



# **Laporan Akhir Projek Penyelidikan Jangka Pendek**

## **The Design and Implementation of the VRPML Support Environment**

**by**  
**Dr. Kamal Zuhairi Zamli**  
**Dr. Nor Ashidi Mat Isa**

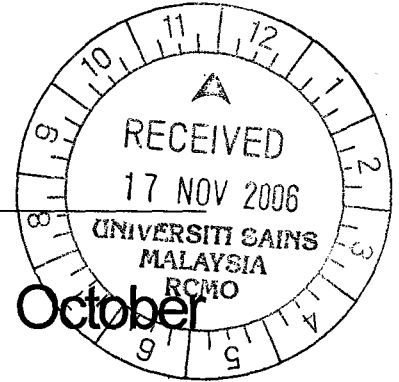
**2006**



UNIVERSITI SAINS MALAYSIA

**USM Short Term Grants –  
The Design and Implementation of the  
VRPML Support Environment**

# Final Report



Period: 1 November 2004 – 31 October  
2006

Document identifier :	304.PELECT.6035127
Date:	31 October, 2006
Version:	1.0
Document status:	Final Report
Author	Dr Kamal Zuhairi Zamli

Signature of Project Leader

(Dr Kamal Zuhairi Zamli)

PPKEE

Signature of Team Member

(Dr Nor Ashidi Mat Isa)

PPKEE

## Background

### **Abstract**

Software processes relate to the sequences of steps that must be performed by software engineers in order to pursue the goals of software engineering. In order to have an accurate representation and implementation of what the actual steps are, software processes may be modeled and enacted by a process modeling language (PML) and its process support system (called the Process Centered Environments i.e. PSEE). Although there has been much fruitful research into PMLs, their adoption by industry has not been widespread. Furthermore, no single PML and PSEE have assumed dominance and accepted as the de facto standard. For these reasons, research into PMLs and PSEEs are still necessary.

This project captures the design of the process support environment for a new process modeling language, called the Virtual Reality Process Modeling Language (VRPML). In doing so, this project identifies the main components of the VRPML process support environments as well as implements the working prototypes. Our experience highlights some lesson learned and offers insights into the design of next-generation PMLs and PSEEs.

**Keywords: Process Modelling Language, VRPML, Software Engineering**

For Bahasa Malaysia Abstract, see Appendix A.

### **Introduction**

A software process can be defined as sequences of steps that must be followed by software engineers to pursue the goal of software engineering. In order to allow a better control of a particular software process, a model of that process (called a process model) can be created using a process modelling language (PML) making the process explicit and open to examination.

Through enactment (or execution) of the process model, automation, guidance, and enforcement of the policy embedded in a particular process model can be usefully achieved. Because of the aforementioned benefits, a PML and its process support environment (termed *the Process Centered Environment (PSEE)*) could form an important feature of future software engineering environments. For these reasons, research into PMLs and PSEEs are still necessary.

This research aims to design and implement the heterogeneous process support environment for the Virtual Reality Modelling Language (VRPML), a visual PML developed elsewhere as part of the author's PhD work. The aim of this research is to investigate the suitable support mechanism as well as the suitable runtime environment to realize some of the main novel features of VRPML, that is, in terms of the integration with a virtual environment, the support for dynamic creation and allocation of resources as well as the support for enactment in a distributed environment.

The objectives of this project are:

1. To identify the main requirements for the VRPML process support environment.
2. To build a heterogeneous prototype runtime support system.
3. To utilize an object-oriented analysis and design techniques using the Unified Modelling Language (UML) for designing the VRPML support environment
4. To evaluate VRPML and its supporting environment under the real software engineering settings.

## **Project Members**

The project members for this project are:

1. Dr Kamal Zuhairi bin Zamli (Project Leader)
2. Dr Nor Ashidi Mat Isa
3. Siti Norbaya Azizan (RA for two months)
4. Iza Sazanita Isa (RA for two months)

## **Progress**

The duration of the project is 2 years beginning 1 November, 2004 till 31 October, 2006. Referring to Figure 1, this project has now been completed.

