An Example of Process Change

Robert Snowdon*

ICL, West Avenue, Kidsgrove, Stoke on Trent, ST7 1TL, England

Abstract. Processes in businesses and other organisations are not set in concrete. They have to be flexible and responsive to stimuli which cause them to change. Process technology must relect these characteristics if it is to find realistic application. This paper illustrates how the evolutionary nature of processes may be supported in a particular process technology, that originally developed in the IPSE 2.5 project.

1 Introduction

The traditional view of software development is that it is a process starting with the establishment of requirements, moving through phases such as design and implementation and finishing in something called "maintenance". Of course, we all know that reality is rather different with the process being rather more cyclic in nature, phases overlapping one another and so on. Process technology must not be restricted to the simple view if it is to find credible application. Change and concurrency are essential elements of this and many other application domains.

Such concerns were taken into account in the thinking behind the IPSE 2.5 project [4] and in the process technology originating in that project and subsequently developed as the PS system and the language PML [1].

This paper illustrates the particular features of this technology which are concerned with these aspects. It does this by means of a concrete example utilising a particular paradigm of change developed in the IPSE 2.5 project. This paradigm is known as PMMS (Process Model for Management Support). It provides an integration of the notion that a process is something which is developed and changed as a process itself.

Example PML encodings are included, the intention being to allow the reader to see in detail what is going on, what needs to be described and how it is described. A brief explanation of the form and effect of PML descriptions is covered in sections 2 and 3. This should be sufficient for the purposes of this paper although it is, of course, not a complete definition of PML or PS.

2 Some Basic Points about PS and PML

PML is a language in which a process description is expressed. The primary purpose of a description is to be the basis for the enactment of the process by appropriate "agents" (people and tools). Enactment support is provided by means of a computer system (called

* This work was supported by a grant from the Royal Society and the Science and Engineering Research Council while the author was a Visiting Fellow at the Department of Computer Science, University of Manchester, England

the PCE, the Process Control Engine) which uses PML descriptions to provide appropriate capabilities to the agents participating in the process itself. More details of these concepts can be found in, for example, [4],[1] and [5].

In PML a process is considered as comprising a number of **roles** which communicate through **interactions**.

A role can be thought of as being rather like a single-threaded process having a private set of resources and some knowledge of its environment. It may help to envisage a role as being "played" by a person, although strictly this is rather a limited interpretation of a more general concept. An interaction is an asynchronous, uni-directional, buffered channel by which roles communicate.

PML is a language based on the idea of **Class**. The primitive notions of role and interaction are supported by `Role Class` and `Interaction Class` definitions respectively. A class defines properties for its members, objects which are instances of the class. PML also supports `Entity Class` and `Action Class` definitions of which such properties may be formed. Subclassing is supported (with some restrictions) through an `isa` hierarchy.

The predefined classes of PML are available in all roles. A role (that is an instance of `Role Class`) may also possess a set of class definitions of its own. A newly created role will possess class definitions depending on its definition and the class definitions of its creator.

A role exhibits behaviour. That is, actions operate on the resources of a role and possibly on its environment according to a description of behaviour associated with the role via its membership of a particular subclass of `Role Class`. This behaviour is described in terms of actions which may occur if appropriate conditions (known as triggers) are satisfied. The single threaded nature of roles means that only one action will occur at any one time.

PML provides a number of predefined subclasses of `Action Class`, though new subclasses may also be defined. Of particular note for this paper are definitions of action subclasses to create new roles (the subclass `StartRole`) and to modify existing roles (the subclass `BehaveAs`). Actions are provided to support communication between roles in the form of the subclasses `Give` and `Take`. Interactions are identified in roles by `GivePorts` and `TakePorts`, specialised as to the objects that may be transmitted.

Finally, PML incorporates the idea of **Agents**. An agent represents a tool or person participating in a process. The `Agent Class` is a subclass of `Role Class`, so agents are roles and communicate with other roles via interactions. In practice PML provides specially defined actions (of `UserAction Class`) which encapsulate much of the communication between roles and agents which represent people. The differences between agents which represent tools and agents which represent people are captured by predefined subclasses `ToolAgent` and `UserAgent` of `Agent Class`[2].

3 The Starting Position

Every newly delivered PS system is already primed to support the enactment of a very simple process. This has two roles, one of which is a user agent supporting the participation of a person known as 'root'. The other role is known as `base` and is defined to interact with the 'root' user agent in a very specific way as shown in the following PML extract.

