A SURVEY AND ANALYSIS OF PROCESS MODELING LANGUAGES

Kamal Zuhairi Zamli and Nor Ashidi Mat Isa
Software Engineering Research Group
School of Electrical and Electronics Engineering
Universiti Sains Malaysia
Engineering Campus
14300 Nibong Tebal
Pulau Pinang, Malaysia
Tel: 604-5937788 ext 6079
Fax: 604-5941023
email: eekamal@eng.usm.my
ashidi@eng.usm.my

ABSTRACT

Process Modeling Languages (PMLs) are languages used to express software process models. Process Centered Software Engineering Environments (PSEEs) are the environments used to define, modify, analyse, and enact a process model. While both PMLs and PSEEs are equally important, it is the characteristics of PMLs that are the focus of this article. Over the past 15 years, there have been many PMLs (and PSEEs) developed. Despite many potential advances, the use of PMLs in industry has not been widespread. As PMLs could form a vital feature for future software engineering environments, it is useful to reflect on the current achievements and shortcomings, and to identify potential areas of omission. It is also useful to explore issues emerging from related research areas, the adoption of which could improve the applicability and acceptance of PMLs. Given such potential benefits, this paper presents a critical analysis of existing PMLs identifying each language's strong points and weaknesses, thereby forming guidelines for the future design of PMLs.

***Keywords:** Process Modeling Languages, Software Process, Software Engineering*

1.0 INTRODUCTION

A PML is analogous to a programming language in the sense of providing a model solution of a particular problem. However, there is a subtle difference between a PML and a programming language which lies in terms of their computational models. Unlike the familiar computational model in computer science where sequences of operations specified by computer programs are automatically executed by the central processing unit, the computational model demanded by PMLs is somewhat different. This is because PMLs require some operations to be executed by (error-prone) software engineers using some defined software tools, while other operations are suitable for automatic execution (e.g. automation of routine operations). As such, developing an ideal PML to support such a computational model poses a challenge to researchers in computer science. In response to this challenge, many PMLs have been developed and described in the literature over the last fifteen years.

Despite many potential advances, the use of PMLs in industry has not been widespread [19]. As PMLs could form a vital feature for future software engineering environments, it is useful to reflect on the current achievements and shortcomings, and to identify potential areas of omission. It is also useful to explore issues emerging from related research areas, the adoption of which could improve the applicability and acceptance of PMLs. Given such potential benefits, this paper presents a critical analysis of existing PMLs identifying each language's strong points and weaknesses, thereby forming guidelines for the future design of PMLs.

2.0 OVERVIEW OF PML ISSUES

A number of researchers have classified these different language paradigms in the context of a PML based on their experiences with programming languages. Liu and Conradi [21] identify five categories of PML language paradigms:

- Active Database PMLs – PMLs which relies on database triggers employing Event-Condition-Action rules as the basis for the language.
- Rule-based PMLs – PMLs which exploits rule-based planning techniques or blackboard architectures in the language.
- Graph/Net PMLs – PMLs which utilises graphs or Petri Nets.
- Process Programming PMLs – PMLs which define a process model as a computer program based on a general purpose programming language.
- Hybrid PMLs – PMLs which fits into more than one of the above categories.

Although with different headings, Lonchamp [22] proposes a similar classification of PMLs, consisting of the following categories:

- Graphical PMLs – PMLs which provide a graphical syntax.
- Net-oriented PMLs – PMLs which utilise nets such as Petri nets.
- Procedural PMLs – PMLs which adopt a procedural programming language.
- Object-oriented PMLs – PMLs which utilise some features from object-oriented languages (e.g. objects and inheritance).
- Rule-based PMLs – PMLs which exploit rules-based techniques mainly based on Prolog.
- Multi-paradigm PMLs – PMLs which utilise more than one of the above categories.

The slight difference between the classification from Liu and Conradi with that from Lonchamp is that the latter includes object-orientation.

Building from the classifications from Liu and Conradi, and Lonchamp, Huff [18] identifies four different categories of PML language paradigms:

- Non-executable PMLs – PMLs which provide defined syntax but without executable semantics.
- State-based PMLs – PMLs which uses hierarchical state machines, Petri nets, or formal grammars as the basis of the language.
- Rule-based PMLs – PMLs which rely on a rule-based approach (i.e. based on Prolog) or database triggers (i.e. based on Event-Condition-Action (ECA) rules).
- Imperative PMLs – PMLs which rely on a model of computation whereby software processes are modelled as step by step sequences of commands.

The notable difference between the work of Liu and Conradi, Lonchamp, and Huff is that the latter includes the non-executable category of PMLs. Essentially, the outcome of Huff's non-executable categorisation is that graphical high-level notations such as Integration Definition and Function Modeling (IDEF0) and Entry condition, Tasks, Verification and Exit Criteria (ETVX) can also be considered as PMLs because of their syntactic abilities to express a software process [18]. It should also be noted that Huff's non-executable category of PMLs does not implicitly imply that PMLs in other categories are enactable. This is because some PMLs can belong to more than one category.

A more recent classification of PMLs is that of Ambriola *et al* [1]. This classification breaks away from the earlier classifications, as PMLs are classified according to the process lifecycle that they support rather than being based on their language paradigm. The main categories are:

- Process Specification Languages (PSLs) – used in the specification phase of the software process, and typically make use of formal notations.
- Process Design Languages (PDLs) – used to support the design phase of the software process.
- Process Implementation Languages (PILs) – used to support the implementation phase of the software process.

Regardless of which classification is used, some PMLs can fit into more than one category. This is because some PMLs combine different language paradigms, and therefore cannot be classified under one particular category. Additionally, such an overlap may also occur because the scope of coverage of a PML can be very large covering many aspects of the modeling process from the requirement phase to implementation.

As far as assessing the existing PMLs, much research has already been done. Lonchamp [22] reports some results of evaluating PMLs using a set of questionnaires given to the authors of each PML. The questionnaires covered:

- The modeling approach – language constructs used to express activities and their pre-conditions and post-conditions as well as ordering constraints and parallelism, input and output artifacts, and roles.
- The underlying language paradigm.
- The tools support (e.g. editors, compilers).
- The enactment capability, the meta-process and the evolution support.
- The resulting assessment, however, is targeted to the PMLs developed under the European research consortium called Process Modeling Techniques Research (PROMOTER).

Complementing from the work of Lonchamp, Ambriola *et al* [1] define an assessment grid for evaluating both PSEEs and PMLs. As far as evaluating PMLs is concerned, the assessment grid covers:

- The PML scope of coverage – the part of the process lifecycle the PML supports.
- The underlying language paradigm.
- The modeling approach.
- The support for modularity, composition and reuse.
- The mechanism for process enactment and evolution.
- The tool support.

Like Ambriola *et al*, Conradi and Jaccheri [12] also define an assessment grid in the forms of requirements for PMLs and PSEEs, identifying the primary and secondary process elements (i.e. in terms of what constitute a software process) that a PML and a PSEE need to support. In the context of this research work, only the primary process elements are considered because they constitute the requirements of PMLs whilst the secondary process elements will be ignored as they constitute the requirements of PSEEs. According to Conradi and Jaccheri, the primary process elements consist of:

- Activities, their pre-conditions and post-conditions, ordering constraints, and parallelism.
- Input artifacts and output artifacts as products.
- Human and their roles representation.
- Tool support.
- Evolution support.

As has been shown, there have been a number of attempts to classify and characterise the requirements of PMLs. However, one aspect perceived to be lacking in the previous classifications of PMLs is the consideration of the human dimension which relates to the issues surrounding the software engineers (or process engineers) who create the process models and initiate enactment as well as the software engineers who are subjected to process enactment.

Paradoxically, human dimension issues have always been a major concern in research into software processes [2, 26]. However, current trends in the way software engineers work (e.g. cross organisational boundary, geographically and temporally distributed locations) suggest that much more could be done to address these issues. These issues include: providing support for process engineers in terms of utilising visual syntax; enactment within a virtual environment; supporting user and process awareness; process visualisation; virtual meetings; as well as reflecting that support in the features provided in a PML [32].

Capitalising on these issues relating to the human dimension and building from the earlier characterisations and requirements of PMLs, Table 1 presents an alternative characterisation of PMLs which forms a taxonomy for PMLs. This taxonomy for PMLs is based on our earlier work (described in [32]) and differs from other work mainly by the inclusion of the human dimension issues.

Table 1: Taxonomy for PML

	PML Characteristics
Modeling Support	Sequential and parallel activities as well as their constraints
	Input and output artifacts
	Role representations
	External tools
	Abstraction and modularisation
Enactment Support	Enactment in a distributed environment
	Dynamic allocation of resources
Evolution Support	Reflection
Evaluation Support	Collection of enactment data
Human Dimension Support	Visual notations
	User awareness
	Process awareness
	Process visualisation
	Virtual meetings

Each issue given in Table 1 will now be discussed in detail next.

