PADM: Towards a Total Process Modelling System

R.F. Bruynooghe, R. A. Snowdon,
ProcessWise Portfolio Centre,
ICL, Kidsgrove,
Staffordshire ST7 1TL U. K.

rfb, ras@kid01pml.icl.co.uk

R. M. Greenwood,
Declarative Systems Laboratory,
Department of Electronocs and Computer Science,
University of Southampton,
Southampton SO9 5NH U. K.

M. Greenwood@ecs.soton.ac.uk

I. Robertson, J. Sa, B.C. Warboys,
Department Computer Science,
Univertsty of Manchester ,
Manchester M13 9PL, U. K.

ir, jsa, brian@cs.man.ac.uk

To be published in:

'Software Process Modelling and Technology', edited by A. Finklestein, J. Kramer and B. Nuseibeh, Advanced Software Development Series, Research Studies Press.

CHAPTER 1

PADM: Towards a Total Process Modelling System

R. F. Bruynooghe

ProcessWise Portfolio Centre, ICL, Kidsgrove, Staffordshire ST7 1TL, UK.

R.M. Greenwood

Declarative Systems Laboratory, Department of Electronics and Computer Science, University of Southampton, Southampton SO9 5NH, UK.

I. Robertson, J. Sa, B.C. Warboys

Informatics Process Group, Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK.

1. Introduction

We describe the current state of the ongoing work of the Informatics Process Group at Manchester University (IPG) in developing a total methodology for the acquisition, modelling and enactment of Software Process Models. This emphasis implies that we are concerned with the production of a Process Analysis and Design Methodology (PADM) [27] and its mapping to rich enactment environments. The work currently utilises the ICL product, marketed as ProcessWise Integrator (PW Int)[1]

[2], as the enactment system. It has a number of important properties not the least being that it supports dynamic change and that it is industrially robust. It also supports a generic approach to Process Modelling and enactment thus making it a suitable vehicle for the general study of the mapping of models to rich enactment environments.

Using the example process developed at the 6th International Software Process Workshop (hereinafter referred to as ISPW-6 [12]) to illustrate our approach, we show how an enactment system was developed to support the ISPW-6 process. The two principal technologies used to develop the model:-

- The Base Model (BM) [19] Specification Method
- The ProcessWise Integrator (PW Int) and its Process Management Language (PML)

are outlined together with a brief history of the development of the PW Int from its research predecessor the IPSE 2.5 Project [23][10].

Our motives for doing this are to illustrate our progress towards a total system and in particular we lay great emphasis on the work concerned with the validation of the BM specification against the narrative description of ISPW-6 and the consistency of the BM specification with the PW Int implementation.

In the context of a total solution our approach to modelling the ISPW-6 example is also interesting. Rather than provide a one-off solution to the problem we have taken the opportunity to develop a generic approach to the modelling of Software Development processes. Hence our solution (both the BM specification and the PML implementation) contains not only the specific solution to the ISPW-6 process but a meta-method contributing to the development of our total framework.

The chapter is organised as follows. First we describe the Base Model (BM) specification method and the Process Wise Integrator product derived from the IPSE 2.5 experimental system and now marketed by ICL. We then outline both the BM specification and the PML implementation of the ISPW-6 process, followed by their validations. Finally we conclude with some remarks on future work and conclusions.

2. BM

BM is an important component of PADM. PADM is built on the earlier Process Modelling Cookbook [11]. It is partly theoretically based, drawing on work in the fields of Socio-Technical Design, Soft Systems Methodology, and Organisation

1. ProcessWise Integrator was formerly known as Process Support System (PSS).

Theory, as well as Process Modelling, and partly based on action research through the case studies in which IPG has been involved.

2.1 Introduction

BM [19] was developed at the IPG as a specification method for modelling processes. BM provides a framework so that specifications are written in a systematic way. A temporal logic semantics [1] is given for each specification construct. This formal foundation makes it possible to reason about BM specifications mathematically.

The previous case studies of BM were concerned with using BM as a specification method to produce formal representations of processes. Such an example can be found in [20] in which a BM specification is defined to describe the co-ordination of work amongst a number of system designers working on the design of a large mainframe computer. Those case studies have shown that BM is useful in providing an abstract and yet precise description of a process. In this case study we will examine the role of BM in the *development* of an enactment model of a software process.

During this work we also developed a tool which allowed us to execute the BM specifications. The behaviour of objects is described in BM using operation patterns. A ''BM Stepper'' tool was developed using the modelling language Enact [8]. The stepper allowed a user to transcribe an operation pattern and then step through alternative sequences of operations. This ability to interact with the BM specification proved very valuable. It was used both to check the BM specification against the problem description, and also to check the BM specification against the enactment model written in PML.

2.2 Overview of BM

2.2.1 General Concepts

In BM, a system is considered to be composed of components which may be executed in parallel. Each component provides a number of operations and contains some state variables which can only be accessed by the operations. An operation may call other operations in other components. There are two kinds of operations: active and passive. An active operation of a component is started by the component itself, and a passive operation is only started when it is being called by another operation in another component (or an agent). The behaviour of a component is specified by the pattern to which its operations, when executed, conform. Such patterns are called *operation patterns*. The execution of a component is as follows: if the next allowed operation in the operation pattern is an active one, it can be executed without waiting for any external trigger; if it is a passive one, the component must wait until it is called by another operation in another component (or an agent).

A component may only communicate with another component by calling the operations of the other component.

2.2.2 Underlying Model

The semantics of BM constructs are defined using a linear time temporal logic. A brief overview of the underlying model is described below. A detailed definition is given in [19].

The behaviour of a component is defined as the set of all the possible execution sequences of the component. An execution sequence of a component is a sequential combination of its operation sequences. An operation sequence is a sequence of events. There are four kinds of events:

- `send` a request to call an operation
- `receive` a result (or end of operation indication) from the called operation
- `accept` a call request
- `return` a result (or end of operation indication) to the caller

The sequence of an active operation may contain a list of `send` and `receive` events. In addition to a list of `send` and `receive` events, the sequence of a passive operation also contains an `accept` event and a `return` event at the starting and finishing points respectively.

The model matches `sends` to `accepts` and `receives` to `returns` when composing the full specifications of a system. The behaviour of a system is defined as the set of all possible execution sequences of the system. Each step of an execution sequence of a system, one consisting of several components, is interleaved from execution sequences of the components unless there is a synchronisation between some of the components. Two components are synchronised when one is calling the other. In the case of synchronisation, the corresponding events from the two components, e.g. `send` and `accept`, must take place in parallel.

2.2.3 Basic Constructs

To specify a system, BM provides two constructs - a single object and a composed object.

A single object is used to describe the behaviour of a component by specifying:

- the behaviour of the possible (sequential) combinations of the operations, i.e. the operation pattern
- the behaviour of each operation provided by the component; the specification of an operation consists of three parts:

  call templates: specifying the interaction events between this operation and other components

  c*all pattern*: specifying the orders of these interaction events

pre- and *post-conditions*: specifying the functionality of the operation

A composed object is used to describe the behaviour of a collection of components (executing in parallel) by specifying the behaviour of each component (as described above).

3. ProcessWise Integrator (PW Int)

3.1 Introduction

ICL Processwise comprises a suite of three services designed to support organisational process activity in a number of ways. The ''Business Modeller's Workbench'' offers a tool to model and to simulate process behaviour; the ''Integrator'' is a process modelling, integration and enactment system; and the ''Guide'' describes a particular methodology designed to exploit the features of these technologies. The particular technology of immediate interest is ProcessWise Integrator [2].

This system takes the role/activity/interaction notion of defining the mechanism for causing transformation events and provides for their enactment in the same context. It provides a framework within which tools can be effectively co-ordinated to service a process representation of an information systems development methodology. This representation is in the Process Management Language (PML).

3.2 The Process Programming Language (PML)

The similarities between process programming and conventional software programming have already been remarked upon [14]. The process programming language PML is central to ProcessWise Integrator. It has evolved from the IPSE 2.5 Process Modelling Language [17][18], and still retains many of its former characteristics. Despite its origins in the concepts of knowledge representation [7] [15], practical experience has shown PML to be more amenable to imperative rather than declarative descriptions of processes. In developing the IPSE 2.5 system into PW Int, new language features to facilitate the imperative style of representation have been introduced. We can identify 6 aspects of PML which have helped its users to encode processes:

- *Concurrent threads of execution:* The threads provide a convenient abstraction that simplifies the program design task.
- *Dynamic thread creation:* The dynamic creation of new threads captures the unpredictability with which new activities can start within a process.
- *Subtyping:* The support for subtyping allows operations to be defined that can be applied to data objects of different but related types.
- *Persistence:* The persistence provided by the PW Int environment relieves the programmer of any concern as to whether program data is held in primary or secondary storage.

- *User Interface:* PML allows the user to define a simple relationship between the state of a process thread and the display its owner sees.
- *Application Interface:* The ability to start and then transfer data to and from applications that are running outside PW Int, allows the process program to co-ordinate their operation with the state of the process.

The language features that PML provides to achieve these characteristics are described below.

3.3 The Type System

PML was conceived as a class-based language, which in its initial implementation was not type-checked. It is now a strongly typed language, though it still retains a class inheritance structure. 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 act as schemas for subsequent execution thread creation. Note that, in the current context, the term ''role'' refers to an on-line role, i.e. one which is described fully in PML.

3.4 Roles: The Independent Threads of Execution

A role class definition defines the data values, and also the behaviour of its instances through a single thread of process. The data properties are defined in the resources category, and these data values are owned solely by that role. One role can only communicate with another by sending it a message. Using roles as data encapsulators both simplifies the programmer's task and makes it less error-prone [24].

The body of a role is also divided into named action properties, each guarded by a *when* expression. These properties describe the behaviour of the role. The execution of these properties is controlled by an intra-role scheduler (see Section 3.7). First it discounts all action properties which have a guard expression that evaluates to false. It then chooses one of the remainder as the next action property to execute.

When a new role instance is created, it may have initial values assigned to its resource properties by the role which starts it. This allows an initial set of communication channels to be established for the new role instance. If further channels are required, in order to enlarge the role's communication network, their values can be transmitted along the initial set of channels.

The inter-role communication is by message passing. When a communication channel is created two tokens are returned corresponding to the two ends of the channel. The channels are uni-directional and their tokens are typed by the end to which they correspond and the type of message they expect. Only the token for the giving-end can be duplicated. This allows many roles to give messages to the same taker, but ensures that there will only ever be one taker.

