol structure of any type. The type check on the structure is made when the structure is accessed rather than at compilation time. In particular, a table contains entries of type `pntr`, which permits anything to be stored in a table by wrapping it up in a PS-algol structure.

The last attribute is the *callable compiler*. It is possible for a program to modify itself by compiling some PS-algol source embedded in a string into a procedure which may then be called in addition to or alternatively to its existing procedures.

3 Process Support System Overview

The process support system PSS supports the enactment of process models by executing process programs written in the language PML. A process program is a set of roles each of which encapsulates its local data and represents an independent thread of execution. Roles can communicate with one another through interactions which are uni-directional, asynchronous, buffered channels.

The PSS system is composed of three architectural elements: the Process Control Engine (PCE), the UI servers and the tool servers (Figure 1). An instance of the UI server is run on each user's workstation. The act of logging in connects the server to the PCE. A high-level protocol is used to communicate the contents of the display to the workstation and the user's actions back to the PCE. A similar login procedure is performed by the tool server.

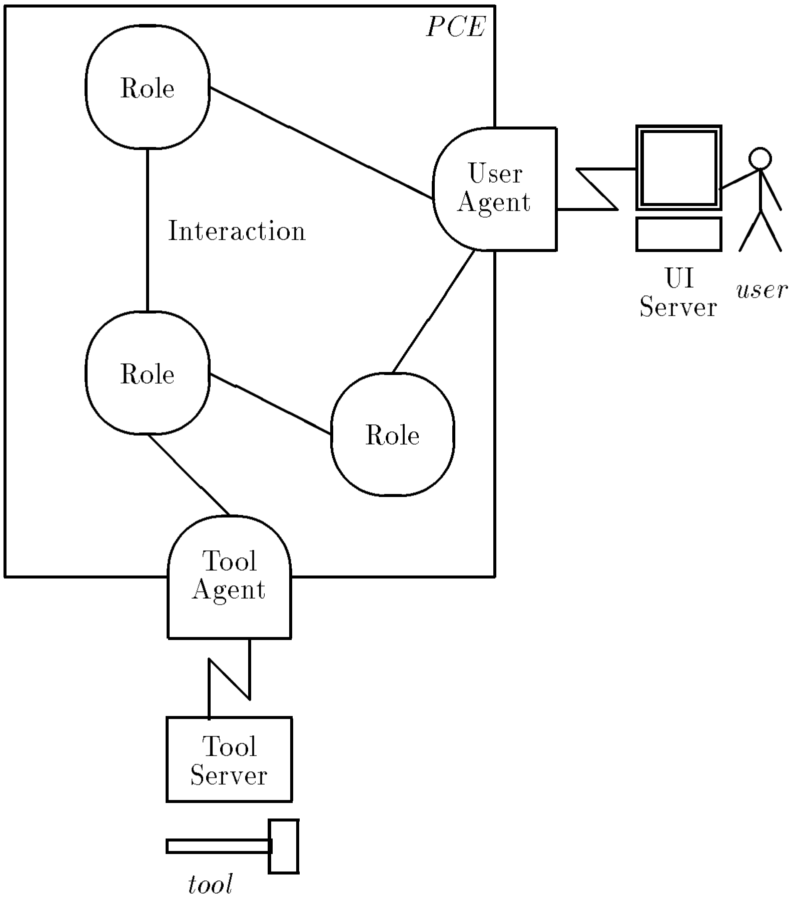

Figure 1: Architecture of the PCE

The basic scenario is that a PSS service will be started, either automatically or by an operator, and the process will re-start executing. The users and tools can log in and out of the PSS and participate in the process. Users and tools can be considered as processors, just like the machine executing the PCE, which execute their part of the process. The system is dynamically evolving with new roles, interactions, users and tools becoming involved in, or dropping out of, the process.

A user logging in to the central PCE will be connected to a user agent representing his or her view of the executing process. There is a many-to-one mapping between roles and users and correspondingly the user's view is structured into a role agenda, with one entry per role, and an action agenda for each role.