Modeling

Derniame *et al* [15] have identified the following constituent parts of a process model which therefore have to be represented in a PML:

- Activities – Activities are any actions performed by software engineers or by computers to achieve a certain set of goals. Examples of an activity can include high-level design or compilation of a program. In term of enactment, activities can be sequential or parallel and are always associated with artifacts (described below) and sets of pre-conditions and post-conditions. Also, depending on its needs, an activity may be performed by a single software engineer (e.g. modify a design) or collaboratively performed by a group of software engineers (e.g. review a design).
- Roles – Roles identify the skills required for performing a particular activity. In many cases, software engineers may assume many different roles based on their skills.
- Artifacts – Artifacts represent the inputs to and the outputs from an activity. Generally, artifacts are referred to, produced or maintained when an activity is performed. Examples of artifacts include: design documents; source code; and object code. Because artifacts are potentially manipulated by many software engineers when performing their activities, artifacts often require some associated access rights. Access rights ensure that artifacts are manipulated in accordance with the pre-conditions or post-conditions of an activity.
- Tools – Tools are external programs which are needed either to transform artifacts or to support inter-person communication (e.g. email, video conferencing program) which is seen as an important aspect of collaborative activities such as software processes [31].

The last three constituents of a process model from the bulleted list above are often referred to as *resources*.

In order to achieve reuse of process models, a PML also needs to support abstraction and modularisation. Through modularisation and abstraction, large process models for a particular project, for example, can be broken into a number of smaller process models (or modules). In turn, these smaller process models can be reused to form other process models for different projects.

In summary, the categories for PML modeling issues are:

i. Support for expressing both sequential and parallel activities and their constraints.
ii. Support for expressing input and output artifacts.
iii. Support for role representations.
iv. Support for expressing and invoking external tools.
v. Support for abstraction and modularisation of process models.

Enactment

In order to directly support the activities of software engineers, a process model needs to be enacted. Enactment of process models requires a PML that has executable semantics. Furthermore, as software processes often involve software engineers who may or may not be collocated, a PML also ought to support enactment of process models in a distributed environment.

Enactment of a process model raises an issue relating to resource allocations. Because software processes are highly dynamic, rarely can resources for a process model be completely specified ahead of time. For instance, the number of people assigned (as a resource) for a particular activity, and hence how many instances of a particular activity are created, must not be fixed since it will depend on the dynamic needs of a project. Therefore, it is desirable for a PML to support the dynamic allocation of resources. Here, the dynamic allocation of resources means that resources are allocated at the last moment, just as the activity is about to be started. Allowing dynamic allocation of resources as a feature of PML gives the process engineers the flexibility to consider the current needs of a particular project before deciding on the necessary resource allocation for a particular activity. As a consequence, because resources are allocated dynamically, enactment of a process model can commence even when resources have not yet been completely specified.

In summary, the categories for PML enactment issues are:

i. Support for enactment in a distributed environment.
ii. Support for dynamic allocation of resources.

Evolution

To handle its evolution in a controlled and integrated manner, a process model also needs to capture issues concerning the meta-process [1, 9, 10]. The meta-process is in charge of maintaining and evolving the process model according to specific and desirable rules and procedures. Therefore, the data manipulated by the meta-process are part of the process model itself.

Because enactment of a process model is typically long-lived and subjected to unpredictable changes (e.g. to cater for new needs arising from the current enactment), there is a need for the PML to provide a mechanism to allow the meta-process to be able to access the process model even though it is running. It has been suggested that reflection, a feature of a PML which allows enactable code to be manipulated as data, provides such a suitable mechanism [1, 9, 10]. With reflection, the meta-process can be modeled and enacted as part of the process model itself. As a result, evolution of the process model (and evolution of the meta-process itself) can be achieved dynamically, and be supported by the meta-process. It follows that while an enactable PML without a reflective facility can be used to support modeling and enacting of software processes, it may not be able to support an integrated process model consisting of both the software process and the meta-process, and to support dynamic changes to both during enactment.

In summary, the sole category for PML evolution issues is:

i. Support for reflection.

Evaluation

If a PML, through enactment of the process model, is being used to guide or enforce software engineering practice, it is vital that issues of importance are measured in some way so that evaluation of the process can take place. This is especially important to provide support for software process improvement. Thus, a PML ought to provide relevant software metrics, although little work is reported in the literature [32].

In summary, the sole category for PML evaluation issue is:

i. Support for collection of “enactment” data.

Human Dimension

Because software processes are carried out primarily by people, it is necessary that human dimension issues be considered. The human dimension can cover issues for the process engineers (or project managers) who create the process models and initiate enactment (e.g. facilitating the construction and comprehension of process models) as well as issues for software engineers who are subjected to process enactment (e.g. supporting software engineers at work).

In terms of the human dimension issues surrounding the process engineers who create the process model and initiate enactment, it is obviously desirable to have a process model which is simple and easy to understand. This places a requirement on a PML to be intuitive in its syntax and semantics. It is generally believed that this can be obtained to a certain extent by adopting a visual syntax and notation. The reason is that "pictures" are normally thought to more readily relate to the cognitive part of the human brain as compared to text. Employing visual notations in a PML helps create an easy to use yet expressive language, thus making a PML more acceptable and accessible. There are a number of visual PMLs discussed in the literature, as will be seen in the next section.

In terms of the issues surrounding the software engineers who are subjected to process enactment, it is desirable that enactment of a process model provides some form of awareness in terms of providing information about other parts of the model such as other users and other activities. In the literature, the importance of awareness has been established in the field of Computer Supported Cooperative Work (CSCW), a field of study which places emphasis on the nature of humans working together collaboratively to achieve a common goal as well as on the possibilities of technology to support and improve individual and group efficacy. Clearly, there is a need to include support for awareness as a feature of a PML since enactment of a process model also involves collaborative work similar to CSCW. The support for awareness seems increasingly relevant in line with the growing trends in the way software engineers work in geographically and temporally distributed locations (e.g. software designers in London, reviewers in Washington and programmers in New Delhi) and across organisational boundaries. There are two types of awareness a PML must support. They are:

- User awareness – User awareness is providing knowledge about other group members involved in the cooperative system. In general, having user awareness can often encourage informal interaction. Such informal interaction is normally useful if people are working on shared artifacts (as in a typical software process).
- Process awareness – Process awareness is providing knowledge about the tasks in their working contexts, for example in terms of what the previous task was, what the next task is and what needs to be done to move along as well as what resources are required. Typically, having process awareness is valuable as it gives a sense of where and how the pieces fit together into the whole picture. Additionally, process awareness should also make people aware of tasks not only involving themselves but also others. Such awareness may help improve the process – for instance, people can plan and anticipate their workloads as needed to meet the project deadline. One way to enhance process awareness is to provide support for visualisation of the process model. This seems to be a useful feature to have in a PML as software processes can be very complex and full of subtleties. Process visualisation can provide multiple views of the same process with different perspectives which, in turn, enhances human intuition about the tasks they are involved in.

Apart from supporting awareness and visualisation, there is also a need for a process model to be able to accommodate meetings as they are an important characteristic of software engineering. In fact, in the context of supporting software development over distributed locations, it is desirable for a process model to accommodate virtual meetings, that is, meetings that are held online. Accommodating virtual meetings could reduce costs if meetings would otherwise have to be held face to face. Thus, a PML needs to be able to specify virtual meetings as part of the process model.

In summary, the categories for PML human dimension issues are:

i. Support for visual notations.
ii. Support for user awareness.
iii. Support for process awareness.
iv. Support for process visualisation.
v. Support for virtual meetings.

3.0 ANALYSIS OF PMLs

This section provides a detailed analysis of existing PMLs using the characteristics of PMLs identified in the previous section. For each PML, this section presents: a brief description of the language; and an analysis of PML issues related to the modeling, the enactment, the evolution, the evaluation, and the human dimension.

3.1 SLANG

SLANG [3], a PML for the PSEE called SPADE, is based on Petri nets. The semantics of SLANG are defined by high-level Petri nets called Entity Relations (ER) nets. The main addition that ER nets add to conventional Petri nets is the ability to incorporate timing as the criterion to fire transitions.

In addition to the usual Petri nets syntax, SLANG provides an interface defined in terms of input and output *places* (as *entry* and *exit* points), input and output transitions (as *entry* and *exit* actions) as well as *shared places* to allow sharing of data, all of which are connected by a set of input and output arcs. In fact, an interface serves as a SLANG modularisation facility; an interface may be decomposed to show the overall internal SLANG net structure.

Besides the normal places and the shared places, SLANG also defines *user places*. Normal places and shared places represent place holders for tokens consisting of typed artifacts stored in an object oriented database, while user places represent place holders for tokens consisting of internal messages generated as a consequence of external events occurring within the PSEE.