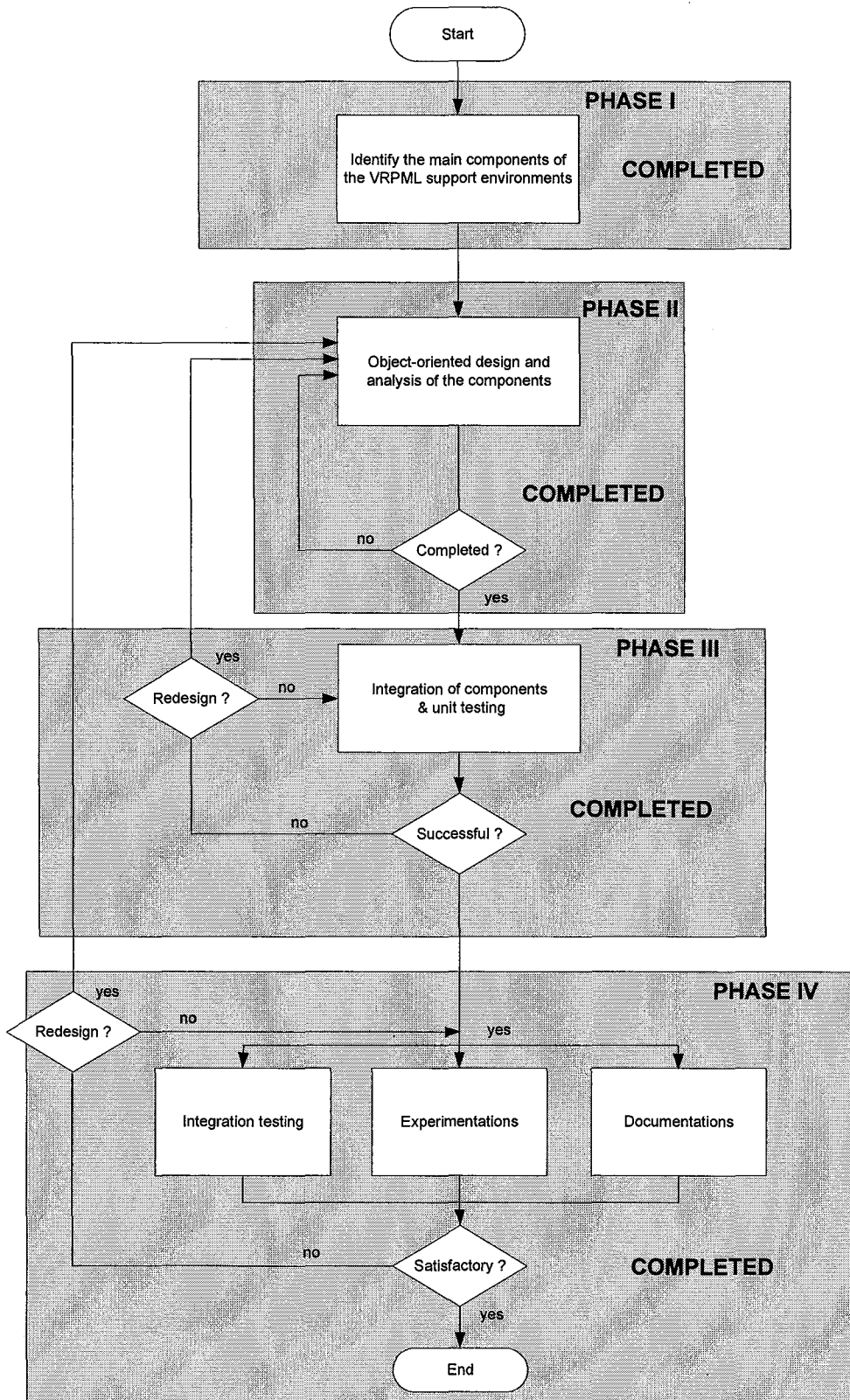


Figure 1: Project Flowchart

**PHASE I: Requirement elicitation**

In this phase, the requirements for the VRPML support system will be identified based on existing systems. Using these requirements, the main components of the support system will be decided.