Roles are persistent processes. Their persistence is orthogonal both to the login and logout of users and to the stopping and re-starting of the PCE for activities such as machine maintenance. In addition, the system is resilient to machine and communications failures. There is no shared data between roles. This means that our requirements are not to control several processes accessing a shared store but to regularly record the current state of processes (roles) in a persistent store. Re-starting the PCE is merely a matter of retrieving the roles from persistent store and resuming their execution. The scheduling of roles is done within the PCE and therefore we have one central application which supports many users, and external tools.

Interactions are persistent messages. Once a role has sent a message and the sending of the message has been recorded in persistent store, that message is guaranteed to arrive and is not affected by login, logout and the PCE being stopped and started after the message was sent and before it arrived. It is this that makes the system a persistent object system. This persistence does not yet extend to messages sent to and from external tools. It does, however, extend to messages sent to human users because the user interface is persistent as is described later in the paper.

4 PS-algol for PSS

So, what is it about PSS that made the implementation of it in PS-algol on VME the "right" thing to do? Many factors contributed; some are listed below.

First, in general, there is the important benefit that it is simpler, quicker and less error-prone to use a truly persistent language that treats persistence as an orthogonal property of data. Strictly, the "persistence" of an object is the length of time for which it exists, irrespective of how it is stored; however, the term tends to be used in particular to refer to objects which reach non-volatile memory - the Persistent Store. (It is only then that they can be regarded as participating in an atomic transaction, and be available for sharing between concurrently executing programs.) The point about a language treating persistence as an orthogonal property of data is that it unburdens the programmer from the traditional situation of having to deal with mappings of his data which differ according to how long the data objects are going to exist.

But to be more specific, "Process" is about people (or tools) interacting with each other, performing tasks. Each person may play many roles - often intermittently. Resumption of the last state of play in a particular role is the very essence of the nature of process execution. The state may have changed in the meanwhile only as a result of the passage of relevant time, or by the interactions of others in their roles. The machine supporting the system may have suffered a power cut, or communication failure, but on the restoration of service, the roles must be unaffected. With persistence, this comes free.

Although one can argue that code is only a type of data, traditional databases tend not to cater for the storage of code. But the procedures which comprise the process are themselves liable, indeed likely, to change, and it is helpful if they and the data they operate on are subject to the same controls and support. An argument against allowing code and data to coexist has been that segregation was necessary for the integrity of the system - "don't overwrite bits of code, and don't try to execute data". With PS-algol's strong typing, extending to run-time type checking, there is no longer any reason to abandon the benefits of keeping them together.

The other aspect of change, vital as far as process is concerned, is that it must be possible for the process to evolve while live. Incorporation of new or modified code into the running system is necessary, and greatly assisted by PS-algol's treatment of code as a first-class object, by its provision of a callable compiler as a standard function, and by the flexibility of binding allowed by PS-algol's universal polymorphic pointers. A further aid to binding lies in the provision of tables, each entry representing a binding between a name and a data object.

To summarise, features of PS-algol of particular relevance and importance to the PSS implementors were:

- first-class procedures
- orthogonal persistence
- callable compiler
- strong typing
- universal pointer type
- tables

5 Process Control Engine Implementation

The Process Control Engine (PCE) provides a single environment for the development and execution of PML. A new PCE contains a single base role with a single action. This action provides the capability of accepting PML, compiling it and merging it into the functionality of the base role. The PCE is written in PS-algol, that is both the scheduler which calls the compiled PML and also the support routines which are called by the compiled PML. The PML compiler generates PS-algol which is then compiled by the PS-algol compiler.

We will not discuss PML in detail. The following section gives an example which illustrates several aspects of the system and provides a focus for the following description of how PS-algol is exploited.

5.1 PML Introduction

PML is the language developed and exploited by the IPSE 2.5 Project to write executable process programs. It has been influenced by the requirements modelling language RML [Ould 1988] and early prototype implementations were Smalltalk based. PML is a class based language with single inheritance. The class hierarchy supports three kinds of classes: entities, actions and roles. Entity class definitions create record types. Action class definitions introduce procedures, and role class definitions are schemas for subsequent creation of independently executing roles. In addition, interactions support inter-role communication.

In PML, compilation means change: the introduction of new PML into the system is always done as a dynamic change to a currently executing role instance. Because of the potential longevity of roles their dynamic evolution is vital. This is provided by a predefined action `BehaveAs` which takes a string of PML source text and a role to be changed. The PML text is compiled into a set of changes which are applied to the role.