[2] PML constructs for tools are not discussed here.

```
resources
        modification : String {''}
        errors : String
        warnings : String

actions
  !--- Provide an action to modify the base role

   modify: {
           Modify(agendaLabel = 'Modify base role',
                  label = 'Type in modification to this role ',
                  icon = 'Pencil',
                  object = modification);
           BehaveAs(roleInst = role,
                    modification = modification,
                    compilationErrors = errors,
                    warnings = warnings)
           }

  !--- While there are errors, provide an action to view them

   showErrors: {
               ViewResource(agendaLabel = 'View compilation errors',
                            object = errors,
                            label = 'Error messages from PML Compiler');
               Assign(to = errors)
               }
   when nonnil errors

  !--- While there are any warnings, provide an action to view them

   showWarnings: {
                 ViewResource(agendaLabel = 'View warnings',
                              object = warnings,
                              label = 'Warning messages from PML Compiler');
                 Assign(to = warnings)
   when nonnil warnings
```

The `base` role has three resources, all of which are instances of `String Class`, itself a subclass of `Entity Class`.

The `base` role also has three action properties labelled `modify`, `showErrors` and `showWarnings`. These actions are controlled by triggers, indicated in the description by clauses commencing with the keyword `when`. The `modify` action does not include an explicit trigger, this being equivalent to one which is always true.

An action may occur if its trigger is true. Thus the above definition states that the `modify` action may always occur, whilst `showErrors` may only occur when there are errors (i.e. the resource `errors` is not empty) and `showWarnings` may only occur when there are warnings (i.e. the resource `warnings`) is not empty. If more than one trigger is true, than an arbitrary choice is made, subject to special considerations if actions involve interaction with other roles or agents. Specifically for actions which will cause user actions to occur, the effect

is to give the choice to the appropriate user agent and thus to the person the user agent represents.

Each of `modify`, `showErrors` and `showWarnings` is a sequence denoted by the { and } brackets and the ';' operator. In each case the first element of the sequence is actually a user action, the effect of which, if selected, is to interact with the 'root' agent to offer an action to whoever is participating in the capacity of 'root'. In the initial state of the process so described, only the `modify` action can occur so the user participating as 'root' is offered the single action identified by the user action `Modify` with its various parameters. In the PS system this will have the effect of offering the 'root' user a window such as is shown in Fig. 1.

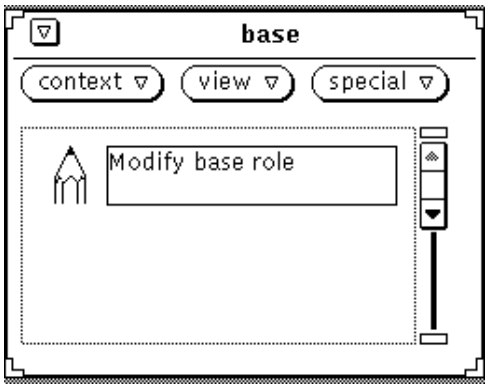

Fig. 1. User modification action.

The reader may care to relate the effect on the user interface with the actual parameters of `Modify`. Once the 'root' participant has selected the offered action, then a string of PML must be entered. This is stored in the string named `modification`, the `object` parameter for `Modify`. This string is then used by the second element of the sequence, `BehaveAs`, to apply the PML as a modification to the `base` role, the name `role` used as the actual parameter denoting self identification. Errors or warnings detected in making the modification are assigned to the resources `errors` and `warnings` respectively. Notice that such assignments would trigger the offering to the 'root' participant of actions to view the error or warning strings. In the case of successful modification the `base` role object will be changed and thus possess new or modified properties.

4 PMMS

By using the `modify` action it is, of course, possible to enhance the initial process in any way desired. Whereas such freedom may be valuable, it can easily lead to chaos and confusion. In practice there is a need for evolution and change to be based on something more systematic.

