Integrating a Formal Specification Method with PML: A Case Study

Jin Sa and B.C. Warboys,

Department of Computer Science,
University of Manchester,
Manchester M13 9PL,
England.

Published in the proceedings of the Second European Workshop on Software Process Technology EWSPT'92, September, 1992, Trondheim,

Norway, also in LNCS 635.

Integrating a Formal Specification Method with PML: A Case Study

Jin Sa and Brian Warboys

April 19, 1994

Abstract

This paper outlines a Case Study of the application of one approach to the Formal Specification of Parallel Object Oriented Systems to a Process Modelling language. The purpose was to both check whether such an approach was feasible and useful and also to give insight into the ongoing design of such Process Modelling Languages.

Keywords:*Process Modelling Languages, Formal Specification*

1 Introduction

Several formal approaches for process modelling have been developed. Some interesting examples can be found in [1], [2], [3] and [4]. These different approaches are generally so because they are based on different underlying models rather than for methodological reasons. In this paper we review a small case study which we recently undertook to investigate the application of an existing formal framework to Process Models (and Modelling Languages) . The modelling language chosen was the Process Modelling Language (PML) of the IPSE 2.5 Project [6] and the specification method was a modified version of the technique developed by our group at Manchester as part of the Esprit funded EDS project EP2025. This latter is known as Base Modelling (BM) and is reported in [7, 8].

Some recent case studies, which we have undertaken, have clearly demonstrated that the type of system that can be *specified* by BM is very similar in nature to the type of system that can be *modelled* by PML. This raises the question of what exactly we mean by *specified* and *modelled* and we shall return to this issue later in the paper.

As an immediate result of these case studies we have already modified BM so that it is more suitable for use as a specification language for applications targetted by PML. In particular we introduce the notion of *asynchronous* communication between the PML objects called Roles.

In this paper we use a *real-life* example to show that BM is suitable (and useful) for the type of applications targetted by PML. The example is specified using BM. Some of the implications of this type of Formal Specification to Process Modelling are then discussed.

The paper is organised as follows: section 2 briefly describes PML, section 3 describes the extended BM and section 4 describes how we specify systems described by PML using BM. Then section 5 describes the case study specification. Section 6 discusses the possible roles for such approaches in Process Modelling and we conclude in section 7 with a brief discussion of the suitability of the BM style for PML and the possible further integration of BM into the PML development process.

2 PML

PML is an object oriented process modelling language. The basic constructs being: Roles, Interactions, Actions and Entities (The resources owned by Roles may be transferred between Roles via Interactions). An instance of a Role is similar to an Object in that it can respond, in an independent way, to messages sent from other roles. Its behaviour being defined in terms of actions operating on its local entities, or variables. The computation of a Role is single threaded [1] in the sense that only one action is allowed to be active at any one time.

A system consists of a collection of Role instances executing in parallel and communicating with each other. A Role instance may communicate with another Role instance by sending messages to that other Role instance.

[1] UserActions in PML may be suspended which complicates the single threading rule but we shall ignore this complication at this time.

PML provides interactions as asynchronous, uni-directional buffered channels. Clearly, these can be used to establish different kinds of inter-role communication; in this paper we conveniently summarise them into the three following categories.

Given two role instances R1 and R2 then:

- function call: Role R1 passes an object to R2 and awaits a result.
- synchronous send: Role R1 passes an object to R2, R1 continues with no expectation of a result.
- asynchronous send: Role R1 passes an object to R2, R1 continues but expects a result at some later stage.

3 BM

The full description of BM is given in [7, 9]. In this paper only a brief outline is given.

3.1 Overview of BM

BM is a specification method aimed at the specification of object-based systems.

In BM, a system is considered to be composed of components which execute in parallel. Each component supports a number of operations and contains some state which can only be accessed by the operations. The computation of a component is single threaded. The behaviour of a component is determined by the pattern (BM *ordering expressions*[2]) to which its operations, when executed, conform. A component may only communicate with another component by calling the operations of the other component. In [9], BM is extended so that it is expressive enough to specify the three types of communication allowed by PML. Thus it supports function calls and synchronous and asynchronous sends as outlined above.

The semantics used as based on Temporal Logic and an brief overview of the model will suffice to illustrate the approach.

[2]There are two types of *ordering expressions* - operation patterns and call patterns.

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 set of sequences of events.