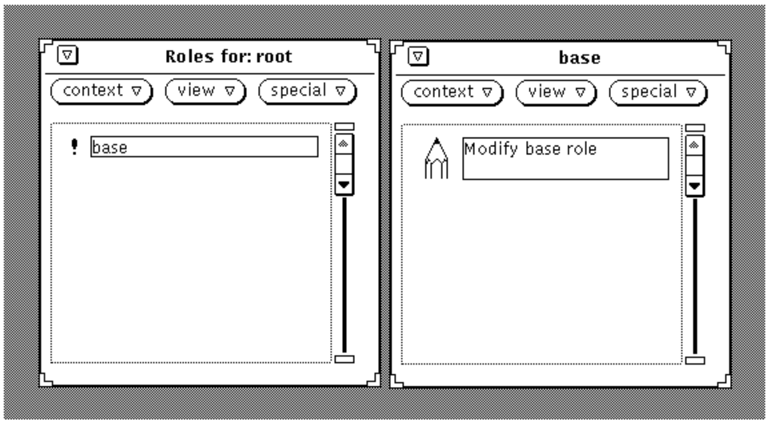

Figure 2: Initial Role

When a system is started there is one initial role, assigned to the user root, which allows the user to supply PML text to be applied to it (Figure 2). This text can change the role allowing it to start further roles. In turn these roles may have the ability to change themselves, or to be changed by roles which have a reference to them. Figure 3 gives an example piece of PML text.

The PML text in Figure 3 has three sections: the definition of a role class `ChangeableRole`, the definition of an entity class `RolePack` and modifications to the role instance. All classes are seen as being composed of named components which can be replaced in a subclass definition. The use of guard expressions, `when`, provides a way of separating executable components. Compiling this PML into the role will extend the role's environment with the classes `ChangeableRole` and

```
classes
ChangeableRole isa Role with
resources
        modification : String {'\n'}
        errors: String
        warnings: String
actions
   modify: {  Modify( agendaLabel = 'Modify this role',
                      label = 'Type in modification to this role ',
                      object = modification);
              BehaveAs( roleInst = role,
                        modification = modification,
                        compilationErrors = errors,  warnings = warnings) }
     when true
  showErrors: { ViewResource( agendaLabel = 'View compilation errors',
                              object = errors,
                              label = 'Error messages from PML Compiler');
               Assign( to = errors ) }
    when nonnil errors
endwith     ! end definition of ChangeableRole
RolePack isa Entity with
assocs
    therole : ChangeableRole
parts
    rname : String
endwith     ! end definition of RolePack

resources   ! start of modification to role instance
        name : String { 'Role 1' }
        myroles : collof RolePack { }
        newrp : RolePack
actions
   startr : { QueryString( agendaLabel = 'start role ?',
                           icon = 'UserAction',   question = 'Role name?',
                           answer = name ) ;
              NewEntity( class = RolePack,  object = newrp,
                         rname = name ) ;
              StartRole( roleInst = newrp.therole,
                         agendaLabel = name ) }
     when true
   addr : { AddToCollection( item = newrp, collection = myroles );
            Assign( to = newrp ) }
     when nonnil newrp
   ! end modification to role instance
```

Figure 3: PML Example

`RolePack`, add three new resources to the role's local data and two new actions to its behaviour. If present, existing resources and actions will be overwritten. The execution of a role's actions is controlled by triggers; for user actions, which include `Modify`, `ViewResource` and `QueryString`, the guard expression is augmented by a test that the user has selected the action from the action agenda. The effect of compiling this text is that the role to which it was applied will now offer its user (Figure 4) the chance to name and start a role of class `ChangeableRole` and store a record of this in its `myroles` resource. When a role of class `ChangeableRole` is started it will allow the user to supply some text which it will compile as a change to itself. If there are errors in the PML text then the error messages will be displayed and no change will be applied to the role.