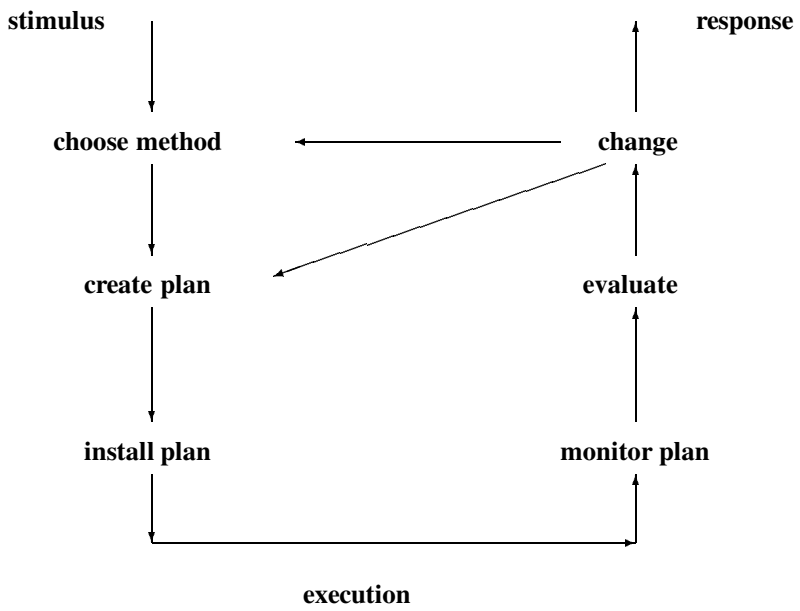

Fig. 2. Basic management cycle.

The IPSE 2.5 project developed several such schemes, one of which is shown in this paper and is known as PMMS (Process Model for Management Support).

The PMMS scheme is based on a cycle as shown in Fig. 2. It reflects a simple conceptualization of the process of establishing and managing activity. In response to some stimulus (usually called a "terms of reference") it is decided to establish some activity (a "project"). The approach to carrying out such activity needs to be determined (i.e. decide the "method" by which the activity will be effected) and applied to the particular problem in hand. This produces what is often called a plan, showing not only the timing and interdependent nature of activities, but also the resources to be deployed, the results to be produced and so on[3]. The plan is then installed and carried out (executed). The activities are monitored to ascertain their behaviour in comparison with expectations. Such monitoring provides the basis for the assessment of the acceptability of the activities carried out under the plan and for any necessary changes to be made. The process is essentially continuous until the original aims, or their revisions, are deemed to be met. Further 'stimuli' may be received during the cycle and 'responses' given.

The PMMS provides a basic structuring and encoding for the process described above. The major elements of the structure are four roles.

[3] The term *plan* is used here in the sense of something which might be familiar to a project manager. There are obvious similarities with the technical AI *planning* paradigm such as described in [2]. However, the relationships between such interpretations and a change model such as PMMS are not considered in this paper.

1. managing. This is the hub role. It receives/establishes terms of reference and acts as the initiator of PMMS behaviour.
2. technology. This role is responsible for determining the method to be employed in carrying out activities identified by the managing role in the terms of reference.
3. logistics. This role is responsible for producing the plan, based on the method described by technology, to satisfy the requirements identified in the terms of reference.
4. administrating. This role installs the plan produced by logistics. It should be noted that installing such a plan may also imply changes to (at least) the managing role in order that monitoring information and results can be obtained as the plan is carried out.

The above description tends to emphasise the initial establishment and carrying out of a plan in respect of a given terms of reference. Of course, as changes are identified, the relevant sub-components (managing, technology, logistics, administrating) will be involved to cause the necessary amendments to be made. This could also include changes resulting from amendments to the terms of reference.

Although the example to be described below uses the PMMS concepts to set up an environment for the enactment of a very simple process, it should be clear that the PMMS idea can be applied recursively whereby an organisation can be described (and enacted) as a number of inter-related PMMS structures, each with its own responsibilities and scope of change.

5 An Application

To illustrate these various ideas in practice in PML, the following describes how a set of PMMS roles are established from the initial process. These roles are then used to set up a specific process called simpleBanking which is concerned with the running of a bank account. This example is particularly trivial but serves to illustrate the basic principles.

5.1 Basic Approach

The idea is to set up a PMMS framework to oversee a number of applications, one of which is simpleBanking. Figure 3 shows a summary of the intended structure.

The PMMS roles will be set up from the initial process. In turn, these roles will be modified to set up a role identified as SBdoing which will be responsible for establishing instances of the simpleBanking application "on demand". On each occasion a report is prepared and transmitted to the PMMS managing role, where it is accumulated with earlier reports.

Notice that there are just two roles in simpleBanking. The first, identified as Customer0, represents the customer in a bank, the other, Customer1, represents the handling of the account itself[4].

[4] These names are derived from the scheme used in [3] from where the example itself is abstracted.