In SLANG, transitions designate events. Transitions firing depends upon the availability of tokens and the defined guards (explained below). Additionally, timing information, such as the time interval within which an event may or must occur, can also be specified by associating time-stamps with tokens and time changes with actions (described below). In addition, SLANG also allows transitions to be textually augmented with scripts consisting of three parts:

i. A header containing an event's name and typed parameters which must match the types of the transition's input and output places.
ii. A guard containing a Boolean expression which checks the firing rules. A guard may be thought as the pre-condition for enabling a transition.
iii. A set of actions that performs some computation on the input tokens to produce some output tokens.

There are two types of transitions in SLANG: white transitions and black transitions. White transitions are similar to procedures in a general purpose programming language – they receive some input parameters (through the defined guards that need to be satisfied) and predefined statements are executed in sequence to produce some output parameters. Black transitions, unlike white transitions, allow invocation of external tools. The combinations of black transitions and user places allow SLANG to have some control over the events generated by external tools. This capability allows SLANG to support automation at external tool levels, for example, by detecting events such as the opening or the closing of external tools.

Analysis of SLANG

In terms of the modeling support, SLANG seems to provide support for most of the characteristics identified in Table 1 with the exception of the representation of roles. In SLANG, activities and their constraints are modeled as sets of actions associated with transitions. Tokens represent typed input/output artifacts which are stored in an object-oriented database. Being a Petri net based PML, SLANG naturally supports parallelism. Finally, modularisation and abstraction facilities are also supported in SLANG through interfaces.

In terms of enactment support, SLANG supports enactment in a distributed environment. However, SLANG does not directly support dynamic allocation of resources but process models (and resource allocation) can be evolved while enactment is taking place through SLANG's reflection facility. No support is provided for the collection of enactment data. As far as the human dimension support, SLANG only supports visual notations based on Petri nets (and augmented with textual scripts).

3.2 LIMBO and PATE

LIMBO and PATE are the two PMLs for the PSEE called OIKOS [23]. Because both LIMBO and PATE exploit the idea of coordination to support a software process as inspired by Linda [16], it is worth describing the basic idea from Linda.

In Linda, coordination is based on *tuples* and *tuple spaces*. A tuple consists of a set of variables or values. A tuple space can be viewed as a distributed shared memory into which tuples can be inserted or removed. Each tuple in a tuple space is produced by some executing thread and it remains in the tuple space until some other thread consumes it. When a thread requests specific tuples which do not exist, the thread may be suspended until those tuples are made available.

Borrowing from Linda, LIMBO and PATE support a software process by exploiting a reactive system based on threads (called *agents*) communicating using shared tuple spaces called blackboards.

LIMBO and PATE originated from Extended Shared Prolog, and adopt a rule-based approach to support the modeling and enacting of software processes. Using Ambriola's classification discussed earlier, LIMBO is the specification language whilst PATE is the implementation language. It is possible to obtain a PATE process model by successive refinement of a LIMBO specification. In the context of this paper, because LIMBO serves as a specification language for PATE, LIMBO will not be discussed further.

In PATE, a process model is constructed in terms of a hierarchy of agents. Each agent is connected to its own blackboard when the agent is activated. Agents react to the presence of tuples (mainly Prolog facts) on their blackboards by removing tuples, and inserting tuples into their own blackboards or other blackboards that they know of through a customisable service provided by the PSEE.

The behavior of an agent is defined by a *theory* consisting of a set of action and reaction patterns along with a sequential Prolog program (called the *Knowledge base)*. Each pattern defines a stimulus and response pair. The stimulus consists of a *Read guard* and an *In guard*. The response consists of a *Body* and a *Success set*. Optionally, a *Failure set* can also be specified.

A pattern can fire when the tuples (in terms of Prolog facts) in the agent's blackboard satisfy the read and the in guards; these tuples will be consumed and removed from the agent's blackboard. Whenever several patterns can fire, one is chosen non-deterministically and the specified actions are performed (i.e. the related pattern body and the sequential Prolog program will be executed). The execution of the pattern body and the sequential Prolog program is achieved in such a way that no side effects are allowed to the agent's blackboard whose pattern is fired (as this can only be done by the success set or the failure set).

The most common use of the sequential Prolog program in terms of supporting a software process is to invoke external tools to manipulate artifacts. Finally, depending on the outcome of the execution of both the pattern body and the sequential Prolog program, some tuples (i.e. a success or a failure set) will be inserted back onto the blackboard whose name is specified by that pattern. This sequence of pattern firing can then go on for other agents until the enactment is completed.

Blackboards can be dynamically created or destroyed during the course of enactment. Creation of a new blackboard can be achieved by an agent inserting an *activation goal* (i.e. Prolog facts) along with a *termination condition* (also Prolog facts) and a list of connected agents in its own blackboard. Destruction of a blackboard occurs when the success set or the failure set matches with the blackboard termination condition also expressed as Prolog facts, as a result of firing a pattern. In this case, all the tuples in the blackboard will disappear and all connection to agents will be aborted.

Analysis of LIMBO and PATE

In terms of modeling support, the modeling of activities and their constraints are indirectly supported by specifying the theory of each agent (i.e. the action and reaction patterns and the knowledge base). This can be achieved either from the LIMBO specification (and later refined to PATE) or directly in PATE. How role representation is supported in PATE is not clear. Tools can be invoked in pattern bodies or in the knowledge base. Artifacts are accessed in terms of identifiers through some standard services provided by the PSEE. Parallel activities can be readily supported because agents are effectively executing threads communicating using multiple tuple spaces. However, modularisation and abstraction of process models are not supported in PATE.

In terms of enactment support, a process model expressed in PATE can be enacted in a distributed environment. However, PATE does not support dynamic allocation of resources. In terms of evolution support, PATE does not support reflection. No support is provided for the collection of enactment data nor for the human dimension issues.

3.3 BM and PWI PML

Base Model (BM) and Process Wise Integrator Process Management Language (PWI PML) are the two PMLs for the PSEE called PADM [6]. BM adopts temporal logics semantics; PWI PML adopts object-oriented technology. Using Ambriola's classification, BM can be seen as a process specification language whilst PWI PML can be seen as a process implementation language.

Because BM and PWI PML are two compatible PMLs, it is possible to gradually refine the process model specified by BM into PWI PML. In doing so, a special tool called the BM stepper can be used to assist checking of the BM specification against the problem description and the refinement of the process model in PWI PML. Because this paper concentrates on enactable PMLs, discussion of BM will not be developed further as it mainly serves as a specification language for PWI PML.

PWI PML is an object-oriented PML. The primary construct for supporting the modeling and enacting of software processes is the *role*. The concept of a role in PWI PML carries a more subtle meaning than that defined earlier. PWI PML defines two types of roles: *User Roles a*nd *System Roles*. A user role corresponds to the identification of skills for performing a particular activity in a software process whilst a system role may correspond to some abstractions of an activity or a user role.

A software process model in PWI PML is a set of executing role instances connected by interactions (described below). A role is a subclass of the pre-defined PML Role class. Within the subclass the following properties must be specified for modeling and enacting software processes:

i. *Resources* describe the data objects (e.g. artifacts and tool definitions) belonging to the role. These data objects can be derived from some pre-defined classes in PWI PML.
ii. *Assocs* provides references to the communication channels linking one role object to another.
iii. *Actions* define the list of interactions and activities which are performed by the role. Actions may be thought of as sub-activities of a role. Each of these actions has a name, and is guarded by *when* conditions which must be satisfied before the action can be performed.
iv. *Categories* contain conditions to determine the start and stop conditions of a role.

Enactment is achieved by instantiating roles, and a role may also instantiate other roles. Each role instance has a separate thread of control with its own local data. Role instances communicate using message passing via typed one-way asynchronous communication channels.

Analysis of BM and PWI PML

In terms of modeling support, the modeling of activities and their constraints are supported using the specification language BM and later refined into PWI PML. In particular, activities can be abstracted as a sub-class of the Role class, with each sub-activity representing some actions properties in the role. Parallelism between activities is supported as each role has its own separate thread. The representation roles described in the previous section are represented as user roles and are considered as resources of the system roles. Similarly, artifacts and tools are also considered as resources of the system roles. Finally, modularisation and abstraction in PWI PML are supported by the role definitions.

In terms of enactment support, PWI PML supports enactment in a distributed environment. However, PWI PML does not seem to directly support dynamic allocation of resources although user roles can be bound to system roles dynamically. In terms of evolution support, PWI PML provides support for reflection. No support is provided in PWI PML for the collection of enactment data nor for the human dimension issues identified earlier.