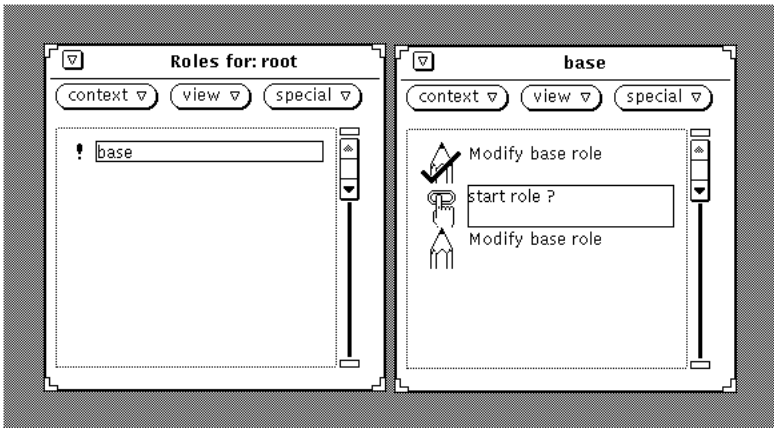

Figure 4: Initial Role after compiling Example

This example illustrates several key requirements on the PCE:

- creating a new thread of execution - `StartRole`
- intra-role scheduling through trigger evaluation
- communication with the user through predefined actions
- creation of a new record structure - `NewEntity`
- compilation of PML source text

Other facilities not illustrated include:

- giving and receiving data through interactions
- communication with external tools through predefined actions
- creating a procedure template through defining an action class
- exploitation of class-based sub-typing

5.2 Scheduling Roles

The top level of the PCE is a lightweight scheduler which is responsible for timeslicing the processor between roles. It also handles incoming messages and performs regular checkpoints. It is implemented as an abstract data type with the following interfaces:

```
Start - proc( proc() read.event; proc() wait.event )
Stop -  proc()
Schedule - proc( proc() process; bool front -> proc() )
```

The `Start` interface is called when the PCE restarts; its parameters are used for handling incoming messages. The scheduler maintains a round-robin queue of ready roles, with each entry in the queue containing a procedure which is the continuation of the role's execution. The scheduler calls the first procedure and, when it returns, moves to the next procedure and calls it. The procedures are designed to be altruistic so that no role hogs the processor. All the entries in the scheduler queue correspond to roles which have work to do. The `Schedule` interface is used to add a role to the ready queue; it returns a procedure to remove it from the queue. This is used when a role must wait for data, from its user or from an interaction. The boolean variable `front` determines whether the role is placed at the front of the scheduler queue and permits a limited amount of priority scheduling. It is used to give priority to roles processing messages coming directly from the user interface.

The system's resilience to failure comes from the scheduler's regular calling of PS-algol's `commit` standard function which ensures that the current system state is recorded on stable store (usually disc). A `commit` is performed whenever the ready queue is empty or when a message has been received from the user interface.

5.3 Capturing a Role's Execution

The execution of a role consists of a set of action calls. These are analogous to procedure calls and may be sequenced and/or nested. The role returns control to the scheduler from time to time in order to permit it to share the processor with other roles. Consequently the state of execution must be remembered and this is done by holding the nested calls in a stack.

The entries in the stack are procedures. The procedure at the top of the stack is the next part of the role's execution. Its execution may result in further procedures being pushed onto the stack. The stacker abstract data type has the following interfaces:

```
stack - proc( proc() the.proc; pntr the.diags )
schedule - proc( bool front )
deschedule - proc()
create - proc( string name -> pntr )
terminate - proc()
execute - proc()
```

The `stack` interface places `the.proc` on the top of the stack. The `schedule` interface calls the scheduler `Schedule` interface passing as a parameter the `execute` procedure. When the scheduler schedules the role it calls `execute` which then removes the procedure at the top of the stack. The procedure returned by the `Schedule` procedure is remembered for use by the `deschedule` procedure. The `create` interface is used by one role to create a stacker for any role which it starts.

The relationship between the scheduler and the stackers is an example of the exploitation of first class procedures. Holding the scheduling queue as a list of parameterless procedures permits the creation of a scheduler which is independent of the objects being scheduled. The implementation of the stacker is another example. A role is a persistent process and the parameterless procedures in the stack encapsulate the state of execution of the role: they are the *continuation* of the role [Stoy 1977].

5.4 Intra-role Scheduling

When a new role instance is created there is only one entry on its procedure stack, the role's intra-role scheduler. The behaviour of a role is data-driven and encoded in a set of actions of the form:

```
structure action.part( string name; proc(->bool) guard; proc() perform )
```

These are stored in a PS-algol table that allows new actions to be added through compilation. (The PML example in Figure 3 would have added actions for `startr` and `addr`.) The intra-role scheduler will scan this table calling each of the guard procedures until it finds one that evaluates to true and then call the corresponding `perform` procedure.