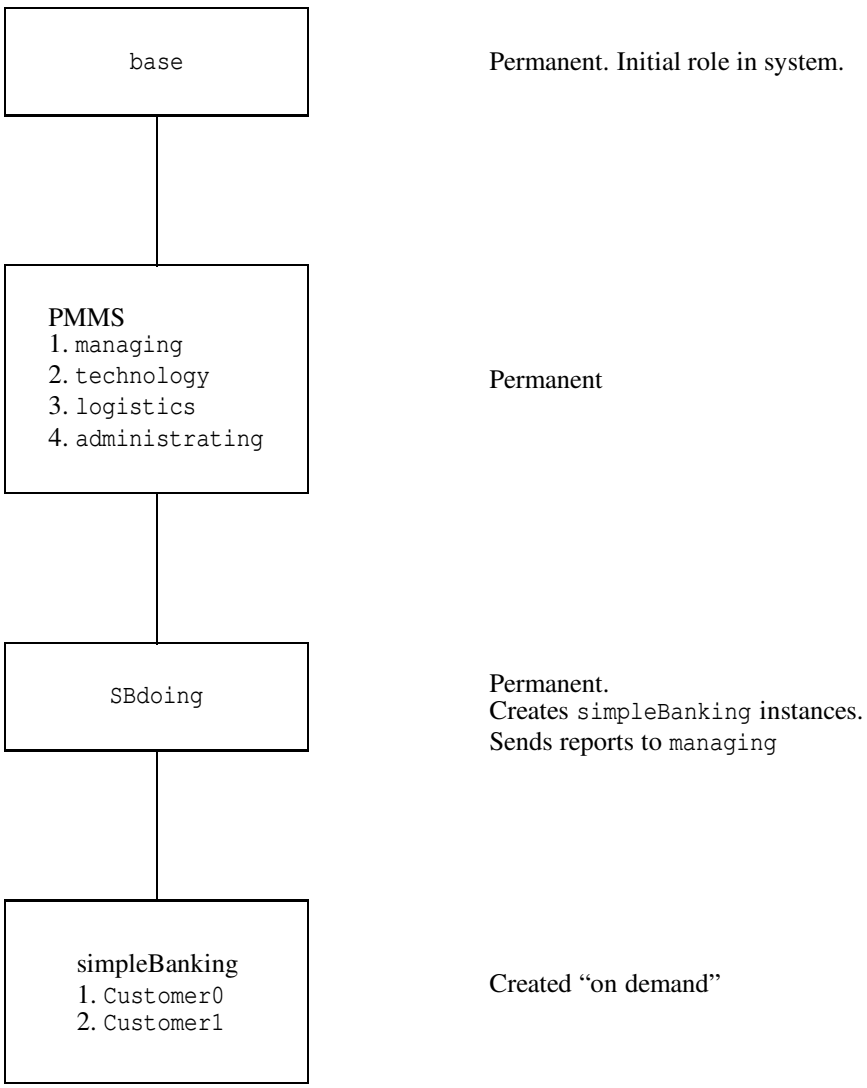

Fig. 3. Basic structure.

5.2 Setting Up the PMMS Roles

From the initial process, the first task is to establish a set of PMMS roles. Recall that the initial process is set up so that the person who is participating as 'root' can select the `modify` action. Selecting this action allows the input of PML text. Normally this will be from a local file. The following shows an excerpt from the PML defining the classes for PMMS. For brevity, only the definition of the `Technology` class is shown completely. The input of these definitions will result in their being added to the `base` role as described in Sect. 3.

```
BaseModelling isa Role with ...

PMMSRole isa BaseModelling with ...

Technology isa PMMSRole with

assocs
  methodRequested : takeport Bool
  log : PMMSRole

resources
  makeNewMethod : Bool{false}
  makingmethod : Bool{false}
  dummy : Bool{false}
  errors : String
  warnings : String

actions
  makeMethod : {QueryAlert(answer = dummy,
                  question = 'Use Modification (as required)
                            and then makeMethodAvailable',
                  agendaLabel = 'Make Method');
                Assign(from=true,to=makingmethod)}
           when
                     ~makingmethod & makeNewMethod = true

  askedForNewMethod : Take (gram = makeNewMethod,
                            interaction = methodRequested)
          when
                 (makeNewMethod = false & nonnil methodRequested)

  assist : QueryAlert (answer = dummy,
                    question = 'methodRequested still nil!',
                    agendaLabel = 'Help')
          when
                    (isnil methodRequested)
end with

Logistics isa PMMSRole  with ...

Administrating isa PMMSRole with ...
```

```
Managing isa BaseModelling with ...
```

At this time no new objects have been created. Rather, new class definitions have been added to the `base` role.