3.4 MERLIN

MERLIN [20] is a Prolog-like PML. In MERLIN, the act of modeling a software process is assisted by entity relationship diagrams (a type of diagram which mainly depicts dependencies amongst artifacts) and state charts (a special type of state transition diagram which depicts the allowable transitions of an activity or an artifact from its creation to its completion along with the conditions under which a transition may occur). Based on the created entity relationship diagrams and state charts, a process model is then mapped to Prolog rules and facts as a *knowledge base* about that particular process. A special kind of fact is used to describe roles (*work_on* and *responsibilities* facts), artifacts and tools (*document* facts) as well as activities (*task rules*). Similar to MSL discussed in Section 3.1, enactment of a process model expressed by MERLIN is achieved by the PSEE runtime engine using a forward chaining mechanism to automate an activity that does not involve human intervention, and a backward chaining mechanism to select a particular activity for software engineers based on the role they play.

With the information gathered by the backward chaining mechanism, the MERLIN PSEE runtime engine incrementally builds a *work context* for the software engineers who perform the activity, in the form of a simplified entity relationship diagram which shows only the artifacts and tools necessary to complete the activity. In MERLIN PSEE, softwares engineer may interact with this diagram in a hypertext manner to perform their work.

Analysis of MERLIN

In terms of modeling support, the modeling of activities and their constraints, roles, artifacts and tools are supported by specialised PROLOG facts. Parallel activities are supported by the PSEE runtime engine. However, it is not clear how the modularisation and abstraction of process models are supported in MERLIN.

In terms of enactment support, the process model expressed by MERLIN can be enacted in a distributed environment. However, MERLIN does not support dynamic allocation of resources. In terms of evolution support, MERLIN does not support reflection. For evaluation support, no support is provided for the collection of enactment data. Concerning human dimension support, although MERLIN is a textual language, state charts and entity relationship diagrams can be used to help construct the MERLIN process model. MERLIN provides limited support for awareness. Process awareness is supported but not user awareness. The support for process awareness is achieved by the MERLIN PSEE runtime engine which automatically builds a work context from the specified Prolog facts (utilising the backward chaining mechanism) in terms of only giving a software engineer the necessary artifacts and tools needed to complete the activity. Finally, MERLIN provides no support for process visualisation.

3.5 SPELL

SPELL, the PML for the PSEE called EPOS [11], is an object-oriented PML derived from the Prolog programming language. In SPELL, the main support for software processes is provided by two pre-defined classes: *TaskEntity* and *DataEntity*. TaskEntity forms the root of the task type hierarchy whilst DataEntity forms the root of the data (or artifact) type hierarchy.

Every SPELL task type must be a subclass of TaskEntity which defines a number of predefined attributes that can be tailored to the needs of the process model. Among the important type level attributes within a task type are:

i. *Pre- and post-conditions*: The pre- and post-conditions attributes in a task type are divided into static and dynamic pre- and post-conditions. They are specified in first order predicate logic (as in Prolog). Static pre-conditions and post-conditions are constraints mainly used to build the network of tasks. This is accomplished by the runtime planner using forward and backward chaining. Dynamic pre-conditions and post-conditions are constraints asserted before and after task executions, and are used to dynamically trigger tasks.

ii. *Code*: The code attribute defines the steps that are performed when the task is executed. A task's code (specified in Prolog) is responsible for satisfying the dynamic post-conditions. When the code attribute is empty (i.e. not specified), the task type is assumed to be composite – that is, the task is not performed by a specific piece of code, but rather by executing subtasks (explained below).

iii. *Decomposition*: The decomposition attribute relates to the code attribute described earlier as it allows subtasks to be specified. In SPELL, the subtasks may also be a network of tasks. Like the parent task, the network of subtasks is also created by the runtime planner using the static pre- and post conditions discussed earlier.

iv. *Formal*: The formal attribute permits the specification of the input and output artifacts required for the task.

v. *Executor*: The executor attribute allows the tools used in the task to be specified.

vi. *Role*: The role attribute represents the role for the task.

In addition to these attributes, the TaskEntity class also defines a number of meta-level attributes and methods, essentially allowing further customisation of the TaskEntity class in terms of its execution. These meta-level attributes and methods will not be discussed here as they are mainly there to provide support for the reflection facility in SPELL. Nevertheless, one important aspect that can be specified at this level is triggers, which are special operations invoked before or after the occurrence of a method. Triggers specify the constraints defining when the trigger "codes" should be executed with respect to the method call. With triggers, various internal states of the task executions can be captured and modified if needed. SPELL also defines a family of types derived from the DataEntity class for specifying artifacts. Typical types used for software processes are a text type and a binary type which can be further specialised to other types (such as c-source and object-file).

For process enactment in SPELL, the runtime support system provided by the PSEE consists of two parts: the runtime planner and the runtime execution manager. As discussed earlier, the runtime planner generates a network of tasks from the static pre- and post-conditions by utilising forward and backward chaining similar to MSL; the top-level network of tasks must be generated by the runtime planner before enactment commences but the detailed level network of subtasks can be generated incrementally. The runtime manager executes the given task network (by executing the specified code attribute in the task type) and works closely with the runtime planner such that, when a composite task is encountered, the runtime manager invokes the runtime planner to detail out that composite task based on its static pre- and post-conditions. In this way, SPELL allows the detailed network of tasks to be generated only when it is needed.

Analysis of SPELL

In SPELL, support for modeling of activities is provided by inheriting the TaskEntity class and specifying the various pre-defined attributes discussed earlier. Parallel activities are supported in SPELL and are handled automatically by the runtime planner and the runtime manager. The activity constraints are defined by the static and dynamic pre- and post-conditions. Role and tool abstractions are supported through the role and executor attributes of the TaskEntity class. Artifacts are typed and stored in an object-oriented database called EPOS-DB. Modularisation and abstraction facilities are also supported through the code and decomposition attributes of the TaskEntity class.

In terms of enactment support, the process model expressed by SPELL can be enacted in a distributed environment. SPELL does not directly support dynamic allocation of resources but process models (and resource allocation) can be evolved while enactment is taking place through SPELL's reflection facility. No support is provided for the collection of enactment data nor for the human dimension issues identified earlier.

3.6 MASP/DL

Model for Assisted Software Process Description Language (MASP/DL) [7] is the PML for the PSEE called ALF. In MASP/DL, a software process model is modeled as a number of "fragments" called the *MASP descriptions*. Each MASP description models a software process as five components:

i. *The Object Model* describes the data model representing artifacts.
ii. *The Operator Model* represents an abstraction of the actual activities which a software engineer needs to perform in a software process, in terms of operator types. Operator types allow an individual activity to be described in terms of pre-conditions and post-conditions along with a definition of tools.
iii. *The Characteristics* specify a set of consistency constraints on the process state which are maintained by the runtime engine during the course of enactment; if they are violated, an exception condition is raised.
iv. *The Rule Model* defines some trigger reactions for predefined events that occur during process enactment. The events include database operations (e.g. read, write) and other user defined events.
v. *The Ordering Model* specifies the flow of control of the operators. Operators may be executed in parallel, alternatively or sequentially.

Because a MASP description is generic, it must be instantiated (and compiled to a special format called IMASP) before enactment can be achieved. Instantiation of an MASP description corresponds to the assignment of actual tools and artifacts in the operator and object models. It should be noted that a particular software process model can consist of a number of MASP descriptions (or fragments) with each description defining a number of related activities. So, before enactment can be achieved, each MASP description must be instantiated (and compiled) individually. In this manner, instantiation and enactment may interleave so that the part of the software process that has already been enacted may be taken into account to instantiate a further part.

Analysis of MASP/DL

In terms of modeling support, the modeling of activities and their constraints in MASP/DL are supported in the operator and rule models. Parallelism of activities is supported in the ordering model. The representation of roles is not supported at the PML level as this is achieved at the PSEE level. Artifacts are described in the object model. Tools are described in the rule model along with the defined triggers. Finally, modularisation and abstraction are supported by the MASP descriptions.

For enactment support, MASP/DL supports enactment in a distributed environment. However, MASP/DL does not really support dynamic allocation of resources. This is because resources for each activity defined in a particular

MASP/DL description must be allocated before enactment of that MASP/DL description can commence, although such allocations can be made based on the outcome of the instantiation and enactment of the earlier parts. In terms of evolution support, MASP/DL does not support reflection. Also, no support is provided for the collection of enactment data nor for the human dimension issues identified earlier.

3.7 ADELE and TEMPO

ADELE and TEMPO [5] are two PMLs for the PSEE called ADELE-TEMPO, which is a commercial database product. To understand how the modeling and enacting of software processes is supported by the first PML, ADELE, it is necessary to understand the basic architecture of its PSEE. The PSEE consists of a centralised database which provides long transaction support for artifacts involved in the software development project. Each participating software engineer in the project is provided with their own Work Environment by the PSEE, where artifacts (organised in terms of files and directories) can be checked out. A software process is then modeled in ADELE as a set of defined events and triggers on the artifacts in the Work Environment.