Thus assuming an operation *op* calls an operation *read* then the sequence of events for an execution of *op* is:

- `send` the request to call *read*
- `receive` the end of operation indication from *read*

The sequence of events for an execution of *read* is similarly:

- `accept` the request to execute *read*
- `return` the result of *read* to the caller

The model then matches `send`s to `accept`s and `receive`s to `return`s 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 must take place in parallel.

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 each operation provided by the component
- the behaviour of the interactions with other components
- the behaviour of the possible (sequential) combinations of the operations - the *operation pattern* type of *ordering expression*.

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)

- the external behaviour pattern of the collection of components
- and, in addition, specifying how the components may interact with each other.

Compared with PML, the behaviour of a Role instance can be specified using a single object and the behaviour of a system (a PML model), as defined by PML, using a composed object [3].

4 Specifying the behaviour modelled by PML using BM

In this section, we describe how to use BM to specify the behaviour of a system modelled by PML. In this paper, only the main features of BM are described, the entire concrete BM syntax is not given.

4.1 Specifying the behaviour of a Role Instance

From the description given above it is not difficult to see that the general behaviour of a Role instance can be specified using the BM construct *single-object*. The specific behaviour of a Role can be considered in terms of the types of communication noted in section 2 above. Such individual communication forms can be specified using the BM construct *operation*.

In PML, the behaviour of a Role instance is determined by the ordering and the number of occurrences of the executions of its Actions. Such an ordering is only implicitly defined through the definition of each Action. In BM a construct, called *operation pattern* is provided to allow the explicit specification of the constraint over the ordering of the Actions. These constraints can later be used to check the consistency of the whole model.

We simplify the specification of an Action, as defined in PML, by only considering its behaviour in terms of the following two aspects:

- interacting (or communicating) with another Role instance

[3]The behaviours of some special constructs cannot currently be specified using the current version of BM. e.g The UserActions and BehaveAs constructs of PML. We are currently studying possible approaches to specifying these modelled behaviours.

- updating the values of the variables belonging to the Role instance providing the Action.

Such behaviour may be specified using BM as follows:

- the interaction (or communication) with another Role instance, by means of sending messages, is specified using the BM construct *call template*. As mentioned in section 3 the extended BM is expressive enough to specify all three types of communications (interactions) allowed by PML.

- the ordering and the number of occurencies of the interactions is specified using the BM construct *call pattern*. We seperate the *call pattern* specification from the *call template* specification for two reasons:

  - first the specific behaviour - function, asynchronous etc will usually be added at the next phase of design after identifying the basic *call pattern*
  - secondly the call templates can be used to define a PML code generator tool which can map the specific calls onto the required run-time behaviour library by generating appropriate class definitions. This also ensures run-time conformance to the specification. (In EDS this approach was used to generate C^{++} class definitions with tasking calls to the Chorus Operating System [8])

- the functionality concerning the updating of variables is specified using the VDM style of pre- and post- conditions. Composition of the Temporal Logics, used to specify *ordering expressions*, with the VDM specifications is obtained through the use of *history variables* which allow the appropriate behavioural histories to be used in the VDM pre- and post- conditions.

To summarise, the behaviour of a Role instance is specified by:

- the operation pattern

- the declaration of the variables belonging to the role instance

- and, for each action:

- the call template for each interaction (communication)
- the call pattern
- the pre- and the post- conditions.

In BM, the overall semantics of a single object is automatically derived from the above information. Therefore, the overall semantics of a Role instance can also be automatically derived. The derivation of the semantics is not within the scope of this paper but can be found in [7].

4.2 Specifying the behaviour of a (sub)system

The behaviour of a (sub)system, as defined in PML, can be specified using the BM construct *composed-object*. Each Role instance in the (sub)system is specified using a *single-object*.

A *composed-object* specifies the constraints over the ordering and the number of occurencies of the operations of the (sub)system using the *operation pattern* construct. The specific protocol by which the Role instances synchronise with each other is specified using the *synchronisation template* construct.

As in the case of a single-object, the overall semantics of the (sub)system can be automatically derived from the above information.

5 BM Specification of the *Surrey* example

As a case study, we took a system, called "Surrey Project Forecasting" (referred to as the *Surrey* example throughout this paper), that has been modelled using PML [11] and specified it using BM. The full BM specification of the *Surrey* model can be found in [10]. The case study has shown that the behaviour of the *Surrey* example modelled in PML can be specified in BM. It has nevertheless raised a series of questions about the role of specifications in process models as well as issues concerned with the design methodology which should be used for such modelling. In this paper, we present part of that example to illustrate how it is specified using BM.