The communication mechanism is asynchronous. A giver action is not blocked waiting for a taker. However, a receiver may be blocked waiting for data to be sent. PML provides a notation whereby the programmer can ensure that this does not

happen. This inter-role communication mechanism also benefits from subtyping. PML provides a cast operation which allows any lost information to be retrieved. Its use implies a run time type check.

3.5 Process Evolution

PML provides three language interfaces that are important contributors to the ability of process programs to evolve. One creates new threads of execution, as discussed in Section 3.3. The second is the interface to the PML compiler. This interface accepts the source text for class definitions and turns them into compiled class values.

The third command allows the behaviour of a role instance to be re-defined in terms of a new class definition. As the data space of a role instance cannot be shared, the redefined role is allowed to retain its previous state where it is consistent with the new role definition. This command breaks the data encapsulation model, and so must be used with great care.

On creation, a PW Int system provides a single role. This role offers its user the ability, first, to compile a set of class definitions and then to apply a behaviour change, naming one of the newly compiled role classes. The user then continues to build up the process program using a combination of the three commands. These commands are the foundations on which higher level models of process evolution may be built.

3.6 Expressing Behaviour through the User Interface

We now address the mechanism that PML provides for expressing the user's perspective of behaviour by the building of the action agenda. PML offers a set of interfaces known as *UserActions*. Their presence as the first command in a role's actions property has particular significance: a role will not commit to a *UserAction* until it has been selected by the user. The action agenda is constructed automatically whenever the intra-role scheduler has no actions properties that it can schedule. The agenda will contain an entry for each *UserAction* that appears as the first command of a actions property, provided the guard expression for the property evaluates to true. Once the user chooses an action, then the sequence headed by that action is scheduled. Scheduling the *UserAction* will probably result in the creation of further UI objects.

The PW Int UI model is hierarchic. At the top level the participant's roles are collected into a role agenda. Each role opens into an action agenda. This presents the participant with the list of the user actions that are currently available for that role. The participant can choose one. The chosen action then opens to reveal a further window, the form of which depends on the action that was chosen. It may be, for example, a form to fill in or a text editor window.

Figure 1 illustrates U.I. window behaviour as the system bootstraps a simple user client server application. The participant may log out of the user interface at any time. The roles under his or her control will continue their execution, only

being suspended if they reach a state which requires user intervention for them to proceed.

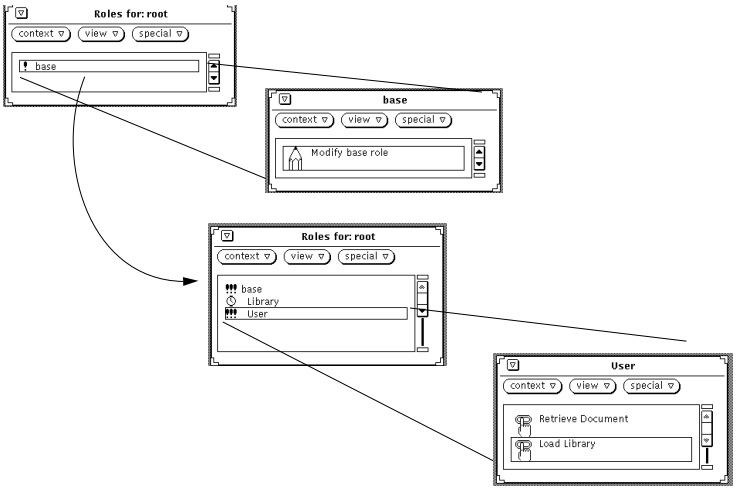

Figure 1 The User Interface

3.7 External Application Interface

PML provides a basic capability that allows the process program to control applications that are running outside the PW Int system. Communication channels are used to represent the dialogue with the application. Each application is seen as a parcel of communication channels. The form of this parcel is initially defined outside the process program. It becomes available to the process program once the application server that manages the application logs into the PW Int. The process program may then ask the server to create an instance of the parcel of communication channels, and by implication an instance of the application with which they are associated. A application server is a program written to execute in the same environment as the applications it manages. Together with the support code in the PW Int system it provides a translation service between the application and the process program.

3.8 System Architecture and Implementation

The PW Int system is composed of three architectural elements: the process engine called the Process Control Manager (PCM), the UI servers and the Application servers. An instance of the UI server is run on each participant's workstation. The act of logging in connects the server to the PCM. A high-level protocol is used

to communicate the contents of the display to the workstation and the participant's actions back to the PCM. A similar login procedure is performed by the application server. An application server executes in the environment appropriate to the applications it will support. PCM implementations exist for SUN 3, SUN 4, DRS 6000 and ICL Series 39 machines. UI servers have been written for NeWS, and X Windows[1] running on SUN workstations and Microsoft Windows running on PCs. Tool servers exist for SUN 3, SUN 4, PC and DRS 6000 environments. VME applications on Series 39 machines can be invoked through the Unix tool server. Standard communication protocols are used to connect the servers to the PCM. Figure 2 is a diagrammatic representation of the choices of PCM and server environments that are available.

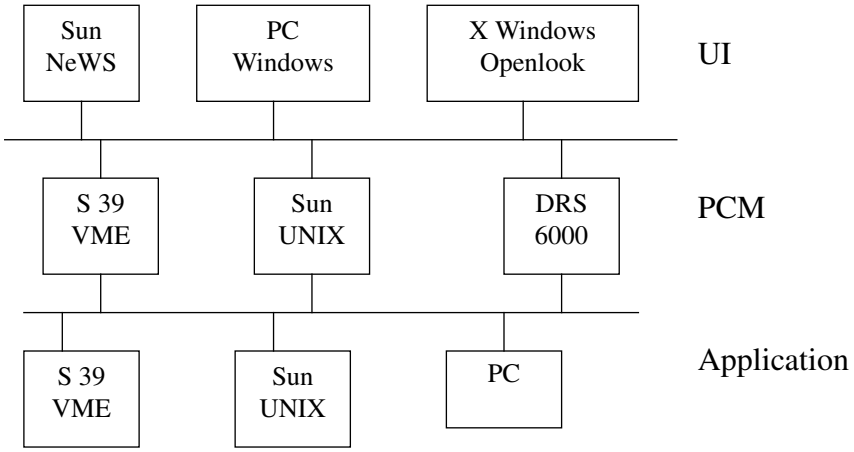

Figure 2 The PCM and Server Environments

The PCM is coded mainly in PS-algol [16], though lower-level languages are used to handle inter-machine communications. The benefits accruing from the use of PS-algol are considerable. As well as persistence, the PS-algol system provides an interface to its compiler which can be called from within a PS-algol program and which, on successful compilation, returns an executable procedure value [5]. We use this interface to provide a similar capability in PML.

Long running processes have to be closed down and restarted to allow for operations like machine maintenance. The PCM must also be resilient to machine fail-

1. X Window System is a trademark of Massachusetts Institute of Technology.

ure. PS-algol allows the implementer to provide these facilities at very low cost. The objects that compose the PCM are kept as data values in the PS-algol persistent store. Frequent commit operations ensure that they are retained in an up-to-date form. Restarting the PCM after either a deliberate or an accidental close down is merely a matter of extracting the objects from the persistent store.

4. Case Study

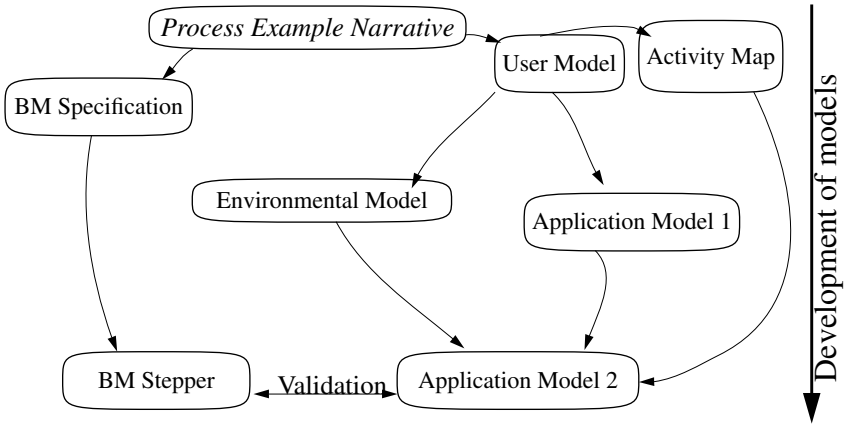

Figure 3 The Meta-Process of the Case Study

The example process developed at the 6th International Software Process Workshop is of a nominally generic process commonly found in organisations concerned with the development and maintenance of software systems. It focusses on activities associated with making changes to specific modules or units which comprise a single software system. No information is provided about participant skill levels, whether the enactment should cater for novice or expert or both, of the software system itself, or of volumes. The focus is clearly on the process itself which was described clearly and concisely in narrative form as a list of activities provided with inputs, outputs, purpose, and responsibility. This statement of requirements corresponds to Feiler and Humphrey's [6] Process Definition level of abstraction, and to Conradi et al's [3] PM2/PM3 meta-process level.

The work of obtaining the BM specification, and of developing the enactable model, is described with reference to this meta-process. Figure 3 makes clear how development of both these threads occurred concurrently and concluded with a validation test between the BM specification and the PML model definition by means

of the BM Stepper. We would expect in future that the source of a PML model would be the BM specification rather than the narrative process description which was used in this work.

The component of the meta-process dealing with BM model development is described in Section 4.1.8, page 14 and that of the PML model in Section 4.3.8, page 26. The validation of the PML model with reference to the BM model is explained in Section 5.1, page 30.

4.1 Specifying ISPW-6 in BM

4.1.1 Elements Described in the ISPW-6 Example

Agents: These are the personnel, e.g. project manager, design engineers, QA engineers.

Steps: These are the actions performed by the agents, e.g. modify design, review design. Each agent may perform several steps and a step may be performed by several agents.

Inputs/Outputs: These are the data required and produced by each step, e.g. current design, design review feedback and modified design.

- The format of Input is:

  Data(where from)(communication means)

  where from may have the following possibilities:

  file(s) and paper(s), e.g. the software design document

  agent(s), e.g. design engineer

  step(s), e.g. review design

  (In this chapter, the communication means is not modelled.)

- The format of Output is

  Data(where to)(communication means)

  where to may have the same possibilities as **where from**.

Constraints: these describe the allowed sequencing of the steps.