ADELE events and triggers are based on the event-action-condition (ECA) rules. Triggers may fire on any of four events – *PRE*, *POST*, *ERROR* and *AFTER* – with respect to a particular activity. In each of the events, some actions (e.g. in terms of some database transactions or invoking of external tools) can be specified. PRE triggers are evaluated at the beginning of an execution of an activity. If successful, the specified action is executed. Similarly, POST triggers are evaluated after the completion of an activity. ERROR triggers are considered when a transaction fails and AFTER triggers are applied after a transaction succeeds.

According to Belkhatir [5], because ADELE is too low level and difficult to understand, the second PML called TEMPO was developed. TEMPO defines a process model based on the concept of role and connection (described below). Although a user role is supported, the concept of role in TEMPO carries a subtler meaning than the one defined earlier. A role allows redefinition of behavioral properties of an artifact based on the defined work context, that is, the operations that can be done on that artifact and the rules that control these operations. Roles, in turn, are connected to other roles through a defined connection, essentially the synchronisation protocol based on temporal-event-condition-action (TECA) rules (i.e. extended ECA rules with timing dependencies).

Analysis of ADELE and TEMPO

In terms of modeling support, the modeling of activities and their constraints is indirectly supported in TEMPO by defining roles and their connections. In ADELE, modeling of activities and their constraints are also indirectly supported although at a very low level in terms of the ECA rules. Tools can be invoked from the action part of the TECA rules in TEMPO (and ECA rules in ADELE) as a result of a condition being fulfilled. In both ADELE and TEMPO, artifacts are defined as objects in the ADELE database. In TEMPO, parallel activities are supported by connections defined by TECA rules (and ECA rules in ADELE). Finally, modularisation and abstraction are only supported by TEMPO (through the abstraction of roles).

In terms of enactment support, process models expressed by TEMPO and ADELE can be readily enacted in a distributed environment. However, neither TEMPO nor ADELE supports dynamic allocation of resources. In terms of evolution support, TEMPO and ADELE do not support reflection. For evaluation support, no support is provided for the collection of enactment data. As far as the support for the human dimension, TEMPO provides (limited) support for process awareness. In TEMPO, process awareness is achieved by giving the work context of a task. No support is provided for user awareness, process visualisation, and virtual meetings identified earlier.

3.8 APPL/A

APPL/A [27]**,** the PML for the PSEE called Arcadia. Being based on ADA, APPL/A inherits many features from that language including its type system, module definition style (package), and task communication paradigm (rendezvous). To support a software process, APPL/A extends the ADA programming language with *shared persistence relations*, *concurrent triggers on relation operations*, *enforceable predicates on relations*, and *transaction-like statements*.

Relations are syntactically similar to ADA package definitions and package bodies. Within a relation, persistent storage of data may be defined. Triggers are similar to ADA tasks and hence are capable of handling multiple threads of control. Unlike ADA tasks, triggers automatically react to events related to operations on the data defined in a relation. Enforced predicates are Boolean expressions which act as post conditions on the operation of a relation; no operations may violate the enforced predicate. Transaction-like statements control access to relations and may affect the enforcement of predicates.

Analysis of APPL/A

In terms of the modeling support, APPL/A does not provide support for all of the PML characteristics identified in Table 1. Modeling of activities and their constraints are supported in APPL/A via relations with enforceable predicates. Parallel activities are supported by exploiting triggers. No direct support is provided for role representations, artifacts, and tools; rather these can be represented using the ADA type mechanism. Abstraction and modularisation in APPL/A is mainly based on the ADA procedures and packages.

For enactment support, it is not clear whether or not APPL/A supports enactment in a distributed environment. Additionally, APPL/A does not support dynamic allocation of resources. In terms of evolution support, APPL/A does not support reflection, only offline process evolution as recompilation is necessary. Finally, no support is provided for the collection of enactment data nor for the human dimension issues identified earlier.

3.9 Dynamic Task Nets

Dynamic Task Nets [17], the visual PML for the PSEE called Dynamite. Dynamic Task Nets describe a software process as graphs consisting of nodes representing tasks connected together with arcs (called *relations*). There are three types of relations:

i. *Control-flow relations* impose an acyclic ordering of the activities to be enacted.
ii. *Data flow relations* are used for data (mainly artifacts) transmitted between connected tasks.
iii. *Feedback flow relations* are used to enable feedback from a successor task back to its predecessor.

There is also another relation supported by Dynamic Task Nets called *successor relations*. Unlike the three relations described above, successor relations refer to nodes rather than arcs. When a task is augmented with a successor relation, that task is said to have multiple versions. What this means is that when such a task has to be reactivated (e.g. as a result of feedback relations), a new task version may be created depending on whether the previous version of the task has completed or not. If the task has already completed, a new version of the task is created requiring a new assignment of a software engineer. If the task has not yet been completed, the runtime system automatically updates the tasks with the new version of the artifacts.

In Dynamic Task Nets, tasks and their corresponding relations can be defined dynamically. The behavior of each individual task (called a *task net)* can be customised in the sense that a task can execute even when its predecessor tasks have not completed (called *simultaneous concurrent engineering*) or when certain input artifacts are available. However, the task completion can only be allowed if predecessor tasks have already been completed. In Dynamic Task Nets, this customisation is achieved by modifying the enactment conditions defined in the PROGRESS specification, which itself is an executable graph rewriting system that Dynamic Task Nets map to for achieving enactment.

Analysis of Dynamic Task Nets

In terms of modeling support, the modeling of activities and their constraints are supported by the task nets and their relations, and can be customised at the PROGRESS specification level. The representation of roles is not directly supported at the PML level. Artifacts are considered as part of the input and output interface to a task. Tools abstraction is not supported directly although can be defined at the PROGRESS specification level. Finally, modularisation and abstraction are supported by the nodes themselves in the sense that each task net can be decomposed into other smaller task nets (although Dynamic Task Nets does not visually differentiate between decomposable task nets and atomic ones).

In terms of enactment support, Dynamic Task Nets support enactment in a distributed environment. Because the creation of task nets in Dynamic Task Nets is dynamic, it naturally supports dynamic allocation of resources. In terms of evolution support, Dynamic Task Nets does not support reflection. No support is provided for the collection of enactment data. In terms of human dimension support, Dynamic Task Nets employs a visual PML. No other support is provided to address the other human dimension issues identified earlier.

3.10 LATIN

Language to tolerate Inconsistencies (LATIN) [13] is a PML for the PSEE called SENTINEL. LATIN describes a software process as a global part and a set of task types. A global part contains global variables, a global invariant (described below), the declaration of the task types that will be instantiated during enactment, and the description of the *main task*. When the process enactment starts, an interpreter for the main task is created. All other tasks are instantiated by the main task or, in turn, by previously instantiated tasks (i.e. their predecessors).

A task type is composed of the following parts:

i. *A header* defines the name of the task type along with its parameter list. Instantiation of a task type consists of assigning the initial state of the task instance.
ii. *An import section* defines all variables imported from other task types.
iii. *A declaration section* declares the local types and variables. The basic data types in LATIN can be integer, real, string, Boolean, enumerated, as well as user defined data types such as records and sets.
iv. *An export section* lists the names of all the variables exported to other tasks.
v. *An init section* lists all initial values assigned in terms of resource assignments during instantiation.
vi. *A set of transitions* which govern the actual enactment of task type instances (described below). Transitions can be associated with invoking of external tools.
vii. *A set of invariants* which serves as special conditions that must hold true in any state of the activity described by the task type.

The behavior of a task type is described by transitions. Transitions are further characterised by a precondition, called an *ENTRY*, and a body. The ENTRY defines the conditions which fire the corresponding transition whilst a body defines *actions* (e.g. invoking a tool) and *value assignments* (e.g. updating variables) through an *EXIT* clause.

LATIN actually offers two types of transitions: *normal transitions* and *exported transitions*. A normal transition is automatically executed by the runtime system as soon as an ENTRY evaluates to true. An exported transition is executed upon the request from the user even if its ENTRY evaluates to false. In such a case, a transition is said to *fire illegally*. The outcome of such a firing is that enactment can now be allowed to deviate from the specified process model, hence introducing *inconsistencies*. In LATIN, enactment can continue as long as the invariants of that task type still hold. But if one of the invariants is violated, enactment is suspended and a *reconciliation activity* is started to allow reconciliation of the actual process and the process model.

Analysis of LATIN

In terms of modeling support, the modeling of activities is supported as a set of task types. The constraints for each activity are specified by the transitions. Task types may execute in parallel. The representation of roles is not directly supported at the PML level. Artifacts can be supported by the user defined types. Tools can be specified and invoked in the transition's action. Finally, modularisation and abstraction are supported by each individual task type.