**PHASE II: Design and implementation of each of the components**

Based on the decision in phase I, the design and implementation of each of the components will be considered utilising the object-oriented analysis and design tools based on the UML. Here, the use case diagrams, sequence diagrams, collaboration diagrams, class diagrams, activity diagrams, and component diagrams will be developed for each components. Further more, the overall design and implementation will undergo a number of iterations until all of the components are successfully completed.

**PHASE III: Unit testing and integration of the components**

In this phase, a number of test cases will be identified for each components. Unit testing will then be administered to ensure that each component behaves properly as expected. Once unit testing has been completed, the components will be integrated into the overall system.

**PHASE IV: Integration testing and experimentations**

In phase IV, integration testing and evaluations will be performed. Opportunities for optimization will also be considered here. Finally, the overall system will be testing as to whether or not the main novel features of VRPML can be supported particularly in terms of the support for integration with a virtual environment, the support for dynamic creation and allocation of resources as well as the support for enactment in a distributed environment

**PROJECT SCHEDULE**

		PROJECT PERIOD: 1 Nov 2004 – 31 October 2006																							
		2004				2005												2006							
Task		N	D	J	F	M	A	M	J	J	A	S	O	N	D	J	F	M	A	M	J	J	A	S	O
		O	E	A	E	A	P	A	U	U	U	E	C	O	E	A	E	A	P	A	U	U	U	E	C
1																									
2																									
3																									
4																									

Research activities:

- PHASE I** : Requirement Elicitation
- PHASE II** : Design and Implementations of components
- PHASE III** : UnitTesting and Integration of components
- PHASE IV** : Integration Testing, Experimentations and Documentation

## **Current Project Outputs - Publications**

These publications are enclosed in the appendix B.

### **Conference Publications**

1. Kamal Z. Zamli, and Nor Ashidi Mat Isa, "The Computational Model for a Flow-based Visual Language", in the proceedings of the AIDIS International Conference in Applied Computing 2005, Algarve, Portugal, pp.217-224 , Feb 22-25, 2005.
2. Kamal Z. Zamli, Nor Ashidi Mat Isa, Ahmad Nazri Ali, "Coordinating Business Processes Using a PML", in the proceedings of the International Conference on Information Integration and Web-based Applications and Services, (IIWAS 2005), pp 445-455. Sept 19-21, 2005, Kuala Lumpur, published by Austrian Computer Society
3. Kamal Z. Zamli, Nor Ashidi Mat Isa, Norazlina Khamis, "Implementing Executable Graph Based Visual Language in a Distributed Environment ", in the proceedings of the IEEE International Conference on Computing and Informatics 2006.

### **Local Journal Publications**

1. Kamal Z. Zamli and Nor Ashidi Mat Isa, "A Survey and Analysis of Process Modeling Languages", in the Malaysian Journal of Computing Science (ISSN No: 0127-9084) Vol. 17, No 2, December 2004, pp. 68-89.
2. Kamal Z. Zamli and Nor Ashidi Mat Isa, "Modeling and Enacting Software Processes: The How and Why Questions", Technical Journal PPKEE (ISSN No: 1594-6153), Vol. 10, December 2004, pp 19-27.
3. Kamal Z. Zamli, Nor Ashidi Mat Isa, and Norazlina Khamis, "The Design and Implementation of the VRPML Support Environment", in the Malaysian Journal of Computer Science (ISSN No: 0127-9084), Vol. 18, No 1, June 2005, pp. 57-69.
4. Kamal Z. Zamli and Nor Ashidi Mat Isa, "Enacting the waterfall software development model", accepted for publication in Jurnal Teknologi UTM (Siri D) , Vol. 43, Dec 2005, pp. 125-142.