A general scenario is that when the constraints are fulfilled and the inputs are available, an agent takes some inputs to perform a step and produces some outputs. Each agent may perform several steps and each step may be performed by several agents.

4.1.2 BM Representation of the ISPW-6 Elements

The ISPW-6 example is modelled as a composed object. Each agent is modelled as a single object. Such an object will be referred to as an agent object in the rest of the chapter. Each step is also modelled as a single object, such an object will be

referred to as a step object in this chapter. An agent object is said to take up a particular role if it accesses a particular step object. An agent object may access several different step objects, and vice-versa, a step object may be accessed by several different agent objects. The former allows an agent to take up different roles and the latter allows team work. The whole ISPW-6 example is modelled as a composed object containing all the agent objects and the step objects.

4.1.3 **Agent Object**

An agent object may be engaged in three kinds of activities: it may deliver artifacts to some steps; (The term *artifact* is used in accordance with the definitions given in [6] and [13]. *Artifacts* mean both input and output data for steps); it may receive artifacts from some steps; it will perform some steps. The following three kinds of operations are used to model these activities:

supply operation: delivers artifacts to a step by calling the receive operation of the appropriate step object. A supply operation of an agent object may be modelled as a passive operation, i.e. being invoked by an agent, or as an active operation.

receive operation: receives artifacts from a step. These operations are called by the supply operations of the appropriate step or agent objects. Receive operations are modelled as passive operations.

step interface operation: calls the corresponding step object to perform (part of) the step. Step interface operations are invoked by the relevant agents. They are also passive operations.

In addition to the above three kinds of operations, each agent object should also have an operation pattern which defines the constraints on the ordering of the operations for that agent. There is no general common constraint for agent objects.

4.1.4 **Step Object**

In order to perform a step, a step object usually needs to get some artifacts (inputs) from either some agent objects or some other step objects or both. On completion of a step, some artifacts (outputs) may be delivered to either some agent objects or some other step objects or both. Therefore step objects also need to provide three kinds of operations:

receive operation: receives the input data from either an agent or a step object. These operations are called by the supply operations of the appropriate agent or step objects. Receive operations are passive operations.

supply operation: delivers the output data to either an agent or a step object. These operations call the relevant agent or step object. A supply operation of a step object is active because the operation is started by the step object itself when it is ready to deliver data.

step operation: performs the step. This operation is called by the relevant agent objects. Step operations are passive operations.

As in an agent object, in addition to the operations, each step object should have an operation pattern. Usually the operation pattern for a step object consists of a few receive operations, followed by, possibly an iteration of, the step operation and followed by a few supply operations.

4.1.5 **Input and Output flows**

In the ISPW-6 example, the scopes of the data are not explicitly described. In this document, we assume that files and papers are available to their step objects only, therefore they are modelled as state variables of the step objects. (Data may be modelled as objects if they are globally available.)

- Data from file(s) and paper(s):

  The data is obtained by reading from the state variable(s).

- Data from another agent or step:

  The data is obtained through the appropriate receive operation.

- Data to file(s) and paper(s):

  The data is delivered by writing to the state variable(s).

- Data to another agent or step:

  The data is delivered to the agent or step object by calling the appropriate receive operation, through the relevant supply operation, with the data as the parameter(s).

4.1.6 **Relationships Between the Different Objects**

- Each supply operation of an agent object calls a receive operation of either a step object or another agent object.
- Each step interface operation of an agent object calls the step operation of a step object.
- Each supply operation of step object calls a receive operation of either an agent object or another step object.
- The agents, i.e. the project participants, invoke the operations provided by the relevant agent objects.

4.1.7 **Effects of the Specified Operations**

In our specification, the only effect of a supply operation is to call the corresponding receive operation, and similarly the effect of a step interface operation is to call a step operation. The receive and step operations take some parameters and update

the relevant variables. The details of the operation specifications are not included in this chapter. A full BM specification of the ISPW-6 example can be found in [21].

4.1.8 The Meta-Process Component: Obtaining a BM Specification

The following is a guideline for modelling a software process using BM.

1. Identify the agent objects
2. Identify the step objects
3. For each agent object

- identify the supply operations
  one supply operation for each step to which the agent delivers artifact
- identify the receive operations
  one receive operation for each step from which artifacts are received
- identify the step interface operations
  one step interface operation for each step the agent performs
- define the operation pattern for the agent object
  In [12], the overall constraint for each agent is not explicitly given.

4. For each step object

- identify the receive operations
  one receive operation for each step or agent from which input data is received
- identify the supply operations
  one supply operation for each step or agent to which output data is delivered
- name the step operation
- define the operation pattern for the step object

4.2 BM Description of (part of) the ISPW-6 Example

In the section, the BM specifications of the Modify Code step, Modify Unit Test Plan step and the Test Unit step are described. The list below contains the abbreviations used in this sub-section.

Abbreviation	Description
SAT	Schedule and Assign Task
MD	Modify Design
MC	Modify Code
MTP	Modify Test Plan

MUTP	Modify Unit Test Package
TU	Test Unit
MP	Monitor Progress

Modify Code—*MC*

- identify the receive operations

Operation Name	Description
reqfrSAT	requirement change from *SAT*
desfrMD	design from *MD*
packcodefrTU	code from *TU*
codebackfailfrTU	disapproved feedback from *TU*
codebackokfrTU	approved feedback from *TU*

- identify the supply operations

Operation Name	Description
srctoMUTP	(modified) source code to *MUTP*
codetoTU	object code to *TU*
endMCtoMP	notification of completion of the step to *MP*

- Let *doingMC* denote the name of the step operation.
- The operation pattern for *MC*

  let

  MC1:= desfrMD; doingMC;srctoMUTP;codetoTU

  The pattern is

  reqfrSAT; MC1; (codebackfailfrTU;packcodefrTU;MC1);*
  codebackokfrTU;endMCtoMP;fin

  In the above formula, $P;Q$ denotes that P is followed by Q, $P^*;Q$ denotes that P may take place zero or more times before being followed by Q, and *fin* indicates the end.

Modify Unit Test Package—*MUTP*

- identify the receive operations

Operation Name	Description
reqfrSAT	requirement change from *SAT*

tplanfrMTP	test plan from *MTP*
desfrMD	modified design from *MD*
srcfrMC	source code from *MC*
packbackfailfrTU	disapproved feedback from *TU*
packbackokfrTU	approved feedback from *TU*

- identify the supply operations

Operation Name	**Description**
packtoTU	unit test package to *TU*
endMUTPtoMP	Notification of completion of the step to *MP*

- Let *doingMUTP* denote the name of the step operation.
- The operation pattern for *MUTP*

  Let

  MUTP1:= srcfrMC;doingMUTP;packtoTU

  The pattern is

  reqfrSAT; desfrMD;tplanfrMTP;MUTP1; (packbackfailfrTU;MUTP1)**;*
  packbackokfrTU; endMUTPtoMP; fin

Test Unit—*TU*

- identify the receive operations

Operation Name	**Description**
reqfrSAT	requirement change from *SAT*
codefrMC	object code from *MC*
packfrMUTP	unit test package from *MUTP*

- identify the supply operations

Operation Name	**Description**
codebacktoMC	feedback of code to *MC*
packbackoktoMUTP	approved feedback of test package to *MUTP*
packbackfailtoMUTP	disapproved feedback of test package to *MUTP*
*succtoM*P	notification of successful testing
endTUtoMP	Notification of completion of the step to *MP*

- Let *doingTU* denote the name of the step operation.
- The operation pattern for *TU*

  Let

TU1:= codefrMC;packfrMUTP;doingTU

The pattern is

reqfrSAT; TU1; (packbackfailtoMUTP;packcodetoMC; TU1)**;*

packbackoktoMUTP;succtoMP;endTUtoMP;fin

4.3 Implementing ISPW-6 in PML

4.3.1 Overview of model architecture

A number of writers confirm the difficulty of representing process activity in a meaningful and concise manner. Curtis [4] confirms the four perspectives widely accepted as being needed to describe a process: organisational, functional, behavioural, and informational, but it is not an easy matter to obtain clean abstractions of these perspectives with the tools currently at our disposal. Conradi et al [3] take a rather more pragmatic approach and recognise at least five sub-models:-

(1) Activity model. This is the executable model.
(2) User model. Describing the service provided to the process participants by the system, and the corresponding obligations placed on them.
(3) Organisation model. The model of organisation role structure and functional relationships which associate these roles.
(4) Product model. The data structure and relationships.
(5) Tool model. Description of tool architecture.

This latter model was considered more appropriate as a framework in which to describe the different facets of the subject process. The models needed were not exactly as postulated in [3]: the nature of the process is such that there was no need for a tool model, and, because of the focus on enactment, there was a distinct need for a model of the work infrastructure or environment appropriate for participants; this was called the Environment model.

4.3.2 Methods and Tools

The work described herein does not fall neatly into the phases of the PM1-5 meta-model [3], as the focus of the ISPW-6 example is on the implementation of a model already fairly well-defined in narrative form, with adequate information to begin the PM3 Customisation phase. PADM (Process Analysis and Design Methodology [27]) provides a coherent strategy and techniques for much of the area of PM1 through to PM4, and indeed outside of this framework, identifying the need for a ‘‘baseline’’ model, stressing that once a ‘‘target’’ process (or even processes) has or have been identified, the implementation route will of necessity be from ‘‘what is’’ to ‘‘what ought to be’’. The ISPW-6 example assumes a ‘‘green field’’ implementation, i.e. there is no need to take into account current work practice and cultural influence.

Certain of the representation methods advocated by PADM were also used in this study: the first using the role-interaction paradigm of process [9] to identify user roles (i.e. sociological roles within the organisation, commonly aligned to job descriptions), to map tasks or activities to these roles, and to identify at a coarse grained level discrete states and associated events of the process. The resulting diagrams are known as RADs [15]. The second was the use of the activity diagram (AD) to map activities independently of roles [1]. There is another type of diagram in PADM known as dependency (DD) diagram which provides a high level abstraction of the essence of complex processes. This was not required in this simple example. The Methodology does, as yet, provide no guidance for implementation or for selection of support platforms.

4.3.3 **The User Model**

In this study the user model took two forms. The first was a RAD model of roles, tasks and artifacts shown in Figure 4, and this was used as a basis for the application model described in Section 4.3.5 “The activity or application model” on page 21. It provided a very concise representation of the process. The greater detail needed to structure an implementation was provided by the second form which was much more fine-grained, and comprised narrative lists of user activities which might be ascribed to these roles. For example a designer would want to be notified of an impending task, to be made aware that documents were ready and available for use, to be able to work possibly at intermittent intervals, to dispatch the work item and finally to be notified if re-work was necessary or if the item had received approval. We would assume that the task would not be complete until the design had been approved in review, so the facility to iterate would be necessary.