Instances of the classes `Managing`, `Technology`, `Logistics` and `Administrating` are now created. This is done by first creating an instance of `Managing` and giving it a job to do. This instance will be known in `base` as `md1`. The definition of `Managing` (not shown) creates the other three with general properties and establishes communication arrangements between them.

The detail of how this is done is not given here. It follows the usual scheme except that the modification applied this time to `base` provides a new action property enabling the participating 'root' person to start up an appropriate `Managing` role and thus put the whole PMMS creation process in train.

The next modification shown introduces a new user agent to the system. As things stand, the only user agent is that identified with the person known as 'root'. However, it would be nicer to have a real person participate in the process, so the next thing to do is introduce this person as the person really enacting the PMMS roles.

The `Managing` role was started from the `base` role. As a result `base` has knowledge of this new role through the property `md1`. This knowledge can be used to modify the `managing` role so that the user agent associated with it is not 'root' but a new user agent called 'ras'.

```
assocs
    ras : UserAgent

actions
    delegatemd1 : {StartRole(roleInst=ras,
                             password='ras',
                             username='ras');
                   BehaveAs(compilationErrors=errors,
                            warnings=warnings,
                            roleInst=md1,
                            useragent=ras,
                            modification=' ')}
    when isnil delegatemd1
```

This modification to `base` introduces a new property to identify a new user agent and an action which will create the user agent and modify the `useragent` property of the role identified via `md1` to be the new user agent. One minor point to note here is the use of an action property name in the action trigger. The meaning here is that the trigger will only be true if the action has not previously occurred.

As soon as the modification is made to `base`, the new `delegatemd1` action can occur and, as a result, create the new user agent and change the user agent associated with the `managing` role (`md1`).

We can now forget the person known as 'root'. As enactment continues the participant will be 'ras' not 'root'. For the purposes of this paper no other user agents are used. Thus, as will be seen, 'ras' is also the participant in all the other roles that will be defined! However, the means of introducing other participants should be clear.

5.3 Using the PMMS

The four PMMS roles established are unspecific in the behaviour so far defined for them. In terms of the behaviour illustrated in Fig. 2 the process is at a point prior to **choose method**. However, what happens next **is** defined and may be enacted to move things forward.

The task for which the PMMS roles have been established is to run applications of the `simpleBanking` process, making reports to the manager on each occasion. This corresponds to the setting up of a `simpleBanking` application, **execution** corresponding to the enactment of `simpleBanking` itself and **monitor plan** corresponding to the reports made back to the PMMS manager.

Thus the immediate next stages must correspond to **choose method** and **create plan**. According to the PMMS concept, it is the responsibility of the `technology` role to supply the PML class definitions describing the method of carrying out `simpleBanking` applications. The approach, as always, is that the `technology` role supports a `UserAction` to modify itself as a means of acquiring new resources and definitions. The PML relating to the setting up of the `simpleBanking` application and the appropriate reporting is given next. Complete definitions are not given. Rather each example contains some relevant part in detail with the remainder shown only in outline.

New resources are given to the `technology` role in the form of three 'strings' plus a set of other associated definitions. Notice that the quote character (') needs to be escaped by prefixing it with the character '\' within a string itself.

The three 'strings' are:

1. `sbadministrating`. This is PML to be eventually applied as a modification to the PMMS `administrating` role. The behaviour of this role represents the operation **install plan** in Fig. 2 as can be seen by the three actions found at the end of the string. These are labelled `createInteractions`, `changemanager` and `startsb`. Note that this text contains not only the application code itself (including `Customer0`, `Customer1` and `SBdoing` as role classes) and the actions to install it, but also an action to modify the PMMS manager role to receive reports and accumulate them. The actual definition of these last modifications is in the string `sbmanaging` as described below.

```
sbadministrating : String{'

   classes

   Taggedrecord isa Entity with ...

   Investment isa Taggedrecord with ...

   Payin isa Taggedrecord with ...

   Withdraw isa Taggedrecord with ...

   Terminate isa Taggedrecord with ...

   Customer0 isa Role with
     assocs
```

```
    c : giveport TaggedRecord
  resources
    transrec : Taggedrecord
  actions
    inv : {GetNew(label=\'INVESTMENT\',
                  object=transrec,
                  class=Investment,
                  agendaLabel=\'Invest\',
                  icon=\'Idea\');
           Give(interaction=c,
                gram=transrec) }
      when isnil inv

    payincase : ...
      when nonnil inv

    withdrawcase : ...
      when nonnil inv

   terminatecase : ...
     when nonnil inv

   termconds
     nonnil terminatecase
end with

Customer1 isa Role with
  assocs
    c : takeport TaggedRecord ...
  resources
    genselectflag : Bool {false}
    cr : Taggedrecord ...
  actions
    inv : {Take(interaction=c,
                gram=cr);
           ViewObjectNow(object=cr as Investment,label=\'INVEST\')}
      when nonnil c & isnil inv

    genselect : ...
      when nonnil inv & ~genselectflag

    payincase : ...
      when genselectflag & cr.tag=\'PAYIN\'

    withdrawcase : ...
      when genselectflag & cr.tag=\'WITHDRAW\'

    terminatecase : ...
      when genselectflag & cr.tag=\'TERMINATE\'

  termconds
```

```
    nonnil terminatecase

end with

SBdoing isa Role with
assocs
   c0 : Customer0
   c1 : Customer1
   c0toc1 : giveport TaggedRecord
   c1fromc0 : takeport TaggedRecord
   c1ending : takeport Bool
   endinginc1 : giveport Bool
   ...

resources
   sbreport : String
   date : String
   inprogress : Bool {false}
   ...

actions
   simplebanking :
      {UserAction(agendaLabel=\'simplebanking\');
       GetNewNow(label=\'Details of SB usage\',
                 object=sbreport);
       Stringdate(s=date);
       Assign(to=sbreport,
              from=date ++ \'   \' ++ sbreport);
       NewInteraction(giver=c0toc1,
                      taker=c1fromc0);
       NewInteraction(giver=endinginc1,
                      taker=c1ending);
       StartRole(roleInst=c0,
                 c=c0toc1,
                 agendaLabel=\'Customer\');
       StartRole(roleInst=c1,
                 ending=endinginc1,
                 c=c1fromc0);
       Assign(to=inprogress,from=true)}
   when ~inprogress

   modelending : ...
   when nonnil c1ending

   sendreport : ...
   when c1message & inprogress

   terminate : ...
   when nonnil getTerm

   getTerm : ...
```

```
      when nonnil directions & isnil getTerm

   termconds nonnil terminate

   end with

   assocs
      sbd : SBdoing
      sbdtomgr : giveport String
      mgrfromsbd : takeport String
      sbdfrommgr : takeport String
      mgrtosbd : giveport String

   resources
      sbmanaging : String

   actions
      createInteractions:
         {NewInteraction(taker=mgrfromsbd,
                         giver=sbdtomgr);
          NewInteraction(taker=sbdfrommgr,
                         giver=mgrtosbd)}
      when isnil createInteractions

      changemanager :
         BehaveAs(roleInst=manager,
                  modification=sbmanaging,
                  compilationErrors=errors,
                  warnings=warnings,
                  sbdirections=mgrtosbd,
                  sbreceive=mgrfromsbd)
         when isnil changemanager & nonnil createInteractions

      startsbd :
         StartRole(roleInst=sbd,
                   agendaLabel=\'Simple Banking\',
                   directions=sbdfrommgr,
                   reportup=sbdtomgr)
         when isnil startsbd & nonnil createInteractions

'}
```