5.1 The *Surrey* Example as Modelled by PML

The *Surrey* example is about the co-ordination of work amongst a number of system designers working on the design of a large mainframe computer.

The PML model of the example, as defined in [11], has centred on the assignment of workpackages by the design manager to the design engineers. It has modelled the following type of roles:

- The **System Administrator** establishes the system by setting up potential workpackages and designers.
- The **Design Manager** manages the design activities of the overall system by allocating workpackages to designers and keeping track of the work progress.
- A **Designer** is to carry out the work defined in a workpackage.
- An **Auditor** is to examine the work produced as part of a workpackage.

In this paper, we will only present the BM specifications of the design manager role, called DM in the PML model and the designer role, called Workpackage in PML.

5.1.1 DM

DM provides the following actions:

- StartDM - Initialises the variables of the DM instance.
- AssignWorkpackage - binds a workpackage to a designer. This action causes an instance of workpackage to be created.
- SeeGroupProgress and SeeWorkpackageProgress - display the appropriate information collected from the individual workpackage instances.
- ReceiveDayReport - receives a day report from a workpackages instance, and updates the table containing the information of all the bound work packages.

5.1.2 Workpackage

The work in a workpackage is divided into phases. The work is sent to the auditor to be audited after each phase. The workpackage may proceed to do work in the next phase. However the work in a phase is not allowed to be sent for auditing until its previous phase has passed for auditing. The following actions are provided:

- StartWorkpackage - initialises the variables of the workpackage instance.
- Dowork - Models any type of work during a phase, e.g. simulate chip design.
- SendforAudit - Sends a phase to be audited. This action causes an instance of auditor to be created.
- ReceiveAuditResult - Receives the confirmation from the auditor. (For simplicity, the PML model assumes that the auditor will always pass the work).
- CompileDayReport - Completes a day report and despatches it to the DM instance.

5.2 Specification of the Behaviour of the DM Instance

In section 4.1, we have described that a specification of the behaviour of a role instance mainly consists of:

- operation patterns
- variable declarations
- specification of each operation

5.2.1 Operation Pattern

The ordering and the number of occurrences of the actions provided by the DM instances are specified by the following **operation pattern**.

$StartDM; AssignWorkpackage;$
$(AssignWorkpackage \oplus SeeGroupProgress \oplus SeeWorkpackageProgress$
$\oplus\ ReceiveDayReport)^+;\ fin$

In this paper, the symbols used in BM are explained as they appear in the example. In the above expression, the symbols denote:-
P;Q specifies that Q follows P.
P $\oplus$ Q specifies that either P or Q occurs
P^+; Q specifies that P may be repeated any number of times before Q occurs.
fin indicates the end of computation.

The above expression states that StartDM is the first action of the DM instance, followed by the execution of AssignWorkpackage. After one Workpackage is assigned, the DM instance is allowed to repeatedly execute any of the following actions: AssignWorkpackage, SeeGroupProgress, SeeWorkpackageProgress or ReceiveDayReport.

It is clear that there is much in common between the semantics of these expressions and the much simpler semantics of path expressions. The motivation for introducing the weightier semantics of Temporal Logic rather than utilising simpler regular expressions is concerned with the specification framework which we are trying to create.

Basically we wish to be able to compose a full specification through the seperate sub-specification of the differing parts of a system definition. In particular we wish to be able to specify the constraints of parallelism seperate from the much easier task of specifying the constrained sequential behaviour. Further we wish to use more *familiar* languages, such as VDM, for this sequential specification. Attempts at extending sequential specification to include parallelism, the alternative approach, have in our opinion generated very complicated languages, making their industrial use very unlikely.

Clearly in our approach, to achieve a complete specification, these seperate parts need to be composed together; hopefully automatically by tools. Temporal Logic provides a suitable framework for such composition.

5.2.2 Variable Declarations

The DM instance contains variables which record information regarding the assigned Workpackages. The syntax for declaring the variables follows the VDM style as described in [12].

When the system administrator creates the DM instance, it passes a pool of potential Workpackages and a pool of potential users (designers) to the DM instance. Two variables are used to maintain this information.

wppool: *WPType-set*, *userpool*: *UserType-set*

wppool contains a set of potential Workpackages, and userpool contains a set of potential users. The precise definitions of the types for workpackages and users are not significant for the purpose of this paper.

When the DM instance assigns a Workpackage to the user, we use a type, called BoundWPType, to represent this information.