The `perform` procedure consists of a sequence of action calls. When it calls a user-defined action it places a parameterless procedure on the stack which represents the continuation of itself. This is an important use of first class procedures.

The use of tables provides provides the flexibility for dynamic change since individual entries in the table can be overwritten, and new entries added.

5.5 Communication with the User

The intra-role scheduling outlined above is complicated by the requirements of user actions and taking data from interactions. A user action is executed if its guard is true and the user has selected the action from the action agenda. The following sequence illustrates how the PCE handles user input.

1. The intra-role scheduler builds an offer of those user actions whose guards are true, sends this to the user agent along with a reply procedure and then deschedules the role.

2. The user agent sends the commands to the user interface server to update the role's action agenda with the offered actions. (Figure 4)

3. The user selects one of the actions by double-clicking.

4. The user agent receives this information and calls the reply procedure which schedules the role.

5. The intra-role scheduler identifies that there is a chosen user action and selects it. One of the user agent's support procedures will be called passing a reply procedure. Once again the role is descheduled.

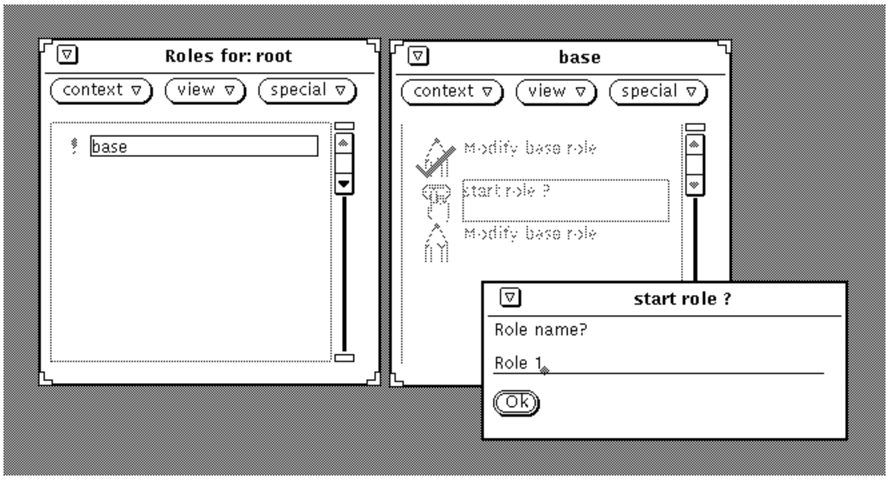

Figure 5: Naming a New Role

6. The user agent sends the commands to the user server which will create a new window on the user's screen. (Figure 5)

7. The user enters some data, for example the role name, and confirms completion.

8. The user agent receives this data and calls the reply procedure which schedules the role.

The user agent maintains a record of the current windows which are on a user's screen. As the user agent is in the persistent store this automatically provides a persistent user image. When a user next logs onto the system the display will contain the same windows as when he, or she, logged out unless subsequent execution of the process program has caused changes to the user's view. This is most graphically illustrated when a user logs in at one terminal when currently already logged in at another; the windows are transferred from one terminal to another.

One of the most convincing examples of the benefits of persistence is the ease with which a persistent user image can be maintained.

5.6 Entity Creation

In PML it is possible to create a subclass of an entity class by adding extra named fields to it. An instance of such an entity may be used wherever an instance of its parent class may be used. This feature is known as subtyping. Unfortunately PS-algol does not support subtyping; a PS-algol structure, which corresponds to a record, must have precisely the correct format when it is used. We overcome this by storing the fields of an entity in a PS-algol table, in which the names of the fields of the entity are the keys.

Although the entries in the tables are all structures they can all be referred to by a variable of type `pntr`, the universal pointer type. The table itself is of type `pntr`. The use of tables to represent all entity types does not allow the language's type system to be broken since the type checker ensures that only valid operations are attempted or a run-time type error is given. Entity creation is therefore creating a PS-algol table and entering the initialisation values. If no initial value is supplied in the class definition, as for `rname` in Figure 3, then the initial value is 'nil'. This is a valid value for all PML types: it can be tested with the functions `isnil` and `nonnil`; any other operator causes a run-time error.