2. `sblogistics`. This is PML to be applied as a modification to the PMMS `logistics` role which is able to modify the definitions given to it by `technology` before the decision is taken to install the new process via the `administrating` role. The `sblogistics` string contains appropriate facilities for this purpose.

```
sblogistics : String {'

   resources
```

```
      sbmanaging : String{\'!
\'}
      sbadministrating : String{\'!
\'}
      installsbmethod : Bool {false}

   actions
      installsbproject :
         QueryAlert(agendaLabel=\'Install/change sb project\',
                    question=\'Confirm ready to install/change sb\',
                    answer=installsbmethod)
      when ~installsbmethod & isnil errors & isnil warnings

      setupsbproject :
         {BehaveAs(modification=sbadministrating,
                   roleInst=admin,
                   compilationErrors=errors,
                   warnings=warnings,
                   sbmanaging=sbmanaging);
          Assign(from=false,to=installsbmethod)}
      when installsbmethod & isnil errors & isnil warnings


      viewsbadmod :
          ViewObject(object=sbadministrating,
                     label=\'sbadmin\',
                     agendaLabel=\'view sbadmin\',
                     icon=\'TextView\')
      when nonnil sbadministrating

      viewsbmgrmod :
          ViewObject(object=sbmanaging,
                     label=\'sbmanaging\',
                     agendaLabel=\'view sbmgr\',
                     icon=\'TextView\')
      when nonnil sbmanaging

      changesbadmod :
          Modify(object=sbadministrating,
                 label=\'sbadmin\',
                 agendaLabel=\'change sbadmin\',
                 icon=\'Pencil\')
      when nonnil sbadministrating

      changesbmgrmod :
          Modify(object=sbmanaging,
                 label=\'sbmanaging\',
                 agendaLabel=\'change sbmgr\',
                 icon=\'Pencil\')
      when nonnil sbmanaging
'}
```

3. `sbmanaging`. This is PML to be applied at installation time to modify the behaviour of the PMMS `managing` role to enable monitoring etc. of the installed process. In this case the text identifies interactions for communication between the PMMS manager and the `SBdoing` role which will be set up as part of the establishment of the application.

```
sbmanaging : String {'

   classes

      SBlog isa Entity with ...

   assocs
      sbreceive : takeport String
      sbdirections : giveport String

   resources
      sbreport : SBlog()
      sbmessage : String
      endsb : Bool {false}
      stopmessage : String {\'stop\'}

   actions
      getsbreport :
         { Take(interaction=sbreceive,
                gram=sbmessage);
           Assign(to=sbreport.text,
                  from=sbreport.text ++ \'
                           \' ++ sbmessage)}
      when nonnil sbreceive

      viewsblog : ...
      when nonnil sbreport

      stopsb : ...
      when endsb & isnil stopsb

      stopsbQuery : ...
      when ~endsb
'}
```

The final modifications made at this time are to the `technology` role itself to alter its behaviour.

```
resources
   makeavailableflag : Bool {false}

actions
   makeavailable :
```

```
   {UserAction(agendaLabel='makeMethodAvailable',
               icon='FaceHappy');
    BehaveAs(roleInst=log,
             modification=sblogistics,
             sbmanaging=sbmanaging,
             sbadministrating=sbadministrating,
             compilationErrors=errors,
             warnings=warnings);
    Assign(to=makeavailableflag,from=true)}
    when makingmethod & nonnil log

methodmadeok : ...
    when makeavailableflag & isnil errors & isnil warnings

viewsblogmod : ...
   when nonnil sblogistics

viewsbadmod : ...
   when nonnil sbadministrating

viewsbmgrmod : ...
   when nonnil sbmanaging

changesblogmod : ...
   when nonnil sblogistics

changesbadmod : ...
   when nonnil sbadministrating

changesbmgrmod : ...
   when nonnil sbmanaging
```

Once these four amendments are made by `technology` to itself, then everything is in place to achieve the desired effect. The person participating in the `technology` role in the enacted process (i.e. 'ras') may make the method definitions available by selecting the action (in the modified `technology` definition) labelled `makeavailable`. This modifies the `logistics` role by the PML text in the string `sblogistics`.

Then whoever is participating in the logistics role ('ras' again) can make any appropriate changes to the definitions received from `technology` (which corresponds to **make plan** in Fig. 2), before installing the `simpleBanking` application by selecting the action labelled `installsbproject` (which can be seen near the beginning of the definition of the string `sblogistics`. Installation has the effect of modifying the `administrating` role based on the modification in the string `sbadministrating`. This in turn establishes an instance of the `SBdoing` role and modifies the `managing` role to receive and view reports.

5.4 The Application

Once the action labelled `installsbproject` is complete, all of the PMMS roles will have been appropriately modified for the application and a `SBdoing` role will have been created.

Sbdoing is defined to possess four actions (see the definition of the string sbadministrating). The first of these (labelled simplebanking) causes the establishment of a new Customer0 role (representing a new bank customer) and a new Customer1 role (representing the new account for the customer). The bank customer can invest, make payments and withdrawals and finally cease to be a bank customer. Each transaction is reflected in the bank account by appropriate messages. A more realistic model might actually maintain the bank account! Figure 4 shows a typical interface for the customer.