In terms of enactment support, LATIN supports enactment in a distributed environment. Although task types can be dynamically instantiated during enactment by the main task or the previously instantiated task, LATIN does not support dynamic allocation of resources. This is because resources need to be assigned to the task types as part of their init section before the overall enactment commences as LATIN does not provide any feature which allows assignment of resources while enactment is taking place. As far as evolution support, LATIN does not support reflection. For evaluation support, no support is provided in LATIN for the collection of enactment data. Finally, no support is provided for the human dimension issues identified earlier.

3.11 JIL

JIL [28] is a PML derived from the authors' experiences in developing APPL/A [27]. The main construct in JIL is the *step*. A JIL step represents a step in a software process, that is, a task which a software engineer or a tool is expected to perform. A JIL process model can be viewed as a composition of JIL steps. The elements that constitute a JIL step include:

i. *Object and declarations section* consists of the declaration of artifacts used in the step (consisting of ADA-like types).
ii. *Resource Requirements section* specifies the resources needed by the step, including people, software and hardware.

iii. *Sub-steps set section* provides a list of sub-steps that contribute to the realisation of the step.
iv. *Proactive control specification section* defines the order in which sub-steps may be enacted. This is achieved through special JIL keywords such as ORDERED, UNORDERED and PARALLEL.
v. *Reactive control specification* of the conditions or events in response to which sub-steps are to be executed. This is specified through special JIL keywords such as REACT.
vi. *Pre-conditions, constraints and post-conditions section*: A set of artifact consistency conditions that must be satisfied prior to, during, and subsequent to the execution of the step.
vii. *Exception handler section*: A set of exception handlers for local exceptions, including handlers for artifact consistency violations (e.g. pre-condition violations).

Apart from local exception handlers which allow a process to react within its own scope, JIL also supports global exception handlers. Global exception handlers allow a process to react to an exception in another process by treating such exceptions as events which can be handled directly by the reactive control specification.

Analysis of JIL

In terms of modeling support, the modeling of activities is supported by a composition of JIL steps. The constraints for each step can be specified as pre- and post-conditions. The step ordering constraints (e.g. parallel activities) can be supported using specialised constructs (e.g. *ORDERED, UNORDERED, PARALLEL*). The representation of roles is not directly supported at the PML level. Artifacts can be supported by the ADA-like types; JIL also allows artifact consistency to be checked as pre- or post-conditions of a step. Tools can be invoked directly through the reactive control specification. Finally, modularisation and abstraction are supported by the JIL step.

In terms of enactment support, it is not clear whether or not JIL supports enactment in a distributed environment. Also, JIL does not support dynamic resource allocation. As far as evolution support is concerned, no support is provided for reflection. No support is provided in JIL for the collection of enactment data nor for the human dimension issues identified earlier.

3.12 Little JIL

Little JIL [29] is a PML based on JIL. Unlike JIL, Little JIL is a visual PML. Like JIL, Little JIL maintains the notion of *a step* from JIL.

A process model in Little JIL can be viewed as a tree of steps whose leaves represent the smallest specified units of work and whose structure represents the way in which this work will be coordinated. As shown in Fig. 1, the Little JIL step notation has a number of components including:

i. *Step name section*: Every step must be given a name.
ii. *Interface badge section*: Resources needed in the step are carried through this badge. In Little JIL, resources include the assignment of software engineers, permissions to use the tools, as well as artifacts. Resources are also typed and always associated with some access rights.
iii. *Pre-requisite and post-requisite badge section*: Here, pre-conditions and post-conditions for the step are specified.
iv. *Control-flow badge section*: Little JIL allows four types of control-flow to be specified in a step. They are *sequential*, *parallel*, *choice* and *try*. Sequential and parallel control-flows allow sequential and parallel steps to be specified. Choice control-flow allows some choices of alternative steps to be specified in a step. Try control-flow is associated with handler badges (described below) to allow an exception to be caught.
v. *Handler badge section*: Handler badges are used to indicate and fix exceptional errors during enactment. In Little JIL, exceptions are passed up the process model tree until a matching handler is found.
vi. *Reaction badge section*: Reaction badges are a form of reactive control similar to JIL. A reaction badge is always associated with a message which is generated in response to some events. Because a message is global in scope, any execution step can receive the message and act accordingly if matches are found.

In terms of its enactment, a step goes through several states. Normally, a step is *posted* when assigned to a software engineer, then it progresses to a *started* state, and eventually it will be in a *completed* state. If a step fails to be started as a result of resource exceptions being thrown, a step may be *retracted* (and potentially reposted) or *terminated* with exception.

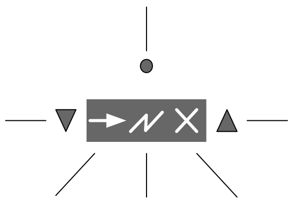

Fig. 1: Little JIL Step Notation

Analysis of LITTLE JIL

In terms of modeling support, the modeling of activities is supported by the Little JIL steps. The constraints for each step can be specified in the pre-requisite and post-requisite badges. Parallel activities can be defined by selecting the proper control-flow badge. The representation of roles is not directly supported at the PML level. In Little JIL, artifacts are typed and have associated access rights. Like artifacts, tools abstraction is also associated with an interface badge. Finally, Little JIL does not support modularisation and abstraction of the process models.

In terms of enactment support, Little JIL supports enactment in a distributed environment. However, Little JIL does not seem to support dynamic resource allocation, although the authors claim such support is provided (discussed below). In terms of evolution support, Little JIL does not support reflection. No support is provided in Little JIL for the collection of enactment data. In terms of human dimension support, Little JIL employs visual notation. No support is provided for process awareness, user awareness and process visualisation.

3.13 CSPL

CSPL [8] is a PML that adopts an ADA95-like syntax. Being based on ADA95, CSPL inherits many features from that language including its type system, module definition style (package), and task communication mechanism. Additionally, CSPL adds a number of predefined types and extensions to enable the modeling of software processes which include:

i. *Event* type and *inform* statements: The event type allows description of an event status of an activity (e.g. approved, completed). The value of an event derived from event type can be asynchronously assigned by CSPL inform statements.
ii. *Doc* type: Doc Type, the base type of all object types in CSPL, allows the description of artifacts and their associated attributes, which can be extended by inheritance.
iii. Work assignment statements are CSPL statements which allow activities, tools and roles to be assigned to one or more software engineers.
iv. Communication related statements allow synchronisation and ordering of tasks with other tasks, similar to the ADA95 rendezvous.
v. Program Units allow assignment of a human to a role (through a *Role Unit*), assignment of an actual tool to a tool (through a *Tool Unit*), and description of dependencies amongst artifacts (through a *Relation Unit*).

To support enactment, the CSPL compiler translates the process model expressed in CSPL into a UNIX shell script.

Analysis of CSPL

In terms of modeling support, the modeling of activities and their constraints are supported by the ADA95-like task specification. Parallel activities are supported through the communication related statements essentially ordering the tasks specified in CSPL. The representation of roles is supported by the role unit. Artifacts can be supported by the ADA-like types. Tools can be defined through the tool unit. Finally, modularisation and abstraction are supported by utilising package specification.

In terms of enactment support, CSPL supports enactment in a distributed environment. However, CSPL does not support dynamic resource allocation. In terms of evolution support, CSPL does not provide support for reflection –

only offline process evolution is supported. For evaluation support, no support is provided for the collection of enactment data. In terms of the human dimension, no support is provided for the issues identified earlier.

3.14 APEL

APEL [14] is a visual PML. The central construct in APEL is an *activity* of which there are two types: an *activity* representing a task for an individual; and a *multi-instance activity* representing a task for a group of people. Visually, an activity provides an interface to define input and output artifacts as well as the roles involved in a particular activity. Artifacts, activities and roles are typed and they are defined in a separate view using Object Modeling Techniques (OMT) diagrams, essentially class diagrams with some defined relationships (e.g. is-a or has-a). The various states which artifacts and activities go through during enactment can also be represented using state transition diagrams.

In APEL, a process model is composed of a set of activities connected together by *control-flow* and *data flow* arcs as well as *And* and *Or* connectors which carry the usual semantics. Activities can be decomposed until atomic activities are reached. To achieve process enactment, APEL relies on the concepts of *event* and *event capture* which can be defined on activities or artifacts. An event and event capture are defined by pairs comprising an event definition and a logical expression. An event is captured by an activity or an artifact when it matches the event definition and the logical expression is true. All events in APEL are broadcast and they are generated automatically.

The notable feature of APEL is that activities and their sub-activities, as well as the flow of artifacts, are shown to the user during enactment (through *a desktop paradigm*) in order to give the sense of awareness (discussed below) about other activities. In addition, the user may also interact with the desktop paradigm to perform the activity. Finally, unlike other PMLs, APEL also supports measurement of the process model by employing the Goal Question Metric Model [4], essentially consisting of self-defined goals, questions related to the process models achieving those goals, and metrics to quantify such questions.