Role Activity Diagrams define the scope of role behaviour. When considering organisation roles, they indicate the different activities which can take place and the dependencies which exist between them. The meaning of the diagram can be clarified by taking the example of the organisation role of Design Engineer:-

Once notified of a task, the Design Enginer can work on either Modify Design, or Modify Source Code. Once the design has been modified and passed to Design Review, the activity goes into a ‘‘waiting’’ state (identified by the tail). If the design is approved, the activity Modify Design is completed and ceases to exist. If Code Feedback arrives, the activity can be re-invoked by the Design Engineer to address the feedback issues. Another kind of dependency is illustrated at completion of Modify Source Code. Here, the Source Code cannot be despatched to QA Engineer until the Approved Design is available.

---

1. A PC modelling tool RADitor is available for role-activity diagrams (developed by Co-Ordination Systems), and the activity diagramming tool is a component of Process Modeller’s Workbench being developed by the Informatics Process Group.

Organisation Roles, Activities, and Interactions

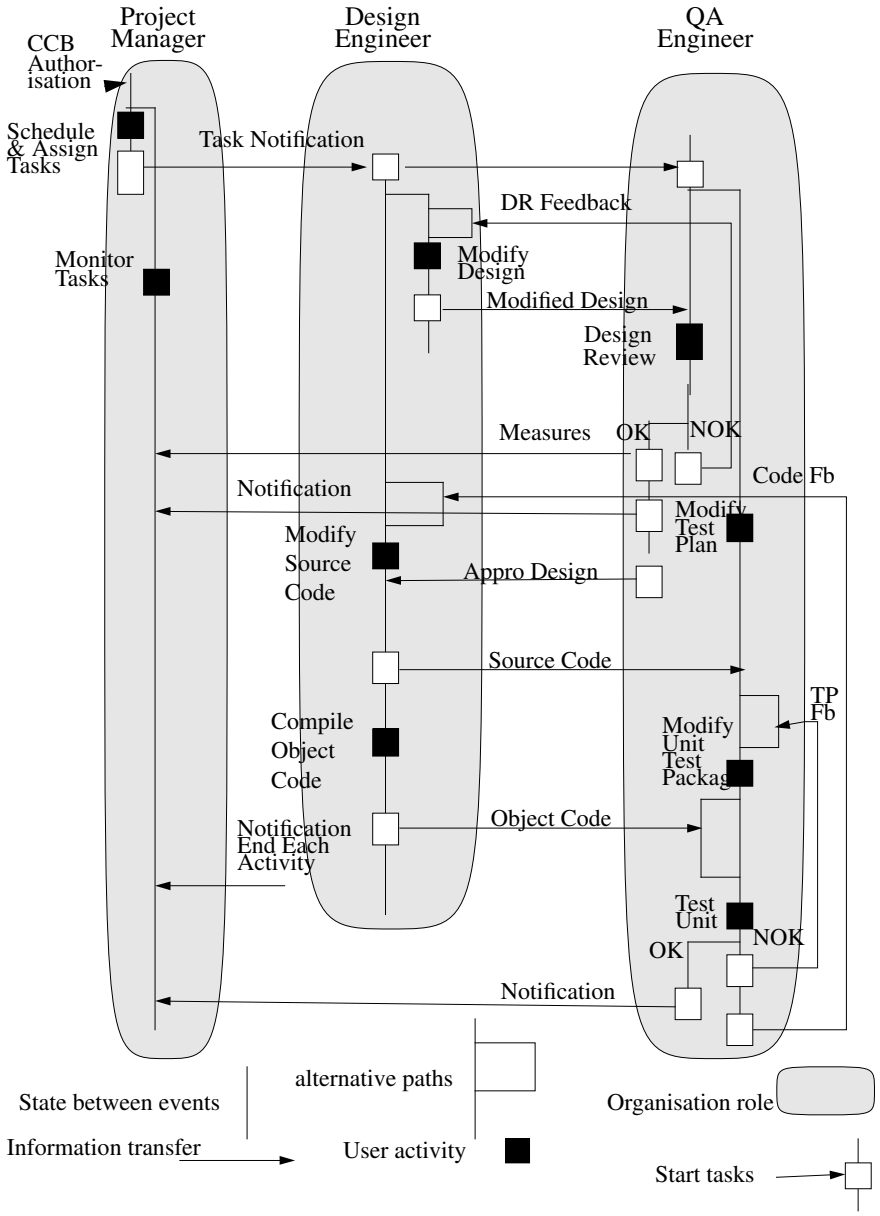

Figure 4 The User Model

The only significant departure from the example process is that joint responsibilites have been re-interpreted into unitary responsibilites. There is, as yet, no mechanism for interpreting joint responsibility in terms Processwise Integrator technology.

It should be emphasised from the outset that we are dealing with two distinct types of ‘‘role’’. There is the organisation role referred to here, which can be conveniently referred to as the user role; and the unique definition of role in ProcessWise (single threaded, defining scope of entities, forming the basis of the user interface, unit of change) which can be called a system role, or in PW Int terminology, an on-line role.

An example of the activity list is provided in the behavioural description of the Application Model for the Modify Code task on page 24.

4.3.4 The Environment Model

The process example was well defined, however to make the enactment meaningful in the context of an organisation some attention had to be paid to the support or infrastructure necessary for the process to be enacted. This helped ensure a reasonably realistic implementation was available to participants, and to try to throw light on some of the issues which could arise in a real-world implementation.

It was found helpful to encapsulate certain ‘‘housekeeping’’ activities into system roles which were permanently bound to participants . These roles supported service activity rather than task activity and were referred to as Admin roles. They are discussed in Section 4.3.8 “The meta-process component: developing the application model in PML” on page 27.

The environmental aspect which was selected for inclusion in this implementation was that of the library service. It provided a more elegant and appropriate solution to information handling and control than provided for in the process example. It also illustrates the use of a system role in both a client-server architecture and as a passive component making an effective and essential contribution to the process.

To be effective, a process enactment system would have to interact with the library facilities used to hold these data items. This is shown in Figure 5 , where the communication links between the user system roles and the library service role are illustrated and the activities associated with each of these roles is described. Only those activities associated with the library have in fact been modelled: other activities are represented but not supported in the model.

4.3.5 The Activity or Application Model

The use of the term ‘‘application’’ is preferred in the context of the executable model as it clearly refers to a system model, whereas ‘‘activity’’ might be either real-world activity, or system activity and can also get confused with activity diagrams which are useful for structuring real-world activities.

This is the model which is ultimately enacted, and in its final form is expressed in the process definition language[1] PML (Process Management Language) for exe-

cution in ProcessWise Integrator system. The development of the model, of what is in essence a process management system, was driven by the nature of PML and the properties of the support system.

Of paramount importance in PW Int is the need to identify from the outset the appropriate system roles and granularity appropriate for the application. It is not impossible (but highly unlikely) that system roles could map directly on to user roles, and at the other end of the spectrum a role might represent but a single activity. However the most likely situation is that the behaviour will somewhere between these two extremes. Key considerations in defining system roles are:-

- Need for run-time concurrency.
- Scope of data entities.
- Requirements of user interface.
- Desired scope of change.

The user model has been transformed to reflect these criteria and this is shown as the Application Model in Figure 6 . There is a small extension to the Requirements to better illustrate the receiving of a modified design. Review system roles are provided to enable users to participate in the review process prior to a review meeting at which decisions might be made. On the other hand, CCB was not modelled as a system role because it increased complexity without contributing to understanding. The control process for the application model, really a component of the meta-process itself, is shown in Figure 7 and discussed in Section 4.3.8.

These diagrams formed the basis of describing the process in PML. The system roles were constructed of three basic overlays utilising the inheritance properties of the language. The essential behaviour of *all* task roles is defined in *WPRole*: -

- to maintain and display when required certain task information (identified as *a_ScheduleDates* below, information is bound on instantiation and subsequently changed by a Re-scheduling action);
- to provide a facility for displaying a message when and if the task was changed by higher authority such as the PM (identified by *displayMessage* changed by a Hold/Restart/Cancel action);
- to dispatch a message to PM on conclusion of the task (*sendIt*).

The PML for this behaviour is illustrated over by the definition of *WPRole* as a sub-class of the pre-defined PML role class *Role*.

---

1. The term ‘‘process definition language’’ is possibly more appropriate than ‘‘process modelling language’’, in that it reflects more accurately the current departure from the original ideals of the language.