### **International Journal Publications**

1. Kamal Z. Zamli, and Nor Ashidi Mat Isa, "The Applicability of VRPML for Supporting Distributed Software Engineering Teams", accepted for publication in the International Journal of the Computer, The Internet and Management (IJCIM) (will be in print for September/ / December 2006 issue)

### **Under Review**

1. Kamal Z. Zamli, Nor Ashidi Mat Isa, "Addressing Race Condition Problems in a Graph Based Visual Language", submitted for publication.

## **Current Project Outputs - Training**

### **Training**

Two RAs have been exposed to software process research, as a subset of software engineering. One of the RAs, Siti Norbaya Azizan, is now pursuing MSc by research in the area under the supervision of Dr Kamal Zuhairi Zamli.

Two MSc students have been using our prototype as a case study for testing the Software Fault Injection Tool (SFIT) (Refer to appendix D).



## Abstrak

Proses pembangunan perisian berkait rapat dengan turutan langkah yang mesti dilakukan oleh jurutera perisian untuk memenuhi matlamat kejuruteraan perisian. Untuk menghasilkan proses yang tepat dan lengkap, proses pembangunan perisian boleh dimodel dan dilari menggunakan bahasa permodelan (PML) dengan dibantu oleh sistem proses bantuan (PSEE). Walaupun terdapat banyak penyelidikan untuk menghasilkan bahasa permodelan PML dan system proses bantuan PSEE, adaptasi diperingkat industri masih kurang. Selain itu, tiada satu pun PML yang diterimapakai sebagai standard. Oleh yang demikian, penyelidikan dalam penghasilan PML dan PSEE masih lagi diperlukan.

Projek ini bertujuan untuk merekabentuk sistem proses bantuan (PSEE) untuk bahasa permodelan VRPML. Projek ini telah berjaya mengenalpasti komponen untuk VRPML dan mengimplementasi prototaip yang diperlukan. Pengalaman menjayakan projek ini dapat memberi gambaran dan sumbangan kepada rekabentuk PML dan PSEE pada masa akan datang.

## Abstract

Software processes relate to the sequences of steps that must be performed by software engineers in order to pursue the goals of software engineering. In order to have an accurate representation and implementation of what the actual steps are, software processes may be modeled and enacted by a process modeling language (PML) and its process support system (*called the Process Centered Environments i.e. PSEE*). Although there has been much fruitful research into PMLs, their adoption by industry has not been widespread. Furthermore, no single PML and PSEE have assumed dominance and accepted as the *de facto standard*. For these reasons, research into PMLs and PSEEs are still necessary.

This project captures the design of the process support environment for a new process modeling language, called the Virtual Reality Process Modeling Language (VRPML). In doing so, this project identifies the main components of the VRPML process support environments as well as implements the working prototypes. Our experience highlights some lesson learned and offers insights into the design of next-generation PMLs and PSEEs.

# LARIAN MODEL AIR TERJUN MENGGUNAKAN VRPML

Kamal Zuhairi Zamli and Nor Ashidi Mat-Isa

*Pusat Pengajian Kejuruteraan Elektrik dan Elektronik,  
Universiti Sains Malaysia, Kampus Kejuruteraan,  
14300 Nibong Tebal, Pulau Pinang, Malaysia  
Tel: 604-5937788 ext 6079, Fax: 604-5941023  
E-mail: { eekamal, ashidi }@eng.usm.my*

## ABSTRAK

Artikel ini menggariskan penggunaan bahasa visual yang baru, Bahasa Permodelan Proses Realiti Maya (VRPML), untuk spesifikasi proses pembangunan perisian. Secara khususnya, artikel ini membincangkan penggunaan VRPML dalam proses permodelan dan larian model air terjun. Matlamat utama kertas kerja ini adalah untuk mengkaji sama ada VRPML mempunyai notasi yang mencukupi untuk tujuan permodelan dan larian proses pembangunan perisian.