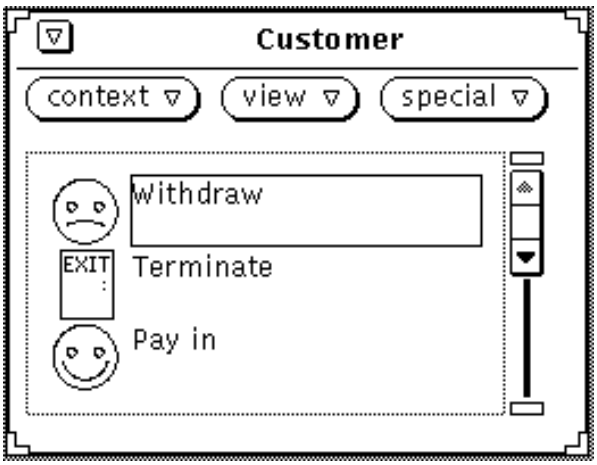

Fig. 4. Customer interface.

When the customer terminates the account, the SBdoing role completes a report and sends it to the PMMS managing role. Here it is accumulated with other reports and may be viewed according to the modifications made to the managing role by the string sbmanaging.

6 Discussion

The preceding section described, in some detail, how a system supporting the enactment of a simple process can evolve to a system supporting the enactment of some other, defined, process and how this evolution itself may be supported as an enacted process in a continuous fashion. The ability to do this is inherent in the language PML and the underlying PS support system.

The specific features of PML/PS that allow this evolution are based primarily on the reflexivity of the language (via the primitive action BehaveAs) and its role based structure.

It is clear that, despite the ability to evolve systems in these ways, these features are primitive and also that there are many other issues which need further work. The PMMS paradigm within which these features have been illustrated in this paper are based on a particular view of how the definition and enactment of a process might be "controlled" from

some form of management process. Such an approach provides an integration of management and work which, as work in IPSE 2.5 showed, is a very general building block. It can also have application in areas conceptually different from that of management in a classic sense. For example, the development of a software system can be considered from such a viewpoint. Here the **plan** of Fig. 2 might be the software itself. The **execution** component represents the use (execution) of the software itself with **monitoring** and **evaluation** being activities which are concerned with determining whether the software is behaving properly or not. Such an interpretation might be applied in terms of testing software itself, or, more broadly, supporting the use of software within customer installations.

Paradigms different from PMMS may be considered. One of which some experience has been gained is based on a problem solving model. Clearly there are others each with its own emphasis.

Change brings with it classic configuration control problems. Each cycle of the process of Fig. 2 may make changes. Is it possible (is it easy or difficult) to maintain a history of these cycles for purposes such as audit or knowledge accumulation? At a primitive level the answer is in the affirmative, However, the technical feasibility of doing something does not necessarily make it possible in reality. The experiences gained with the use of PML/PS to date indicate that there is still potential for much further work, leading perhaps to a richer modelling technology.

References

1. R. F. Bruynooghe, J. M. Parker, and J. S. Rowles. PSS: A System for Process Enactment. In Mark Dowson, editor, *Proceedings of the First International Conference on the Software Process: Manufacturing Complex Systems*. IEEE Computer Society Press, 1991.
2. Karen E. Huff and Victor R. Lesser. A Plan-based Intelligent Assistant That Supports the Software Development Process. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 28-30 November, 1988, Boston*. ACM Press 1988.
3. M. A. Jackson. *System Development*. Series in Computer Science. Prentice/Hall International, 1983. C. A. R. Hoare, Series Editor.
4. R. A. Snowdon. An Introduction to the IPSE 2.5 Project. In Fred Long, editor, *Software Engineering Environments*. Springer-Verlag, Lecture Notes in Computer Science 467, 1990. Also in ICL Technical Journal, May 1989, Vol 6 No 3.
5. Brian Warboys. The IPSE 2.5 Project: Process Modelling as the basis for a Support Environment. In *Proc. of the First International Conference on System Development Environments and Factories — SDEF'89, 9-11 May 1989, Berlin*, May 1989.

Acknowledgments

Much of the original work on PMMS was carried out during the IPSE 2.5 project by Clive Roberts, Alun Jones, Keith Harrison-Broninski and David Gourley of Praxis Systems, Bath, England[5] with encouragement and direction from Mike Tordoff of ICL.

[5] Clive Roberts is now with Co-Ordination Systems, Chippenham, England