5.7 Role Creation

Creating a new thread of execution is achieved by generating a structure which represents the new role instance. The following steps are taken on creating a new role instance:

1. generate a new role instance structure
2. generate a stacker for the new role and assign it to the field in the instance structure
3. introduce the role to its user agent
4. stack the intra-role scheduler
5. schedule the role

The view on the screen after creating the new role is shown in Figure 6.

5.8 Compilation and Change

A PSS system is modified by compiling PML text and binding it into the running system. PML text is first compiled into PS-algol; this in turn is compiled into a PS-algol procedure, which may then be called in the normal way. The binding is achieved by storing the procedure value for later use by updating a PS-algol table. This technique draws on that of the PS-algol object store browser [Dearle and Brown 1988].

The unit of PML compilation is a string of PML text. This will contain a number of class definitions followed, optionally, by changes to the role instance structure. Either the compilation

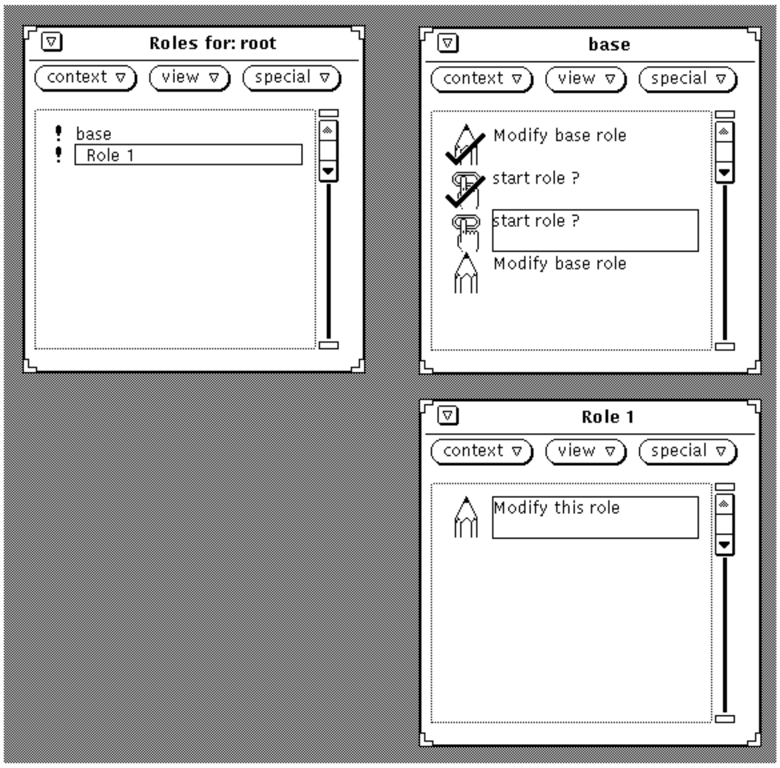

Figure 6: System with newly created `ChangeableRole`

will be successful, changing the role instance, or the compilation will fail, returning errors. During the compilation process this text is divided into segments with each segment being either a class definition or the modification to the role instance. A class definition is compiled into a generator procedure which is stored in the role's classes table. The role instance modification is compiled into a procedure which is immediately applied to the role instance. This procedure may add values to the data and actions tables. For each segment of compilation the PML is translated into PS-algol which is then compiled using PS-algol's callable compiler.

It must be remembered that all compilation changes the environment in which it takes place. This means that compilation is performed in the context of the role's current type environment. Successful compilation will involve updating this environment as well as the generation of procedures mentioned above. As a result each role has its own independent type environment. Subsequent compilations may lead to divergence between the types in the starting and started roles. One result of this is that run-time type checking may be needed for exchanging data between roles. The PS-algol source for our `ChangeableRole` is outlined in Figure 7. The line

```
proc( pntr classes -> pntr )
begin
   let instance = s.lookup( "Role", classes )( Role.gen )( classes )
   let data = instance( data )
   ! ... other declarations of local names used in this procedure
   s.enter( "modification", data, DataInst( ... ) )
   ! ... addition of other local data
   let modify.ex = proc() ; begin ... end
   let modify.guard = proc( -> bool ) ; begin ... end
   s.enter( "modify", actions,
          action.part( "modify", modify.guard, modify.ex ) )
   ! ... addition of other actions
   instance
end
```