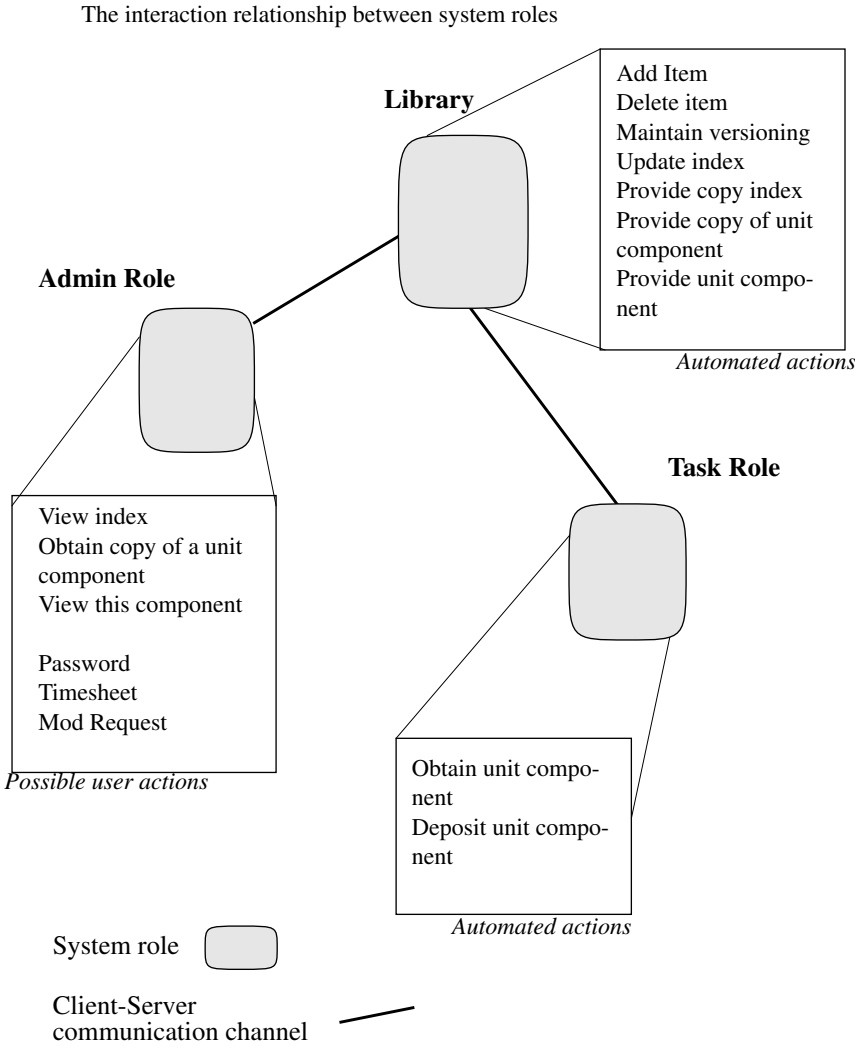

Figure 5 The Environment Model

The interaction relationship between system task roles

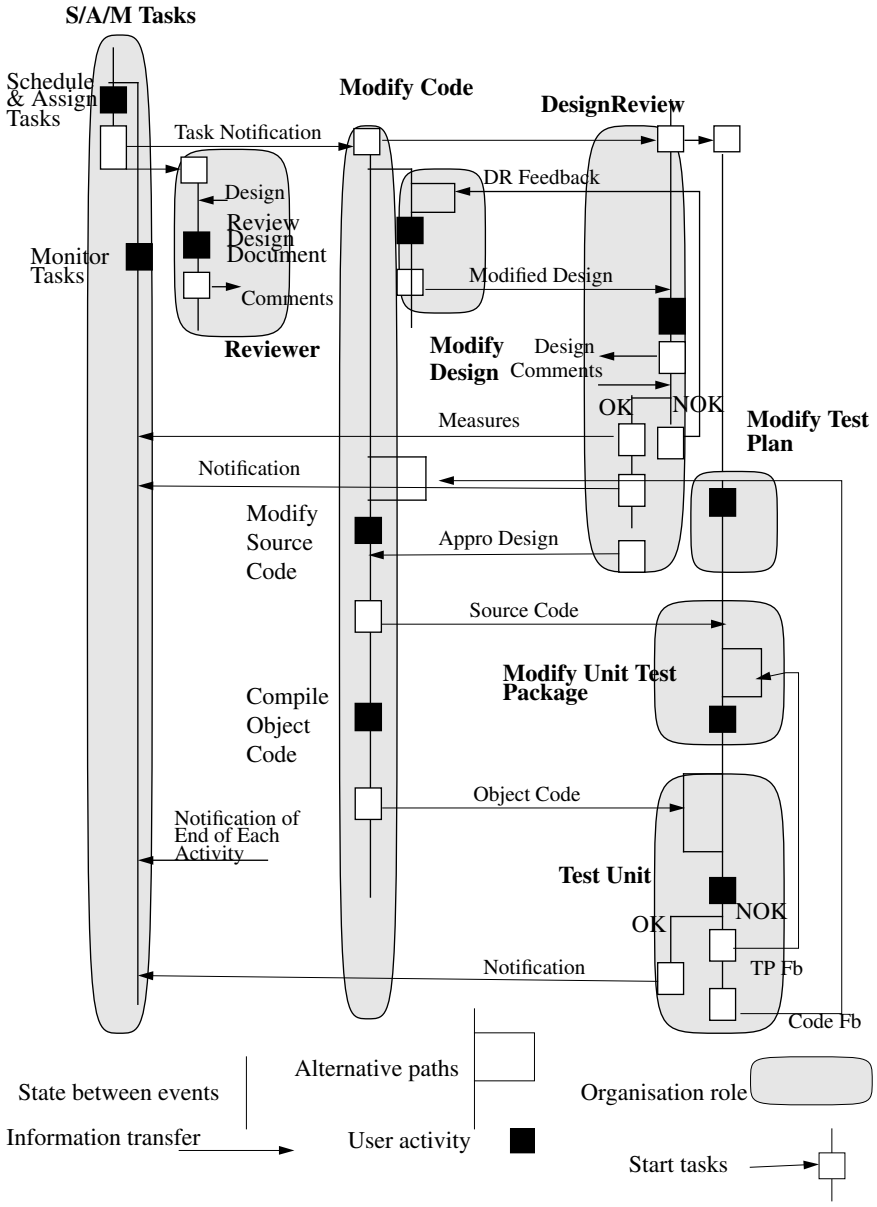

Figure 6 The Application Model 1

```
WPRole isa Role with
classes
assocs
        doneGP   :giveport DonEnt
resources
        wpEnt   : WPEnt
        thisRole:String
        whoiam  : String := ''
        doneEnt : DonEnt
        kill    : Bool := false
        schedFormat : collof UIField:=collof UIField(
           UIField(name='startDate', label='Schedule Start'),
               UIField(name='duration', label='Duration'),
           UIText(name='requirements', label='Requirements'),
               UIField(name='notes', label='Note'))
        seeMessage     : Bool := false
        onHold         : Bool := false
       messageString  : String :='New Scheduling Information '
actions
        a_scheduleDates:{
                UserAction(agendaLabel='Task Information');
                ModifyNow
                       (agendaLabel='Task Information',
                       object = wpEnt, readOnly=true,
                       format=schedFormat)
                        }
        displayMessage:{
              ViewResourceNow
              (agendaLabel='  Message', object=messageString,
              label=whoiam);
              seeMessage:=false
                       }
        when seeMessage
        sendIt:{
                Assign(from=whoiam, to=doneEnt.task);
                Give(interaction=doneGP, gram = doneEnt);
                kill:=true
                }
       when nonnil doneEnt
termconds  kill
end with
```

The specialisations appropriate for each particular kind of task were then defined as sub-classes: for example the Modify Code task role (*MSCRole*) is a sub-class of *WPRole* which incorporates the following behaviour:-

- Create an object *scEnt* to represent source code.

- Allow modification *scEnt* using a text editor.
- Allow dispatch of a copy of this code to Modify Unit Test Package role only after the approved modified design has been received by Modify Code role from Review Design.
- Simulate compilation of the code in *scEnt* into object code.
- Allow dispatch of this object code to Test Unit task role.
- Provide for handling of feedback from Test Unit task role.
- Provide status messages.

Finally, the definitions of the library service client were instantiated as a sub-class of *MSCRole*, called *MSCLibRole*.

The implementation of the library facility shows how a network of roles and interactions can be constructed to work in a client server manner. The library is the server. It understands a number of library requests which are sent to it using an interaction. All roles which use the library can give requests to this interaction, only the library will read them. Each role which uses the library is a client. When it sends a request to the library it includes the giveport of the interaction which the library will use for the reply. The client role will subsequently receive the library's reply on the interaction's takeport.

With this well defined interface, the system roles which use the library do not need to know any details of its implementation. They only have an interaction giveport to which they give library requests. Indeed they cannot tell if the library is implemented by one role, a collection of roles, or one, or more, roles interfaced to some external tools. The current implementation has the library as a single role which handles requests one at a time.

The data held in the library is identified by its unit name, a part within that unit, ''design'', ''source code'', and ''unit test pack'', and where relevant a version number. The library using roles are divided into two types: browsing and updating. A library browsing role can get a copy of any item in the library, but cannot do any updating. A library updating role is able to retrieve items from the library and subsequently send back the item to be stored in the library as a new version.

In this example the admin roles, *PM Admin* and *SWE Admin* have library browsing facilities. The task roles are library updating roles. However, they are specialised library updating roles in that they only retrieve and update one item. For example *MUTP* role which has been created for a change on unit ''scanner'' will retrieve and update only the ''unit test pack'' part of unit ''scanner''. Figure 5 gives an overview of this architecture.

In the current version of the system, the task roles will obtain the relevant item from the library during an initialisation stage, and will replace the new version just prior to the role terminating. This could be changed if it was decided that the latest

working versions of components should be available to others through their Admin roles.

4.3.6 The Organisation Model

This was a simple reporting hierarchy with a single manager having responsibility for and control over a group of software engineering staff who could be identified as being primarily either design engineers or QA engineers. It clearly indicates that as the process spans only a single group of users, the PM is responsible for controlling both the working of the process and the evolution of the process. The other significant concern in the organisation, the mapping of role sets [22] of group responsibilities, such as the Design Review team, to the organisation structure, was not addressed. Delegation issues were not present in the process example.

4.3.7 The Product Model

As product is not the primary concern in this study, this model is trivial. The process example only considers a single system which is comprised of unidentified modules or units each of which is composed of design, object code, source code, test plans, unit test packages under version control. This control was modelled in the library service.

Notwithstanding this simplistic situation, we are aware that the nature of a product may well have an important influence on its associated process model. The significance to future process modelling concerns is further referenced in Section 7 “Conclusion” on page 38 .

4.3.8 The Meta-Process Component: Developing the Application Model in PML

This section describes the evolution of the application from narrative requirements through to implementation and evolution. To facilitate this phase, the narrative description was transformed at a slightly higher level of abstraction into two forms: one a representation of activities in an Activity Diagram, the other to a RAD to allow a more succinct description of the overall process (on a single A4 sheet). The Activity Diagram had elsewhere been found to be a useful tool to map activity independently of role responsibility, but in this more simple case its use was as a quick reference to the requirements. The Role Activity Diagram initially incorporated knowledge of the support system, however this inclusion was subsequently judged to be an unwarranted complexity and was removed. To all intents and purposes, this initial phase was the preparation of the User Model . It can be related to other significant milestones in the development through Figure 3 “The Meta-Process of the Case Study” on page 17.

When considering the needs of a potential software engineer user it quickly became clear that it would be necessary to include in the implementation certain further behaviour not referred to in the ISPW-6 example description; activities which would be required in any proposed enactment to allow participants to function effectively in their organisation and system environment. Such activities were

listed and initially mapped to user roles, those identified in the User Model. These were in effect the requirements for the environment, the Environment Model.