BoundWPType: : *WPdata*: *WPtype*, *User*: *UserType*

Since the DM may assign a number of workpackages, a table of the bound workpackages must be maintained, we use the following type to maintain the table

BoundWPTableType: : *WPIndexType* $\rightarrow$ *BoundWPType*

Let wpinfos be the variable which maintains the information of all the bound workpackages, so

wpinfos: *BoundWPTableType*

Let wpind be the variable which maintains the number of the bound workpackages, so

wpind: *wpIndexType*

5.2.3 Specification of each operation (PML action)

In this paper, we will only describe the specifications of AssignWorkpackage and ReceiveDayReport. The other actions are specified similarly and can be found in [10].

- **call templates** of AssignWorkpackage

  AssignWorkpackage calls the initialisation action of the appropriate Workpackage instance. The **call template** for calling the action Start-

Workpackage is as follows:

$callStartWp_{wpind}$	
prop	*function*
obj	$Workpackage_{wpind}$
op	*StartWorkpackage*
apar	*wpdata* *user*

Since there are several instances of workpackages, each instance of workpackage is subscribed.

In the above template,

- `prop` indicates the type of the call, see section 3 for information regarding the different types of calls (or protocols) . In this case it is of type `function`. callStartWp_{wpind} is the name of the *call template*.
- `obj` identifies the role instance being called. In this case $\text{Workpackage}_{wpind}$ is the name of the instance.
- `op` identifies the operation within `obj` being invoked. In this case StartWorkpackage is the name of the operation.
- `apar` specifies the actual parameters of the call. In this case they are called wpdata and user.

* **call patterns** of AssignWorkpackage. Since there is only one occurrence of a call within the execution of AssignWorkpackage, the call pattern is simply

$callStartWp_{wpind}; fin$

* **pre- and post- conditions** of AssignWorkpackage. AssignWorkpackage takes a workpackage, wpdata, and a user, user, as its parameters, the pre- condition specifies that the workpackage and the user to be bound together must exist. The post- condition specifies that after

the execution of AssignWorkpackage, the following condition must be satisfied:

- the number of bound workpackage is increased by 1,
- the newly bound workpackage is added to the table of bound workpackages.

$$AssignWorkpackage(workpackage, wpdata, user) \quad \triangleq$$

$$\mathsf{pre}\ (wpdata \in wppool) \wedge (user \in userpool)$$

$$\mathsf{post}\ (wpind' = wpind + 1) \\ \quad \wedge\ wpinfos' = wpinfos \dagger \{wpind \mapsto mk(wpdata, user)\}$$

- **call templates** of ReceiveDayReport

  ReceiveDayReport does not send any message to any other instance, therefore there is no call template.

- **call pattern** of ReceiveDayReport

  The call pattern is simply

  $$\mathit{fin}$$

  indicating that there is no occurrence of call.

- **pre- and post- conditions** of ReceiveDayReport

  ReceiveDayReport takes the day report dayrep and the index of the workpackage, wpindex, as its parameters. The pre- condition specifies that the index of the workpackage must be within the range kept by the DM instance, The post- condition specifies that the table containing the bound workpackages must be updated.

  $$ReceiveDayReport(dayrep, wpindex) \quad \triangleq$$

  $$\mathsf{pre}\ wpindex < wpind$$

  $$\mathsf{post}\ wpinfos' = wpinfos \dagger \{wpindex \\ \quad \mapsto mk(new(updata, dayrep), user)\}$$

5.3 Specification of the behaviour of the Workpackage$_i$

5.3.1 Operation Pattern

Before defining the operation pattern of Workpackage$_i$ the following abbreviations are defined.

$Dowork1 \triangleq Dowork; CompileDayReport \ \oplus \ Dowork$
$SendforAudit1 \triangleq SendforAudit; CompileDayReport \ \oplus \ SendforAudit$
$ReceiveAuditResult1 \triangleq ReceiveAuditResult; CompileDayReport \oplus ReceiveAuditResult$

These three abbreviations state that the action CompileDayReport may take place after an execution of one of the three actions: Dowork, SendforAudit and ReceiveAuditResult.

The **operation pattern** of Workpackage$_i$ is:

$StartWorkpackage;$
$(Dowork1^+; \ SendforAudit1; Dowork1^+; \ ReceiveAuditResult1)^+; \ fin$

This expression states that the first action to be executed is StartWorkpackage. Following that Workpackage$_i$ is allowed to repeatedly perform operations such that their execution pattern satisfies Dowork1. After doing some work, the work can be sent to the auditor by executing SendforAudit and after that Workpackage$_i$ may do more work. Finally, Workpackage$_i$ will receive the audited result be executing ReceiveAuditResult.