Analysis of APEL

In terms of modeling support, the modeling of activities and their constraints is supported in APEL by the activities and their events and event captures. Parallel activities can be defined by And and Or connectors. Roles, artifacts and tools are typed and also expressed as part of the process models using OMT diagrams. Finally, modularisation and abstraction are supported as activities that can be further decomposed into sub-activities.

For enactment support, APEL supports enactment in a distributed environment. No support is provided for dynamic resource allocation. In terms of evolution support, APEL does not support reflection. Process evolution can only be achieved offline; this is assisted by a process state server which maintains the state of a process model's enactment. In terms of evaluation support, APEL supports the collection of enactment data through adaptation of the Goal Question Metric Model [4]. In terms of the human dimension, APEL provides (limited) support for awareness. Process awareness is supported but not user awareness. Process awareness is achieved by giving the work context of the overall activity and its sub-activities along with the flow of artifacts. Likewise, process visualisation can also be achieved in the same way. However, no support is provided for virtual meetings.

3.15 PROMENADE

Process-oriented Modeling and Enactment of Software Developments (PROMENADE) [24, 30] is a PML derived from the Unified Modeling Language [25], a language for supporting the object-oriented analysis and design of software systems. In PROMENADE, software processes are modeled using predefined classes (called the PROMENADE *reference model*) consisting of:

i. *Document Class* represents artifacts involved in the software development.
ii. *Communication Class* represents any document used for communication.
iii. *Task Class* represents an activity in a software process.
iv. *Agent Class* represents an entity playing an active part in a software process.
v. *Tool Class* represents any entity implemented through a software tool.
vi. *Resource Class* represents any supplementary help provided during enactment of a software process (e.g. an online tutorial). It should be noted that the word resource in this case carries a different meaning to that used earlier.
vii. *Role Class* represents identification of the skills of software engineers.

In a process model, the connections between instances of these classes are expressed using UML association relationships. In PROMENADE, the most important class is the task class. The task class may be customised to include the definition of shell scripts, the definition of task parameters consisting of input and output artifacts and the definition of task pre- and post-conditions. A task class may also be further broken down to sub-classes consisting of other task classes, either by aggregation or composition.

PROMENADE has not yet provided support for enactment as work is still on-going to provide additional language extensions. Nevertheless, process enactment is planned in PROMENADE by utilising its support for *precedence relationships*, which are textually expressed in each task class in the process model using a variation of the UML Object Constraint Language (OCL). Using the precedence relationships, the ordering of tasks can be defined to allow enactment of some actions according to a plan, defined in terms of pre-conditions and post-conditions, and connections to other task classes. *Strong* precedence dictates that the current task must successfully complete before the following task can be started. *Weak* precedence allows following tasks to start even if the current task has not yet been completed. PROMENADE is also being extended to provide support for reactive control-flows that is, enactment of some actions in response to events.

Analysis of PROMENADE

In terms of modeling support, the modeling of activities is supported by the class diagrams and their association relationship. The constraints for each activity are specified in the class diagram as pre- and post-conditions expressed by using a variation of the UML Object Constraint Language. Parallelism can be expressed in PROMENADE by specifying the precedence relationship in each task class. The representation of roles and artifacts are supported by the role and artifact classes respectively. The abstraction of tools is supported by a tool class and an agent class. Tools also can be invoked directly through the shell script defined in the task class. Finally, modularisation and abstraction are supported by aggregations and composition.

In terms of enactment support, it is not clear whether PROMENADE will support enactment in a distributed environment, and dynamic allocation of resources. Concerning evolution, the authors claim that PROMENADE will adopt reflection. No support is provided in PROMENADE for the collection of enactment data. In terms of human dimension support, apart from being a visual PML, no other support is provided.

4.0 DISCUSSION

Based on the analysis of PMLs given in the previous section, this section summarises the description of existing PMLs. Although only enactable PMLs are considered, some of the categorisations of PMLs (e.g. modeling support, evaluation support) should also be applicable to non-enactable PMLs, although this is not investigated further here.

Table 2 presents the digest of the analysis of PMLs discussed earlier. The table highlights those features that are common to PMLs, and those that are not, and therefore identifies areas for research into PMLs that address new combinations of features which can be explored further.

Referring to Table 2, it can be seen that existing PMLs are deficient in a number of the identified areas: human dimension issues; dynamic allocation of resources; collection of enactment data; and support for reflection. These deficiencies will be discussed next.

The first perceived deficiency in the surveyed PMLs is in terms of providing support for human dimension issues. As identified in Table 2, issues in terms of the support for visual notations, user awareness, process awareness, process visualisation, and virtual meetings seems to be neglected by most of the surveyed PMLs. Modeling and enacting of software processes requires much human intervention during its lifecycle – for example, process engineers (e.g. project managers) model the activities within a software process for enactment, and software engineers perform the activities during enactment. Therefore, it seems appropriate to place emphasis on the importance of human dimension issues.

The second perceived deficiency in the surveyed PMLs is in terms of the support for dynamic allocation of resources. The motivation behind supporting dynamic allocation as a feature of a PML and its underlying semantics is to ensure that resources are allocated based on the dynamic needs of a particular project, and as late as when the activity is about to be started. In this way, the process engineer can be given flexibility to allocate resources based on the current situation.

Table 2: Analysis of PMLs

LEGENDS Implemented feature √ Non supported feature X Not enough information		**SLANG**	**LIMBO & PATE**	**BM & PWI PML**	**MERLIN**	**SPELL**	**MASP / DL**	**ADELE & TEMPO**	**APPL / A**	**DYNAMITE**	**LATIN**	**JIL**	**LITTLE JIL**	**CSPL**	**APEL**	**PROMENADE**
Modeling Support	Sequential and parallel activities as well as their constraints	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	Input and output artifacts	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	Role representations	X	X	√	√	√	X	√	X	X	X	√	√	√	√	√
	External tools	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	Abstraction and modularisation	√	X	√		√	√	√	√	√	√	√	X	√	√	√
Enactment Support	Enactment in a distributed environment	√	√	√	√	√	√	√		√	√		√	√	√	
	Dynamic allocation of resources		X		X		X	X	X	√	X	X	X	X	X	X
Evolution Support	Reflection	√	X	√	X	√	X	X	X	X	X	X	X	X	X	X
Evaluation Support	Collection of enactment data	X	X	X	X	X	X	X	X	X	X	X	X	X	√	X
Human Dimension Support	Visual notations	√	X	X	X	X	X	X	X	√	X	X	√	X	√	√
	User awareness	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	Process awareness	X	X	X	√	X	X	√	X	X	X	X	X	X	√	X
	Process visualisation	X	X	X	X	X	X	X	X	X	X	X	X	X	√	X
	Virtual meetings	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

The third perceived deficiency in the surveyed PMLs is in terms of the support for the collection of enactment data. Only APEL provides this support, that is, based on the Goal Question Metrics [4]. By supporting collection of enactment data, systematic and objective evaluation of a particular process model can be made. In turn, this evaluation could be used as "indicators" to support process improvement.

Finally, the fourth perceived deficiency in the existing PMLs is in terms of providing support for reflection. The main reason that reflection is required as a feature of PML is to support evolution of a process model through a meta-process [1, 9, 10]. In doing so, enactment of the existing activities must not be affected. With reflection, the enacting process model can be accessed as data to be modified by the meta-process, hence allowing the evolution of a process model to occur while enactment is taking place.

The above analysis has identified a novel combination of features which existing PMLs do not provide, thus identifying an area for new PML research. This combination of features includes the support for:

- Visual notations
- User awareness
- Process awareness
- Process visualisation
- Virtual meetings
- Dynamic allocation of resources
- Collection of enactment data
- Reflection

To address the support for the above features, we are investigating a new visual PML called the Virtual Reality Process Modeling Language (VRPML) [32-35]. The main features of the language are that it exploits visual notations, integrates with a virtual environment in order to address user awareness, process awareness, and visualisation issues as well as supports dynamic allocation of resources by manipulating its enactment model. As far as implementation is concerned, the first version of the language definition has been completed (described in [34]), and the prototype implementation is still under development.

5.0 CONCLUSION

In conclusion, this paper has presented a critical analysis of existing PMLs by identifying each language's strong points and weaknesses. Hopefully, this analysis forms a useful guideline for the future design of PMLs.

REFERENCES

[1] V. Ambriola, R. Conradi, and A. Fuggetta, "Assessing Process-Centered Software Engineering Environments". *ACM Transactions on Software Engineering and Methodology*, *6* (3), 1998, pp. 283-328.

[2] S. Arbaoui, J. Lonchamp, and C. Montangero, "The Human Dimension of the Software Process", in J. C. Derniame, B. A. Kaba, and D. Wastell (Eds.). *Software Process: Principles, Methodology and Technology*, LNCS Vol. 1500, Springer, 1999, pp. 165-196.