At this point the next major transformation was to scope the identified activities into system roles, to identify precisely what would be required from the support system. Apart from PW Int considerations identified, the needs of the ''participants'' had to be considered, particularly how a participant might wish to navigate among and through activities, and how a participant's organisational responsibilities might be expressed through the user interface. It appeared to be a very subjective consideration and in this development a principle of ''Least Effort'' was adopted, i.e. make the work of finding a work item by navigating among activities as easy as possible. In the application only one interpretation has been included; ideally a participant ought to be able to structure activities at will (constrained only by high-level process and system considerations). It became clear that was there were two distinct types of activity: those which ought to be permanently available for the participant, and those which were contextual and which ought to have transient existence. These former were scoped into Service system roles, labelled Admin, the latter into Task system roles. The mapping of these system roles to user roles is the first (diagrammatic) phase of the Application Model 1 as a RAD.

The correct establishment of interactions between system roles is very critical to the success of a PW Int application, and this need drove subsequent work. To ensure smooth development an Interaction Map was prepared showing precisely the nature and function of the minimum necessary set of interactions, and when they had to exist. This was used to write the first phase of the application model; to allow instantiation of the task roles, comprising absolute minimum of functionality, in correct sequence and with correct links in place.

Subsequent development of the Application Model 1 was concerned with specialising these class definitions incorporating the information contained in the Organisation, Product and User models, and in extending the fine grained functionality of the task roles. There is wide freedom in PML to define the guards for actions to implement behaviour via the intra-role scheduler (part of the PCM), however this freedom means that guards become very complex very quickly, and even in this simple model it was not easy to implement desired changes to some behaviours.

One portion of the meta-process was implemented in PW Int as the first specialisation of the system *Base* role. This provides a number of functions: to bootstrap the application, to provide a harness within which the application can be executed, to provide services that can usefully exist at a higher level than the application itself, and, most importantly, to provide that portion of the feedback mechanism which allows external change of the application model. One can imagine the cycle of monitoring of process performance, taking a view on the changes needed to effect improvements, and effecting such improvements by dynamically changing

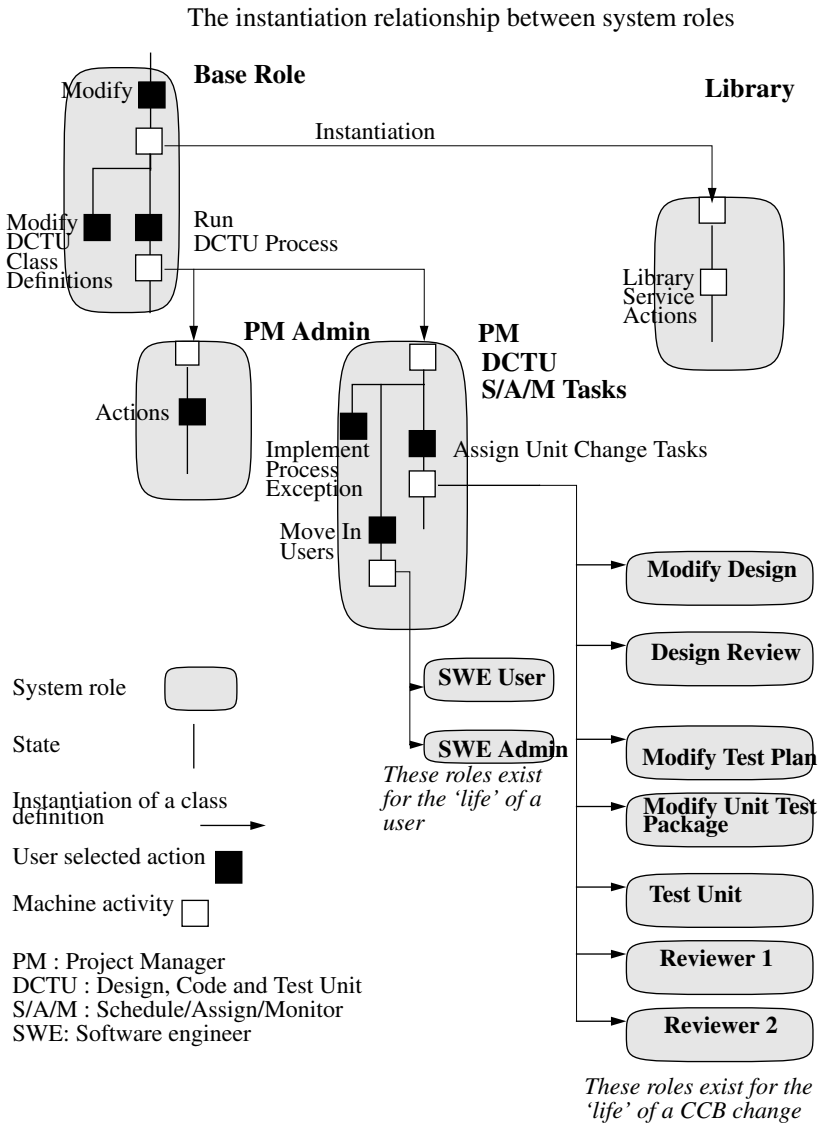

Figure 7 The Application Model 2

either class definitions, enacting instantiations, or both. This process component was modelled as Figure 7 “Application Model 2” on page 29.

The development of the *Library* service role was a significant effort done in parallel with the task role development in the application model. This service was implemented as an overlay on the basic application model: the library class definitions were introduced and task roles specialised to incorporate functionality allowing access to the library service.

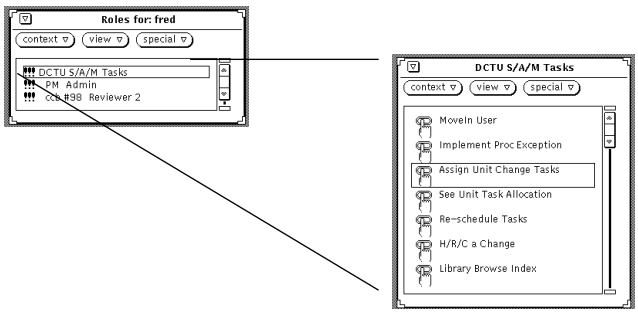

Figure 8 The UI Representation

4.3.9 **Enacting the process**

There are three separate contexts for instantiation of the different components of this implementation. All of them are “dynamic” [6]. The first comprises the Application Models 1 and 2, loaded into the basic PW Int persistent store containing initially an instance of class *Base* role. After compilation, and during the execution of the process itself, a userAgent system role and Admin system role (either *PM Admin* or *SWE Admin*) are created for each participant brought within the scope of the system. To enact the process example proper, an assignment of a particular CCB change (through invocation of the action Assign Unit Change Tasks, see Figure 8) creates specific instances of task roles bound to selected participants.

In the PW Int support system, all implementations use a specific persistent store containing an initial specialisation of system role *Role* as *Base* incorporating functionality to change itself. This facility is used to load the code for both Application Models. At the conclusion of this loading, the library service is started automatically and an action in *Base* offers the action “Run DCTU Process” and, on its selection, a prompt requests the user name of PM and proceeds to instantiate one

DCTU S/A/M Tasks role (Schedule/Assign/Monitor), and one *PMAdmin* role, both bound to PM's user name. Figure 8 illustrates a typical role agenda for a PM whose username is ''fred'', and an action agenda for *DCTU S/A/M Tasks*. In the example shown, it can be seen that fred is also the second reviewer for change reference **ccb # 98**. This bootstrap activity can be followed quite clearly on the role activity diagram Figure 7.

An essential precursor to the demonstration is for PM to ''Move In'' a number of users, identified as QAE or DE. The ISPW-6 process enactment starts by PM selecting ''Assign Unit Change Tasks''. This allows the identification of the unit to be changed, and the choice of DE, QAE, and Reviewers for that particular change.

The implemented process was enacted using ProcessWise Integrator 1.1, the Process Control Manager component sited sometimes in Unix on a SUN server, at other times in ICL VME [25] on an Estriel mainframe located at Manchester. The user interfaces ran on SUN workstations or PCs, usually at Manchester but not infrequently at Southampton, Bristol, and Durham using the X400 protocol on the JANET network.

The system supporting the ISPW-6 example process normally runs continuously, being stopped only once per week at a convenient time for a brief garbage collection. Thus it is always available for demonstration with access from Manchester or from elsewhere.

5. Lessons Learned

5.1 Validation of the BM Specification against the Narrative Description

The BM specification of the ISPW6 problem is in essence a re-writing of the English text description in the BM specification language. Because English text is often ambiguous, this will often involve making decisions about what the textual description means, which leads to a problem. How do we communicate those decisions? One answer is for everyone to read BM fluently. The method we adopted was to develop a stepper for the BM subset used in this problem. (This stepper was developed in the object-oriented executable specification language "Enact" [8]. Our BM description then became an executable specification).

One of the problems of the English textual description and the BM spec is that they are static. It is difficult to understand the dynamic behaviour of the system which they imply. Our aim in developing the stepper is to allow a user to interact with the specification. An example of this interaction is given in table 1 . At each point the user is given the set of operations which the system is prepared to engage in next. The user chooses one operation from this set and the system returns with the set of next operations which are now available following the one chosen. By stepping through the model in this way, the user gets a better understanding of the loops and choice points in the process which form the essence of its dynamic behaviour.

This use of the BM stepper did identify errors in the BM specification. These were mostly typing errors in operation patterns, or simple omissions. Others were more serious discrepancies between objects. For example, in one version of the BM spec we found that MUTP was expecting a new test plan on every iteration around the testing loop, while MTP only sent the test plan once. Re-checking with the textual description it was agreed that MTP was correct. These problems could be identified by careful reading of the BM specification, but the stepper provides a more enjoyable method. It is quite easy to see from the BM that there is an iteration in MUTP and MC if testing indicates the need for revision. What is not so obvious is that during this revision MUTP must wait for the new source code to be available. This is easy to spot using the stepper.

Overall the BM stepper has proved a useful way of helping to compare the BM specification with the English textual description. It helped to identify errors in the early versions of the BM. However, we suggest that its main advantage has been in making the BM more accessible to novice ''BMers''. By stepping through an executable specification, users gain more understanding of what the BM says about the dynamic execution of the system.

5.2 **Developing the Implementation**

5.2.1 **Scheduling**

This is of significant concern for a typical manager and careful thought was given to the extent to which it ought to be supported in the implementation. ''Scheduling'' means the preparation of a plan to accomplish a piece of work in a certain time period. In order that this can be done, the manager needs to know the sequence in which discrete activities can occur, whether or not concurrency can be exploited, other constraints on the activities, and, lastly, what human and machine and other resources (such as office space) are available.