After receiving the audited result for a phase, the whole process is repeated again for the next phase by doing some work, sending for auditing etc.

5.3.2 Variable Declarations

In this paper, we shall only describe the variables used in the subsequent specification.

A variable called cphase is used to indicate in which phase Workpackage$_i$ is currently engaged:-

$cphase\colon 1..maxphaseno.$

SendforAudit is not allowed to send works beyond the current phase to the auditor. cphase is only incremented after the workpackage receives the audited result of the current phase.

5.3.3 Specification of each Operation (action)

Most of the operations are specified in a similar way to those described in 5.2.3. Here only the specification of CompileDayReport is described to illustrate the interactions between the DM instance and the workpackage instance.

- **Call template** of CompileDayReport

callReceiveDayReport	
prop	*async.send*
obj	*OCPD*
op	*ReceiveDayReport*
apar	*report* *i*

  Note that the actual parameters report and i are bound to the formal parameters, dayrep and wpindex respectively, of the action ReceiveDayReport.

- **Call Pattern** of CompileDayReport

$callReceiveDayReport; fin$

- **pre- and post- conditions** of CompileDayReport. The pre- condition would be defined by some predicates stating that the day report is needed, e.g. end of the day. The post- condition would specify that the day report is completed.

6 Specification of process models

This paper has shown that BM can be used as the specification method for applications targeted by PML, although it was not originally designed especially for process modelling. Clearly there is still a wide scope for improvement so that it is a more suitable technique for process modelling in the PML Style.

In general, of course, a necessary starting position when considering **specification** in the context of process models is to be clear what is meant by the term and how having a specification might help.

Traditionally the role of a specification is to describe desired properties for a system (the **what**) in contrast to describing implementations (the **how**). The idea of a **formal** specification is to allow mathematical deductions of implied properties from the specification and, in theory at least, proofs that implementations do satisfy the requirements.

Because systems of interest are usually quite large and quite complex, it is valuable to be able to express specifications (and, of course, implementations) at different levels of abstraction (related to the idea of reification or refinement) and as compositions.

Following these thoughts, a possible use of specification in process modelling is to express, formally, desired properties of the model. This is too imprecise a statement. Thus, to make things a little more concrete the presumptions of process models that underlie PML will be used as a basis.

The idea in PML is that a process model is a "topology of roles and interactions". A role is a single threaded process occupying time and space. A role interacts with other roles through a number of interactions which are uni-directional, buffered channels providing a means of asynchronous communication.

The behaviour of a role (that is the activities which take place and the resources used) are prescribed by a definition written in the language PML. An interpretation of such a definition is that it provides an implementation description of the role. By interpreting the definition in an operational way, the behaviour pre-defined for the role will be enacted and the described effects will occur. Interpreting or comprehending a PML definition in order to determine properties of the objects (roles) denoted by the definition is not easy or simple.

Thus one use of a specification of a process model is as a formal description from which properties may (more easily) be deduced. This would be particularly beneficial if it allowed specifications of "large" models at different levels of abstraction and as compositions of other model specifications[4]. The first of these certainly has methodological implications, suggesting that an abstract model may be refined into less abstract models.

[4]For simplicity's sake this ignores issues of specification in a changing model.

6.1 Some issues arising from the case study

There is insufficient space in this paper to consider all of the issues related to specification and process modelling but it is worth briefly reviewing some rather essential elements that come out of the case study.

As a starting position, assume that it is inappropriate to consider PML itself as a specification (from which we can deduce properties and so on) so we are interested in using some other technique (i.e. Base Modelling) to describe formally the behaviour of a role and thence of a set of roles and interactions.

At a primitive level an individual role can be considered as some sort of autonomous entity which sends and receives messages to and from its environment[5]. It might be valuable to describe the behaviour of such an entity in a form which will allow deductions to be made of what is described, and, further, which can be composed with similar descriptions to form a description of a larger system. The idea is to do this in a way which is *better* than PML.

6.1.1 A place for Base Modelling (BM)?

The example in [10] describes a specification which gives a "userview" of a process model. That example exemplified that BM can be used for such purposes. What does, or might, this mean for a more general PML model?