[3] S. Bandinelli, A. Fuggetta, C. Ghezzi, and L. Lavazza, "SPADE: An Environment for Software Process Analysis, Design and Enactment", in A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.), *Software Process Modeling and Technology*, pp. 223-247, Research Studies Press, Taunton, England, 1994.

[4] V. R. Basili, and H. D. Rombach, "The TAME Approach: Towards Improvement-Oriented Software Environments". *IEEE Transactions on Software Engineering*, 14 (6). pp. 758-773.

[5] N. Belkhatir, J. Estublier, and W. Melo, "ADELE-TEMPO: An Environment to Support Process Modelling and Enaction", in A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.), *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, pp. 187-222.

[6] F. Bruynooghe, R. M. Greenwood, I. Robertson, J. Sa, and B. C. Warboys, "PADM: Towards a Total Process Modeling System", in A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.), *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, pp. 293-334.

[7] G. Canals, N. Boudjlida, J.C. Derniame, C. Godart, and J. Lonchamp, "ALF: A Framework for Building Process-Centred Software Engineering Environments", in A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.) *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, pp. 153-185.

[8] J. J. Chen, "CSPL: An Ada95-Like, Unix-Based Process Environment". *IEEE Transactions on Software Engineering*, 23 (3), 1997, pp. 171-184.

[9] R. Conradi, C. Fernstrom, and A. Fuggetta, "A Conceptual Framework for Evolving Software Processes". *ACM SIGSOFT Software Engineering Notes*, 18 (4). pp. 26-35.

[10] R. Conradi, C. Fernstrom, A. Fuggetta, and R. Snowdon, "Towards a Refence Framework for Process Concepts", in *Proc. of the 2nd European Workshop on Software Process Technology*, Trondheim, Norway, 1992, LNCS Vol. 635, Springer, pp. 3-17.

[11] R. Conradi, M. Hagaseth, J. Larsen, M. N. Nguyen, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri, and C. Liu, "EPOS: Object-Oriented Cooperative Process Modelling", in A. Finkelstein, J. Kramer, and B. Nuseibeh (Eds.), *Software Process Modeling and Technology*, Research Studies Press, Taunton, England, 1994, pp. 33-69.

[12] R. Conradi, and M. L. Jaccheri, “Process Modelling Languages”, in J. C. Derniame, B. A. Kaba, and D. Wastell (Eds.), *Software Process: Principles, Methodology and Technology*, LNCS Vol. 1500, Springer, 1999, pp. 27-52.

[13] G. Cugola, E. D. Nitto, C. Ghezzi, and M. Mantione, “How to Deal with Deviations During Process Model Enactment”, in *Proc. of the 17th Inl. Conf. on Software Engineering*, Seattle, Washington, 1995, IEEE CS Press, pp. 265-273.

[14] S. Dami, J. Estublier, and M. Amiour, “APEL: A Graphical Yet Executable Formalism for Process Modeling”. *Automated Software Engineering*, 5 (1), 1998, pp. 61-96.

[15] J. C. Derniame, B. A. Kaba, and B. C. Warboys, “The Software Process: Modelling and Technology”, in J. C. Derniame, B. A. Kaba, and D. Wastell (Eds.). *Software Process: Principles, Methodology and Technology*, LNCS Vol. 1500, Springer, 1999, pp. 1-13.

[16] D. Gelernter, “Generative Communication in Linda”. *ACM Transactions on Programming Languages and Systems*, 7 (1), 1985, pp. 80-112.

[17] P. Heiman, G. Joeris, and C. A. Krapp, “DYNAMITE: Dynamic Task Nets for Software Process Management”, in *Proc. of the 18th Intl. Conf. on Software Engineering*, Berlin, Germany, 1996. IEEE CS Press, pp. 331-341.

[18] K. E. Huff, “Software Process Modeling”, in A. Fuggetta, and A. Wolf (Eds.), *Trends in Software Process*, John Wiley & Sons, 1996, pp. 1-24.

[19] M. L. Jaccheri, R. Conradi, and B. H Drynes, “Software Process Technology and Software Organisations”, in *Proc. of the 7th European Workshop on Software Process*, (Kaprun, Austria, February 2000), LNCS Vol. 1780, Springer, pp. 96-108.

[20] G. Junkermann, B. Peuschel, W. Schafer, and S. Wolf, “MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment”, in A. Finkelstein, J. Kramer, and B. Nuseibeh (Eds.), *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, pp. 103-129.

[21] C. Liu, and R. Conradi, “Process Modeling Paradigms: An Evaluation”, in *Proc. of the 1st European Workshop on Software Process Modeling*, Milano, Italy, 1991, Italian National Association for Computer Science, pp. 39-52.

[22] J. Lonchamp, “An Assessment Exercise”, in A. Finkelstein, J. Kramer, and B. Nuseibeh (Eds.), *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, pp. 335-356.

[23] C. Montangero, and V. Ambriola, “OIKOS: Constructing Process-centred SDEs”, in A. Finkelstein, J. Kramer, and B. Nuseibeh (Eds.), *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994.

[24] J. M Ribo, and X. Franch, “PROMENADE: A PML Intended to Enhance Standarization, Expressiveness and Modularity in Software Process Modeling - Research Report LSI-34-R”. *Llenguates I Sistemes Informatics*, Politechnical of Catolonia, Spain, 2000.

[25] J. Rumbaugh, I. Jacobson, and G. Booch, *The UML Reference Manual*. Addison Wesley, 1999.

[26] I. Sommerville, and T. Rodden, *Human, Social and Organisational Influences on the Software Process*, in A. Fuggetta, and A. Wolf (Eds.), Trends in Software Process, John Wiley & Sons, 1996, pp. 89-108.

[27] S. Sutton Jr., D. Heimbigner, and L. J. Osterweil, “APPL/A: A Language for Software Process Programming”. *ACM Transactions on Software Engineering and Methodology*, 4 (3), 1995, pp. 221-286.

[28] S. Sutton Jr., and L. J. Osterweil, "The Design of a Next-Generation Process Language", in *Proc. of the Joint 6th European Software Engineering Conf. and the 5th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, 1997, LNCS Vol. 1301, Springer, pp. 142-158.

[29] A. Wise, "Little JIL 1.0 Language Report - Technical Report 98-24". Dept. of Computer Science, Univ. of Massachusetts at Amherst, April 1998.

[30] X. Franch and J. M. Ribo, "A UML-Based Approach to Enhance Reuse within Process Technology", in *Proc. of the 9th European Workshop on Software Process Technology*, LNCS Vol. 2786, Helsinki, Finland, 2003, Springer, pp. 74-93.

[31] Y. Yang, "Coordination for Process Support is Not Enough", in *Proc. of the 4th European Workshop on Software Process Technology*, 1995, LNCS Vol. 913, Springer, pp. 205-208.

[32] K. Z. Zamli and P. A. Lee, "Taxonomy of Process Modeling Languages", in *Proc. of the ACS/IEEE Intl. Conf. on Computer Systems and Applications*, Lebanon, 2001. IEEE CS Press, pp. 435-437.

[33] K. Z. Zamli and P. A. Lee, "Exploiting a Virtual Environment in a Visual PML", in *Proc. of the 4th Intl. Conf. on Product Focused Software Process Improvements*, LNCS Vol. 2559, Rovaniemi, Finland, 2002, Springer, pp. 49-62.

[34] K. Z. Zamli and P. A. Lee, "Modeling and Enacting Software Processes Using VRPML", in *Proc. of the 10th IEEE Asia-Pacific Conf. on Software Engineering*, Chiang Mai, Thailand, 2003. IEEE CS Press, pp. 243-252.

[35] K. Z. Zamli. "Supporting Software Processes for Distributed Software Engineering Teams". *School of Computing Science, Univ. of Newcastle upon Tyne, PhD Thesis,* 2003.

BIOGRAPHY

Kamal Zuhairi Zamli obtained his BSc in Electrical Engineering from Worcester Polytechnic Institute, Worcester, USA in 1992, MSc in Real Time Software Engineering from CASE, Universiti Teknologi Malaysia in 2000, and PhD in Software Engineering from the University of Newcastle upon Tyne, UK in 2003. He is currently lecturing at the School of Electrical and Electronics Engineering, USM Engineering Campus in Transkrian. His research interests include software engineering, software process, software testing, visual languages, and object-oriented analysis and design.

Nor Ashidi Mat Isa obtained his BSc in Electrical Engineering from Universiti Sains Malaysia in 2000 and PhD in Image Processing and Neural Networks from the same university in 2003. He is currently lecturing at the School of Electrical and Electronics Engineering, USM Engineering Campus in Transkrian. He specialises in the area of image processing, neural networks for medical applications, and software engineering.