Figure 7: Example PS-algol for PML

`"let instance = ..."` shows how the inheritance hierarchy is exploited. Each generator looks up the generator of its superclass and calls it, eventually one of the predefined classes `Role`, `Action` or `Entity` is reached, the appropriate instance structure is created and this is then modified by each of the generators. The code following this line is thus the modifications which `ChangeableRole` makes to its superclass. For each element in the role's `resources` section there is a statement to enter the element in the role's data table. For each action the behaviour of the action is translated into a procedure, for example `modify.ex`, and the expression following `when` into a procedure returning a boolean value. These two procedures are placed in an `action.part` structure which is added to the actions table. The addition of entries to tables provides the required dynamic evolution since previous entries will be overwritten. In addition the procedures in the actions table always perform a lookup on the data table to obtain any value. This provides a method of obtaining the correct entry even when further change to the role changes data table entries.

The PML compiler is built using the compiler componentry method described in [Dearle 1988]. Each of the major components, including the code generator, lexical analyser and type checker, is a generator procedure which returns a structure containing a set of interface procedures which is passed as a parameter to other components as required. This method has no problem coping with the fact that the type checker needs to be passed the role's type environment as a parameter. In addition, separate roles performing BehaveAs will generate new instances of the compiler with common code but separate data spaces which allows us to interleave their processing at the compilation segment level.

5.9 Interactions

Interactions are uni-directional message channels. They have two ends: a `GivePort` and a `TakePort`. Creating an interaction involves calling a generator procedure which returns a structure containing two interfaces.

```
structure GiveToken( proc( pntr ) Give.Data ;
                     proc( -> bool ) Data.present.g )
structure TakeToken( proc( -> pntr ) Take.data ;
                     proc( -> bool ) Data.present.t ;
                     proc( proc() -> proc() ) Wait.for.data )
```

A GiveToken corresponds to a PML `GivePort` and a TakeToken to a `TakePort`. The current implementation is one-to-one communication with one role having the `GivePort` and one role the `TakePort`; a many-to-one extension is under development. It is important to note that interactions are persistent; the data which is queued in an interaction between roles is not lost when the system is shut down. The `Wait.for.data` interface provides a further example of the use of a call back mechanism. The role which calls this interface deposits a procedure which will be called when data arrives and the result of this call is a procedure which it can use to remove this call back. Interactions are polymorphic and give an example of how we exploit PS-algol's `pntr` type. As all PML data is represented as structures the implementation of interactions will cope with all PML entities including the predefined integers, reals, strings and booleans, as well as references to role instances and `GivePort`s and `TakePort`s themselves. The ability of interactions to be passed as data down other interactions means that the inter-role communication can dynamically evolve under the control of the process program.

6 Time and Scale

The first process support system to use PS-algol was the IPSE 2.5 baseline 4 release 1 system. Design began around May 1989 and this version was completed in the September. Since then there has been ongoing development work with the seventh version of the system currently in development. The applications performance has been improved both through implementation improvements and through developments of the underlying object store on VME, which is described in [Guy and Robinson]. There is currently a major revision of the PML language and the system's handling of external tools taking account of existing experiences with using the system. A pilot application has already received live usage in ICL's Customer Service organisation.

The PSS system may be considered either large or small depending on the background of the observer. The PCE part itself comprises approximately 32,000 lines of PS-algol.

The Customer Service pilot application utilises

- 30 Megabyte persistent store, containing about
- 600,000 objects, of which
- 16,000 are procedure bodies, and which runs for
- 10 to 12 hours per day, 5 days a week.

PCE implementations exist for SUN 3, SUN 4 and ICL Series 39 machines. UI servers have been written for NeWS, and X Windows[2] running on SUN workstations and Microsoft Windows running on PCs. Tool servers exist for SUN3, SUN4, PC and Series 39 environments.