As has been noted earlier, there is a seemingly obvious relationship between an object in BM and a role in PML. Both are single threaded processes having local state and interactions with their respective environments. If a BM object specification can be given for a role, then the BM notion of composed object should immediately give a way of specifying the behaviour of groups of roles.

6.1.2 Role equals object?

A BM object is specified in terms of the specification of the operations it supports together with an operation pattern. Earlier we saw that each operation is specified by

[5] Thus the work done on the Formal Specification of Actor Systems is clearly relevant.

- pre- and post-condition predicates over the state of the object and the parameters and result of the operation.
- call templates for each further external operation called as a result of the operation being invoked.
- a call pattern which specifies the order and occurrences of the further calls identified by these call templates.

The first problem to be considered is that although a BM object supports a "userview" of the process models it does not explicitly support the more general PML style operations. The operations of BM objects are very much like procedures in programming languages. A call is made, some "computation" (denoted by items 2 and 3) occurs and a result is returned. The extensions to cope with asynchronous calls merely introduce a "hidden" buffer, the underlying model being still retained. The pre-state identified in a post-condition is well defined as being that existing at the invocation of the operation.

A role in PML communicates with its environment through interactions; a rather different concept.

One possibility is to make an abstraction, treating interactions purely as mechanisms, and consider roles as supporting BM-like operations. Such a scheme requires that a role be able to support concurrent execution of operations (at the BM level) because it may receive another procedure call message whilst still processing an earlier one. Hence the asynchronous extensions to BM. However such an approach represents a significantly different approach to process modelling than that associated with PML. Essentially we would be Modelling Roles as traces of possible behaviour. At present a feature of the PML approach is that the major emphasis is given to the description of the specific behaviour of a Role to achieve some given result (objective). A set of rules prescribing its behaviour rather than the observation of its behaviour. Basically BM is giving the set of constraints on the ordering of the Role Actions rather than these prescriptions.

The next two subsections take a different position, considering the ideas of PML as paramount and seeing how BM might fit.

6.1.3 Black box roles

The first idea is to consider a role as an object having state and exchanging messages with its environment through a set of interactions. From the viewpoint of the role, it might be useful to describe what happens when a PML `Take` action is performed in order to receive a message. This is rather like the accept event of a BM operation call in which no result is returned to the caller (the environment).

Suppose that a role object has a set of `TakePorts`

$\{T_1, T_2, \ldots T_i, \ldots\}$

and that there is, notionally at least, a set of $read\_T_i$ operations (equating to PML `Take` actions) which we might associate with these.

Then we might give pre- and post-condition style specifications for the local effect (on the role state) of the invocation of such operations. The only feature of such specifications is the lack of any result parameter in the post-condition and the effect of any intermediate calls. i.e. the history variables mentioned in 4.1 above.

This scheme allows the (limited) specification of how a role is affected by the environment and under what conditions, The operation pattern of BM can be applied to prescribe the ordering of $read\_T_i$ operations. Note that such a prescription does not imply that the "environment" may not send messages in only the prescribed order, just that the role in question will only behave according to the pattern[6].

The next question to consider is the specification of the effect of a role on its environment. That is, in what circumstances do messages appear on outbound interactions?

Following the above, suppose that a role has a set of `GivePorts`

$\{G_1, G_2, \ldots, G_j, \ldots\}$

and that there is, notionally at least, a set of $write\_G_j$ operations (equating to PML `Give` actions) associated with them.

As before, pre- and post-condition style specifications may be given for these operations, the particular feature here being that the pre-condition is a predicate only over the state of the role. Note that the post-condition may

[6]the same is true in BM specs. The behaviour of the environment is "defined" in call patterns and compatibility is a property of later composition.

indicate a change both in the environment and in the local state (unless the PML action is of the `GiveCopy` variety).[7]

Again some sort of operation pattern can be applied to prescribe an ordering over $write\_G_j$ operations. In fact, an ordering expression can be defined which includes both $read\_T_i$ and $write\_G_j$ operations providing some means of describing how they are related. However, this raises an interesting question about BM's application in this instance.

In specifying the (external) behaviour of a role in terms of $read\_T_i$ and $write\_G_j$ operations, individually specified by pre- and post-conditions together with some call pattern, it is clear that something rather important is missing. Generally it will not be the case that the post-condition of $read\_T_i$ operations imply pre-conditions of $write\_G_j$ operations. This does not matter in the BM specification because the call pattern will have defined the ordering of such calls. However the receipt of a message by a role will generally signal some "internal" computation which will most likely transform the state. What is missing is any way to specify such transformations.