Resources were not available to provide any degree of detailed implementation to support this kind of managerial activity. Many excellent software packages exist which can do this and the facility of ProcessWise Integrator to integrate a tool would be the normal way of incorporating this behaviour. The place for PW Int in this context is in the linking of knowledge of current activity state through ''Monitoring'' to ''Scheduling'' and ''Rescheduling''. In a normal design environment, a common obligation placed on designers is to complete a daily time sheet as an indication of how their time was allocated between workpackages. It would seem sensible to allow this information to be collated automatically and used for again automatic updating of records of work done and work remaining to be done in a scheduling tool.

PW Int provides the necessary envelope within which this kind of activity could be sensibly provided. Other, task related, information can be abstracted, e.g. when activities have been commenced, but it is not easy to abstract this information from the system in a sensible and reliable way (the system has no understanding of the

data), and recourse would likely be made to participant-provided ‘‘time sheets’’. The manager then has the current information needed: how much has been done; what has finished; when current tasks likely to finish; what should have started and hasn’t.

5.2.2 User interface

It is necessary to visualise just how a user might want to navigate among tasks and services available. With PW Int as a support enactment system we have the opportunity to ‘‘parcel’’ the behaviour with an entity and refer to it as a system role, or to provide a system role with appropriate ‘‘tools’’ for holding and for manipulating entities, or some combination of the two. Criteria for choice would include just how routine the activity is, what to see if there is no work to do, volume of products, and preference of users.

Certain activities were context-free, for example certain library access functions; others, such as tasks, were very context-dependent. Relevant here for example was what the screen ought to display when there were no process tasks to perform.

In addition, of course the constraints of the PW Int user interface restrict opportunity. Screen representation can be very much an area of individual preference and ideally therefore ought to have defaults accessible by the user. However as the architecture of the application model is heavily dependent on this choice, certain decisions had to be made and closed at an early stage. One of the more obvious examples of this is the location of activities. It may not be at all clear to the process designer which role is the most appropriate for a particular activity, but the choice cannot be left to the user because it may not be an easy matter to move an action subsequently between system roles, particularly if performance is an issue.

5.2.3 Process control

The organisation chart defines who controls which roles. Thus it is clear that PM controls the activities of Software engineers. What does this entail? Essentially it is the ability to allocate resources and to assign tasks, and the responsibility to ensure tasks are completed according to schedule. However it also includes the ability to make changes in the process. The essence of improved process performance is a clear identification of the mechanism whereby the process can be changed. This implies the existence of another ‘‘orthogonal’’ process: not concerned with the day-to-day activities or performance, but the performance of the process as a whole. A process component has been written in PML called the Process Model for Management Support [26]. It provides a context for delegating objectives, planning, implementing, monitoring and re-planning process models and will be incorporated into the implementation in future.

A well-known feature of PW Int is the ability to make change ‘‘on the fly’’. This is thus far modelled in this example process in only one way: to modify an existing instance of Design Review to bypass reviewers. It is accomplished by the

PM selecting the appropriate action which offers a choice of CCB Changes to target, to select the appropriate modification, and to identify the target system role in which the modification is to be implemented. Other aspects of the ISPW-7 "Change" extensions will eventually be incorporated.

5.2.4 Guidance and enforcement

The paradigm of PW Int is essentially one of prescription and the work available is defined in the system and the facilities available are prescribed in the system. However, just how much freedom participants are given is ultimately the choice of the process owner.

This will be an ongoing debate and we have yet to understand if one way is better than another, or under what circumstances one is better than another. Is automating of procedures adequate? Taylorism co-ordinated the work of craftsmen and re-structured the activities to improve productivity. It is not yet possible to co-ordinate the intellectual effort of knowledge workers in the same way.

5.2.5 Co-operative work

PW Int does not at present support multi-user access to system roles and thus there is no means of representing joint responsibility. Group responsibility can only be represented by decomposing the process into more elementary activities which can be assigned as roles to single users in a 1:1 relationship. In this model such difficulties were experienced in "Review Design" and "Test Unit", which are both joint responsibilities. To support the notion of joint responsibility a separate role would have to be created to allow identification of decisions and concurrence with or rejection of them, together with a framework for negotiation.

It is not yet possible to adequately represent the mapping of roles in organisations between groups and individuals. Singh [22] has recently made progress and uses the term role sets to represent group responsibility however they are difficult to manipulate and do not yet adequately cope with the dynamics of responsibility transfer, delegation and subsumption.

5.2.6 Knowledge of state

Difficulty was found in communicating to the participants knowledge of current system state, in the sense "what can happen next and why isn't it happening". A device was adopted through the use of dummy actions to convey this information, but, in addition, a suitable mailserver to support communication in the context of the process would be useful.

5.2.7 Meta-model and Methodology

The work of developing these models has made a small contribution to our knowledge of how to go about implementing a particular (simple!) process description. Although PADM is primarily concerned with mapping and interpreting real-world processes, it played a part in addressing many user-centred issues, particularly

those around choice of a suitable system roles and working environment. In addition the diagramming tools were useful.

It must be borne in mind that this example process is in many ways quite unlike real-world processes in the clear definition provided, the absence of cultural influences and absence of any need to consider the transition to the supported process. It also may represent only a tiny portion of the spectrum of process activity which might be considered for process support.

The ISPW-6 example is coarse-grained and represents a typical description of ''what ought to be'' rather than ''what is''. There is no reference in the PM1 stage of the meta-process to establishing the ''what is'' process as a basis for planning the transition to the ''what ought to be'' process, but, as White mentions [27], in the real world this transition may well be the biggest problem to be dealt with.

This issue is very pertinent when considering the impacts of context and skill. The activities that are performed and the way that they are performed are frequently heavily dependent on the nature of the information to be handled, and on the skill and experience of the individual participants.

5.3 **Validating the Implementation**

During our work on this problem the BM specification and the ProcessWise Integrator implementation in PML were developed concurrently. The prime motivation behind the BM stepper (described in Section 5.1) was to help ensure that the BM specification and PW Int implementation were consistent. For any scenario that we can step through, the same scenario should be supported by the implementation. In particular the implementation should show the same iterative loops and choice points which are identified by the stepper.

Most of the operations identified in the BM specification are explicitly confirmed by a user. This means that there are action agenda entries which we can associate with these operations. (The most notable exception are the receive operations: normally there will be an action agenda entry for "send X to Y", but the receive will be done without additional user confirmation.) Thus our validation is based on making the following associations:

BM Stepper	**PW Int Implementation**
BM operation	action agenda entry
next set	agenda entries visible
step event	selecting entry from action agenda

Given this we can interact with both the BM stepper and enactment system in unison and check that everything remains consistent. An example is shown in Table 2. The action agendas which this refers to are shown in Figure 9.

There is an obvious difference between the global view which is given by the BM stepper and the user interface, organised into role and action agendas, of the enactment system. In general, the enactment system has more detail. There are also differences in the way that things have been coded. In the enactment system, as soon as the test results have been entered the code and test package feedback are sent to the MC and MUTP roles. In the BM specification there is a sequence of operation calls.

The process of concurrently stepping the BM model and interacting with the enactment system did identify discrepancies. Some discrepancies do not indicate flaws in either the BM specification or the PW Int implementation. Rather they are the result of differences in the models. Stepping through the BM model is based on observing the process and so all the operations are visible. In the enactment system, operations which do not require confirmation by the user are not reflected in the user interface. However, in supporting the process participant, the enactment system has some details which are ignored by the BM model. One example is the viewing action (labelled ‘‘Modified Design’’ in the ‘‘Modify Code’’ action agenda in Figure 9.) This gives the user a chance to view the design which they are currently coding.

The concurrent style of this development meant that the discrepancies identified by this exercise were points for resolution rather than definite errors. The resolution could be :-

- revision of the PML enacted
- revision of the BM operation pattern
- an increased understanding of why there was an apparent discrepancy even though both the PML and the BM were correct.

One of the greatest strengths of this exercise has been the greater formality of testing. Given a system with several roles each with associated action agendas, it is difficult to devise a check of all the alternative paths through the system. However, given an example stepping of the BM specification we can attempt the same sequence of operations in the enactment system. Because there is not a 100% one-to-one correspondence this is not ideal. However it is the best we have at the moment. We have a better grasp of how effectively our testing has covered the different paths, and tests are more reproducible when changes have been made.

next set	chosen	action agenda	chosen entry
doingMUTP	*	Modifying UTP(1)	Modify UT Package
codeMCtoTU		Modify Code	
codeMCtoTU	*	Modify Code	Dispatch Object Code
packMUTPtoTU	*	Modifying UTP(2)	Dispatch UT package
doingTU	*	Testing Unit(3)	Prepare Result Report
packbfTUtoMUTP	*	Test Result	...
packboTUtoMUTP		Test Result	...

Table 1: BM stepper and PW Int

6. Future Work

The main focus of this work has not been to provide a critical evaluation of the BM and ProcessWise Integrator technology, but rather to understand the problems in building effective process support systems. The main thread of future work is therefore in trying to formalise what we have done into a more methodological approach. One approach to this is to expand the example tackled; in particular understanding how the design, code and test process might be affected by the product being changed. Another approach is abstract this work to give a technique which can be applied to other problems. Previous experience has shown that any methods to develop appropriate support systems will have to assist in understanding what the desired process is. The precise description of the ISPW-6 problem means that this has not been an issue in this work. Finally, there is future work in developing the technology. In this exercise, the BM stepper proved useful, and therefore worthy of further development and evaluation. In addition, PW Int will evolve as more is learned about the features needed to develop effective support systems. The breadth of possible support technologies: process based environ-