7 Conclusions

The PS-algol callable compiler, table structures and universal pointer type are all features which have been exploited in implementing the Process Control Engine. However, the most powerful feature in terms of system development is the first class procedure: the ability for procedures to be passed as parameters, returned as results and stored in structures. The benefits of persistence, like those of a good butler, are greater for the fact that they are not immediately apparent. The implementation of roles employed the full modelling power of the language without any need to consider how long the data would exist for.

The most important facts about the PSS system are that it does work and it is used. There is no doubt in the minds of the PCE developers that the persistent language PS-algol has proved its efficacy in producing an execution system for PML. It is the opinion of the developers that using a language with orthogonal persistence has been crucial both for the modelling power it provides and for the resulting programmer productivity. The system is key to our ongoing work: gaining "real" experience with a process support system, and understanding the needs of long-lived persistent applications of a reasonable size.

8 Acknowledgements

The VME-based version of PS-algol evolved significantly during the PISA Project. Both PISA and IPSE 2.5 were funded by Alvey, and thanks are due to all those involved in these projects.

[2] X Window System is a trademark of Massachusetts Institute of Technology

We should like to thank those who had the foresight to choose PS-algol for the implementation of PSS. The design of PSS was led by Bob Bruynooghe, together with Jeff Parker and Alan Dearle.

References

[Atkinson et al 1983] ATKINSON, M.P., BAILEY, P.J., CHISHOLM, K.J., COCKSHOTT, W.P. and MORRISON, R. An Approach to Persistent Programming. The Computer Journal, 1983, vol 26 no 4, pp 360-365.

[Atkinson and Morrison 1985] ATKINSON, M.P. and MORRISON, R. Procedures as persistent data objects. ACM TOPLAS, 7(4) 1985.

[Bocca 1991] BOCCA, J.B. MegaLog - A Platform for developing Knowledge Base Management Systems. International Symposium on Database Systems for Advanced Applications, Tokyo, April 1991.

[Bruynooghe et al 1991] BRUYNOOGHE, R.F., PARKER, J.M. and ROWLES, J.S. PSS: A System for Process Enactment. Presented to 1st International Conference on Software Process, Los Angeles 21-22 October 1991.

[Dearle 1988] DEARLE, A. On the Construction of Persistent Programming Environments. Universities of Glasgow and StAndrews Persistent Programming Research Report 65, June 1988.

[Dearle and Brown 1988] DEARLE, A. and BROWN, A.L. Safe Browsing in a Strongly Typed Persistent Environment. The Computer Journal 31(6):540-544, 1988.

[Guy 1987] GUY, M.R. Persistent Store - Successor to Virtual Store. In: 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. Universities of Glasgow and StAndrews Persistent Programming Research Report 44, August 1987.

[Guy and Robinson] GUY, M.R. and ROBINSON, D.J.K. The Implementation of a Persistent Store for PS-algol. In preparation.

[Kay and Rivett 1991] KAY, M.H. and RIVETT, P.J. An Overview of the Raleigh Object-Oriented Database System. ICL Technical Journal, November 1991.

[Morrison 1988] MORRISON, R. PS-algol Reference Manual. Fourth Edition. Universities of Glasgow and StAndrews Persistent Programming Research Report 12, February 1988.

[Ould 1988] OULD, M.A. and ROBERTS, C. Defining formal models of the software development process. In Software Engineering Environments. Ed Brereton, P. Ellis Horwood 1988.

[Pearson 1973] PEARSON, D. CADES. Computer Weekly, July 26th, August 2nd, August 9th 1973.

[Perry and Kaiser 1991] PERRY, D.E. and KAISER, G.E. Models of Software Development Environments. IEEE Transactions on Software Engineering, 7(3):283-295, March 1991 7.

[PML 1990] PML Reference Manual. IPSE 2.5 Project Document STL/608/00070, Dec 1990.

[Snowdon 1989] SNOWDON, R.A. An Introduction to the IPSE 2.5 Project. ICL Technical Journal, 6(3):467-478, 1989.

[Stoy 1977] STOY, J.E. Denotational Semantics. The MIT Press, 1977.

[Warboys 1980] WARBOYS, B.C. VME/B a model for the realisation of a total system concept. ICL Technical Journal, November 1980.

[Warboys 1989] WARBOYS, B.C. The IPSE 2.5 Project: A Process Model Based Architecture. In: Software Engineering Environments research and practice. Ed Bennett, K.H. Ellis Horwood 1989.