In other words the important feature from the PML perspective is the **how** rather than the **what**! Hence the issue of the role of a specification in this context. Of course it could be argued that this is really just a consequence of PML being an imperative rather than a declarative language.

A simple example is sufficient to illustrate the problem. Imagine a role with one `TakePort` and one `GivePort`. This role has one purpose. When it receives an integer on the `TakePort` it adds one to it and then transmits it out through the `GivePort`. If this "receipt, increment, send" behaviour was considered as a single operation called, say *inc*, then we might have a specification as follows perhaps.

$inc\ (n, r\colon int)$

$\mathsf{pre}\ true$

$\mathsf{post}\ r = n + 1$

where r represents the output result and n the input integer. Specifying this in terms of just *read* and *write* is a little more complicated. To save

[7] Something which has been glossed over a bit is whether these $read\_T_i$ and $write\_G_j$ operations are invoked by the environment or by some other means. For now assume that the environment invokes them (that retains an image closer to BM) although that isn't really the case in PML.

space assume that the state of the role comprises a single integer called v. Then we might have

$read\ (n{:}\ int)$
pre $true$
post $v = n$

specifying that the input integer (n) has been stored in the state, and

$write\ (r{:}\ int)$
pre ??
post $r = v$

specifying that the message transmitted was the value in the state. However, where is the requirement that $r = n + 1$ specified? Within the framework suggested here there is probably only one way of achieving this; by the explicit recording of appropriate objects in the state. This seems to defeat the whole purpose of the exercise.

6.1.4 White box roles

The previous section considered a role as a "black box" whose specification was only concerned with a role's interaction with its environment. Some problems were noted, in particular that concerning the activity of a role when it is not reading from or writing to interactions. BM objects are passive and only exhibit behaviour when an operation is invoked. PML roles are active and may exhibit behaviour even though there is no immediate external stimulus.

A formal description of a role could describe its complete behaviour in terms of "operations". Operations on interactions would be handled very much as in the preceding section. "Operation" which are internal to a role could be considered as if they were invoked autonomously.

Thus, taking the example from the previous section we might have, in addition, an *inc* operation to actually increment the integer read from the environment. A possible operation pattern for the role might be:

$(read; inc; write)^+;$

with operation specifications, based on a state containing only a component v as earlier. The VDM formulae are obvious and follow the same pattern as above with the addition of the inc operation.

The level of detail implied by this sort of approach would be such that there would need to be an "operation" defined for each action property of a role at least. Further complications arise if an action property is actually a composite action (because it is expressed in PML in terms of control operators such as ";", either explicitly or in an action class definition). Such instances are examples of where the BM idea of a call pattern (i.e. behaviour invoked by use of an operation) would be used. Certainly the semantics of action properties for a role is such that, once a particular (top level) action property is invoked it will complete[8]. However, the level of detail would only need to be such as to identify *read* and/or *write* operations representing "external" behaviour (cf the call templates of BM).

6.1.5 Composed objects

BM derives a lot of expressive power through the idea of composed objects whose behaviour can be systematically derived from the behaviour of its components. How would this work in the context of BM specifications of PML roles?

The basic rule used in composing behaviours in BM is that of synchronising invocations of operation calls with the behaviour that is thereby invoked. That is, if the behaviour of object A contains a call on operation x of object B, then the behaviour of A combined with B is that of A and B together subject to the synchronising constraint for the call of x in A with the behaviour of x in B.

As a start consider what this means for the "Black box role" specification described above. In such a specification, the behaviour of some role $R1$ will include invocations of *read* and *write* operations. *read*s may be considered as being operations of $R1$ (that is the temporal logic behaviour of $R1$ will contain *accept* or *receive* events - see [7]), whilst *write*s may be considered as being invoked **by** $R1$ (that is the temporal logic behaviour of $R1$ will contain *send* or *return* events)

Composing $R1$ with some other role (say $R2$) would require composing

[8] Disregarding the complications of `UserActions` !

their behaviours subject to constraints synchronising appropriate *accept* of an operation $R2$ and *send* events and *receive* and *return* events.

As interactions are buffered channels in PML, composition of roles isn't quite so straightforward. However, as discussed in [9] the solution to this is to introduce intermediate objects having a behaviour of their own to represent interactions. Thus composing the behaviours of roles requires, additionally, the inclusion of behaviours for all interactions between roles in the composition. Addition of these behaviours should maintain the composability capabilities provided by BM. Clearly, although the above discussion was based on a black box view of a role, the same conclusions should be true for the potentially more useful white box specifications.