System Role Action Agendas

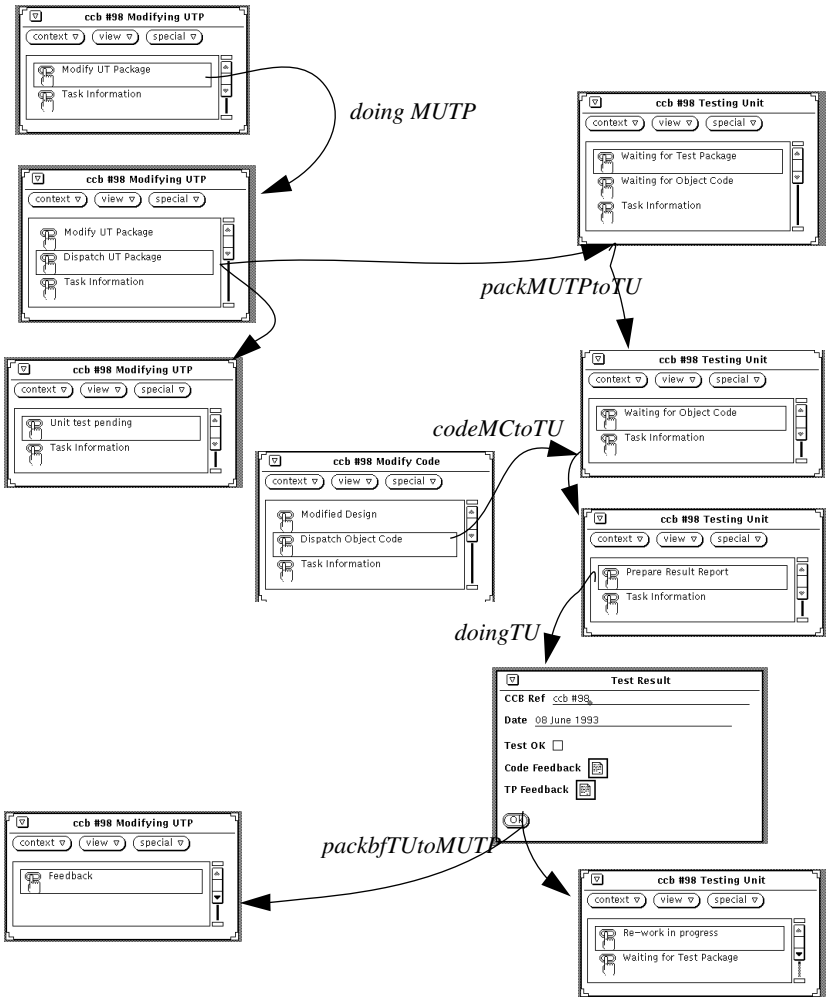

Figure 9 Mapping of BM Stepper to PW Int Action Agendas

ments, groupware and CSCW systems, means that, to be of general applicability, a methodology will have to be independent of any specific support technology.

6.1 ISPW-7

It is intended that development of the implementation described herein should continue to ultimately fully address the issues raised in the extension to ISPW-6 when resources permit. We feel that the instantiation of a coherent ‘‘managing’’ process, to support planning, implementation, monitoring and re-planning, would significantly extend the scope and interest of the example. One concern is that, as a demonstrator, the full process example is quite complex and we anticipate different versions of the implementation to focus on different specific issues.

6.2 Context

There is concern in software engineering circles that context, or the products being developed or being produced, may well have a profound effect on the process under which they are produced. A SERC funded project ‘‘Evolution of Large Software Systems’’, a collaboration between Manchester and Southampton Universities, commenced in late 1993,. This is investigating the means to control the evolution of systems within the context of a system architecture definition and architectural process.

6.3 Methodology

The work bearing fruit in PADM [27] will continue in a proposed SERC funded project Process Engineering Framework of which the Methodology component will complement and extend earlier work under the IOPT project. Of particular interest will be the development of process theory, process engineering practice and techniques. It is of particular interest for us to understand the extent to which the ISPW-6 process, having been designed by a committee, may be considered to be generic and it is hoped to model real world software processes using PADM to determine the extent that they may be ‘‘subclasses’’ of the process example. Ongoing action research with industry will continue to be a valuable source of real-world knowledge and a proving ground for ideas, theories and tools.

6.4 BM Operation Pattern and PML Role Action Guards

In PML, each action is guarded by selection conditions. In 4.3 it was pointed out that these guards can grow to be unmanageable very quickly as application behaviour is refined, or when changes are made to the model. Currently we are working on a method for generating PML guards from BM operation patterns. This facility should then give us a clearer understanding of the conditions under which the actions can

execute and at the same time eliminate redundant conditions.

6.5 OBM

The ISPW-6 example is a well captured process. In general, processes are often not clearly defined. It is important to provide mechanisms to capture the processes. Currently we are working on a method, called OBM derived from BM which provides a means to gradually identify the details of the processes in an organisation and develops the detailed representation of the organisational process by refinement.

7. Conclusion

We have shown through the medium of the ISPW-6 example that we have an approach to Software Process Modelling which allows us to:-

- rigorously specify a model from a narrative description
- validate this model against the narrative description
- implement this specification in an enactment language
- check that the implementation is consistent with the specification
- enact the ISPW-6 process on distributed workstations.

What remains to be demonstrated in terms of a total route for Process Modelling? Apart from obvious continual improvement to the technologies we have described, there are four key areas which we have not discussed in any depth.

- The issue of Process Acquisition. The ISPW-6 process was already *acquired* by its narrative. To complete the exercise we would have to return to the authors and subject them to a with hindsight refection on how the process was acquired. This is probably not realistic. The issue will however be comprehensively addressed in forthcoming work of the IPG.
- The issue of a real refinement of the BM specification into the implementation. We are currently developing our BM method to deal with this aspect.
- The issue of change which we will address by attacking the ISPW-7 process.
- The issue of implementation route. Software implementations have to be integrated into organisations in a planned and coherent manner. This concern will be part of future PADM development

Clearly the BM specification language represents but one aspect of the Process Analysis and Design Methodology (PADM) and some future work will concentrate on broadening our approach to the solution to ISPW-6/7 to include other aspects of Process Modelling. We also intend to experiment with using our solution to ISPW-6 as a generic approach by specialising our model to create an enactment system for a real example Software Development process. This raises two interesting questions. The first is that early on in the development of our BM specification we ran

into the problem of whether we were producing a specification of a model of the ISPW-6 process in order to observe the resulting model or whether we were specifying a model in which we were participating. This is a key issue in process models. Secondly, assuming that we actually used the ISPW-6 process to develop some software, what is the relationship between the process model and the product model? If a snapshot of active step objects in our model was taken this would also represent a definition of the product being developed; modules equating with step objects. Such thoughts of course lead us to the notion of reflexive systems. Thus we have two possible integrations of process and product models. One being an integration of the process used to develop the product with the product. The other being the integration of the process to use the product with the product. We are currently looking at these issues.

8. Acknowledgements

We would like to thank ICL for provision of their ProcessWise Integrator, and to Professor Peter Henderson for providing the Enact system used to construct the BM Stepper.

9. References

[1] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proceedings of the 16th A.C.M. Symposium on Theory of Computing*, 1984.

[2] R.F. Bruynoghe, J.M. Parker, and J.S. Rowles. PSS:A system for Process Enactment. In *Proceedings of the first International Conference on the Software Process, Manufacturing Complex Systems*. IEEE Computer Society Press, 1991.

[3] R. Conradi, C. Fernström, A. Fuggetta, and R.A. Snowdon. Towards a Reference Framework for Process Concepts. In *Proceedings EWSPT'92*, volume 635 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[4] B. Curtis, M.I. Kellner, and J. Over. Process Modeling. *Communications of ACM*, 35(9), September 1992.

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

[6] P.H. Feiler and W.S. Humphrey. Software Process Development And Enactment: Concepts And Definitions. In *Proceedings of the Second International Conference on the Software Process*. IEEE Computer Society Press, 1993.

[7] Sol J. Greenspan. Requirements Modelling; A Knowledge Representation Approach to Software Requirements Definition. Technical Report CSRG-155, Computer Systems Research Group, University of Toronto, March 1984.

[8] P. Henderson. *Object-Oriented Specification and Design with C++*. International

Series in Software Engineering. McGraw-Hill, 1993.

[9] A.W. Holt, H.R. Ramsey, and J.D. Grimes. Coordination System Technology as the Basis for a Programming Environment. *Electrical Communication*, 57(4), 1983.

[10] IPSE 2.5 Project Evaluation Report. IPSE 2.5 Project Document 060/00200, 1989.

[11] P. Kawalek. The Process Modelling Cookbook: Version One. Technical report, Support Technology Focus Project, British Telecommunications, 1991.

[12] M. Kellner, P. Feiler, A. Finkelstein, T. Katayama, L. Osterweil, M. Penedo, and H.D.Rombach. ISPW-6 Software Process Example. In *Proceedings of the First International Conference on Software Process*, Washington, 1991. IEEE Computer Society Press.

[13] J. Lonchamp. A Structured Conceptual and Terminological Framework for Software Process Engineering. In *Proceedings of the Second International Conference on Software Process*. IEEE Computer Society Press, 1993.

[14] L.J. Osterweil. Software Processes are Software too. In *Proceedings of the 9th International Conference on Software Engineering*, Monterey, 1987.

[15] M.A. Ould and C. Roberts. Defining Formal Models of the Software Development Process. In Pearl Brereton, editor, *Software Engineering Environments*. Ellis Horwood, 1987.

[16] PS-Algol Reference Manual. PPRR-12-88, University of Glasgow and University of St Andrews, 1988.

[17] C. Roberts. Describing and Acting Process Models with PML. In *Proceedings of 4th International Software Process Workshop*, Moretonhampstead, 1988.

[18] C. Roberts and A. Jones. Dynamics of Process Models in PML. In *Proceedings of 5th International Software Process Workshop*, Kennebunkport, 1989.

[19] J. Sa and B.C. Warboys. Specifying Concurrent Object-based Systems using Combined Specification Notations. Technical Report UMCS-91-9-2, Department of Computer Science, University of Manchester, July 1991.

[20] J. Sa and B.C. Warboys. Integrating a Formal Specification Method with PML: A Case Study. In *Proceedings EWSPT'92*, volume 635 of *Lecture Notes in Computer Science*, pages 106–123. Springer Verlag, 1992.

[21] J. Sa and B.C. Warboys. A Formal Description of the ISPW-6 Software Process Example. Technical Report UMCS-93-6-1, Department of Computer Science, University of Manchester, 1993.

[22] B. Singh. Interconnected Roles (IR): A Coordination Model. Technical Report CT-084-92, MCC, 1992.

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

[24] R.E. Strom and S. Yemini. NIL: An Integrated Language and System for Distributed

Programming. ACM SIGPLAN Symposium on Language Issues in Software Systems, 1983.

[25] B.C. Warboys. VME nodal architecture: a model for the realisation of a distributed system concept. *ICL Technical Journal*, 4(3), 1985.

[26] B.C. Warboys. The IPSE 2.5 Project: Process Modelling as a basis for a support environment. In *Proceedings of the First International Conference on Software Development, Environments, and Factories*, Berlin, 1989.

[27] P. White. Report on the Process Analysis and Design Methodology. Technical Report 142, IOPT, 1993.