6.2 Discussion

In the above there has been some consideration of the place for a specification technique such as BM in process modelling activities based on PML. It has ignored many of the complications (dynamic creation and modification of objects for instance).

The study has led to some significant reservations, not as to the fact that such a technique **could** be used, but more as to **how** and **why**. Obviously BM could be used as an alternative to PML to give a "white box" description of the behaviour of a role, and at much the same level of detail. Such behaviours could be composed to give a behaviour definition for a group of roles and so on.

We suspect that such a "white box" approach will be a basis for any advances, but, as it stands, it fails to provide a sufficient abstraction of the behaviour of a role (or of a group of roles) and as such the level of detail will be too great. This says that an important next step might be to see if there are useful ways of "hiding" those details which might be considered inappropriate in certain valuable abstractions.

One obvious remedy is to encourage the user (the specification writer) to group together some *small* operations into *big* ones. Thus in our example in section 6.1 above the *read:inc;write* is grouped as *increment* and this is only expanded if one needs to consider the details. In fact in BM *small* operations are grouped together in a pre-determined way via the `send/accept` etc bindings. In order to cope with a more general model the question is whether to just leave it to the user to group operations or whether to introduce a new pre-defined abstraction device (i.e some notion of modularity). We are still undecided on this issue.

Finally it is interesting to note that the specification in [10] appears to be based on a "user view" of the process model rather than on the PML per se. Such an approach is obviously valuable and is probably a variant on the "white box model" in which the actions of interest are limited to `UserActions`. More generally the specification of `UserActions` in PML may not be that straightforward as there are some scheduling peculiarities in this area. This might cause some conflict with the white box view that a role is an autonomous agent invoking its own top level actions as BM operations.

7 Further Work

The example has shown that there are many similarities between BM and PML such that BM can be used as the specification language for systems targeted by PML. However there are still some other aspects of PML need to be investigated. In general BM is expected to expand to include more entities specific to process modelling. The unresolved issue for us is that concerned with the apparent contradiction between PML's desire to concentrate on the definition of the way in which a Role's behaviour satisfies an objective and the desire of BM (and other Formal approaches) to specify what observable behaviour the Role should satisfy.

8 Acknowledgements

We would like to express our thanks to Bob Snowdon not only for initially raising many of the issues discussed here but also for suggesting substantial and helpful revisions to the original text.

References

[1] G. E. Kaiser, N. S. Barghouti, P. E. Feiler, R. W. Schwank, "Database Support for Knowledge-Based Engineering Environments" IEEE Experts, 1988.

[2] G. E. Kaiser, P. H. Feiler, S. S. Popouich, "Intelligent assistance for software development and maintenance", IEEE Software, May 1988.

[3] W. Emmerich, G. Junkermann, B. Peuschel, W. Scchafer, S. Wolf, "Merlin: Knowledge-based Process Modelling", First European Workshop on Software Process Modelling, May, 1991.

[4] V. Gruhn, "The Software process Management Environment Melmac", First European Workshop on Software Process Modelling, May, 1991.

[5] S. Bandinelli, a. Fuggetta, C. Ghezzi, "Software Process as Real-Time Systems: A case study using High-level Petri Nets", First European Workshop on Software Process Modelling, May, 1991.

[6] B. C. Warboys, "The IPSE 2.5 Project: Process Modelling as the basis for a Support Environment", Procs First International Conference on Software Development, Environments and Factories, Berlin, 1989.

[7] J. Sa, B. C. Warboys, "Specifying concurrent object-based systems using combined specification notations", Technical Report Series UMCS-91-9-2, Department of Computer Science, University of Manchester, July, 1991.

[8] A. Apps, "A Prototype BM Specification Tool", EDS document EDS. WP6I. AA16, Department of Computer Science, University of Manchester, Februrary, 1992.

[9] J. Sa, "BM Extensions for Asynchronous Communications", Draft note, Department of Computer Science, University of Manchester, February, 1992.

[10] J. Sa, "BM Specification of the Surrey Project PSS Demonstration Model", Draft Note, Department of Computer Science, University of Manchester.

[11] I. Robertson, "Surrey Project Forecasting - PSS Demonstration", IPG document, Department of Computer Science, January, 1992.

[12] C. B. Jones, "Systematic Software Development Using VDM", Prentice-Hall International Series in Computer Science, 1986