**Kata kunci:** Process Pembangunan Perisian, Kejuruteraan Perisian, Bahasa Permodelan Proses, VRPML

# ENACTING THE WATERFALL SOFTWARE DEVELOPMENT MODEL

Kamal Zuhairi Zamli and Nor Ashidi Mat-Isa

*School of Electrical and Electronic Engineering,  
Universiti Sains Malaysia, Engineering Campus,  
14300 Nibong Tebal, Pulau Pinang, Malaysia  
Tel: 604-5937788 ext 6079, Fax: 604-5941023  
E-mail: { eekamal, ashidi }@eng.usm.my*

## ABSTRACT

This paper describes the use of a new visual language, called the Virtual Reality Process Modeling Language (VRPML), in order to specify a software process. In particular, this paper demonstrates the use of VRPML to model and enact (i.e. execute) the waterfall software development model. The main aim of this paper is, therefore, to investigate whether VRPML provides a sufficiently rich notation to enable the modeling and enacting of software processes.

**Key words:** Software Process, Software Engineering, Process Modeling Languages, VRPML

## 1.0 INTRODUCTION

A software process can be defined as sequences of steps that must be followed by software engineers to pursue the goals of software engineering. In order to allow a better control of a particular software process, a model of that process (termed *a process model*) can be created using a process modeling language (PML) making the process explicit and open to examination. Furthermore, through enactment (or execution) of the process model, automation, guidance, and enforcement of the policy embedded in a particular process model can be usefully achieved.

While there has been much fruitful research into PMLs (see [19] for a recent survey), their adoption by industry has not been widespread [5]. While the reasons for this lack of success may be many and varied, our research identified two areas in which PMLs may have been deficient: human dimension issues; and support for addressing management and resource issues that might arise dynamically when a PML is being enacted [18]. Furthermore, no single existing PML has emerged as the *de facto* standard for supporting the modeling and enacting of software processes. These reasons suggest that research into PMLs is still necessary.

This paper describes our assessment of a new visual PML, called the Virtual Reality Process Modeling Language (VRPML) [14-20] developed as part of our on-going research. The main design objectives for VRPML were:

- To develop an expressive, executable, and easy to use visual PML
- To address some of the perceived deficiencies in existing PMLs particularly in terms of the support for dynamic creation and assignment of tasks and resources, as well as the support for the awareness and visualization issues.

Although VRPML has been successfully employed to model and enact the standard benchmark problem in software engineering (e. the ISPW-6 problem) involving the software change request [7], the author felt that such experience may be insufficient to evaluate VRPML completely. One reason is that the ISPW-6 problem is perhaps too specific to the software change request process.

A more general case study process, particularly involving the software processes for a complete software development model is required. These processes must be explicit and well-defined in terms of their inputs and outputs. Arguably, if one could use an existing definition of a development model in which activities and their inputs and outputs have already been well-defined, more effort can be concentrated on the modeling and enacting issues and less on defining the stages (and activities). Because the waterfall software development model seems to fit well into this category, it will be used here. The focus of this

paper is, therefore, to explore the expressiveness of the VRPML notation for supporting the modeling and enacting of a complete software development model.

This paper will be organized as follows. Section 2 gives an overview of VRPML. Section 3 describes the waterfall software development model. Section 4 presents the waterfall model expressed in VRPML. Section 5 presents some of the lesson learnt from the experiences. Finally, section 6 presents our conclusion.

## 2.0 OVERVIEW OF VRPML

VRPML is a control-flow based visual PML for supporting the modeling and enacting of software processes. In VRPML, software processes are generically modeled. Resources (in terms of software engineers, artifacts and tools) can be dynamically assigned and customized for specific projects from a generic model.

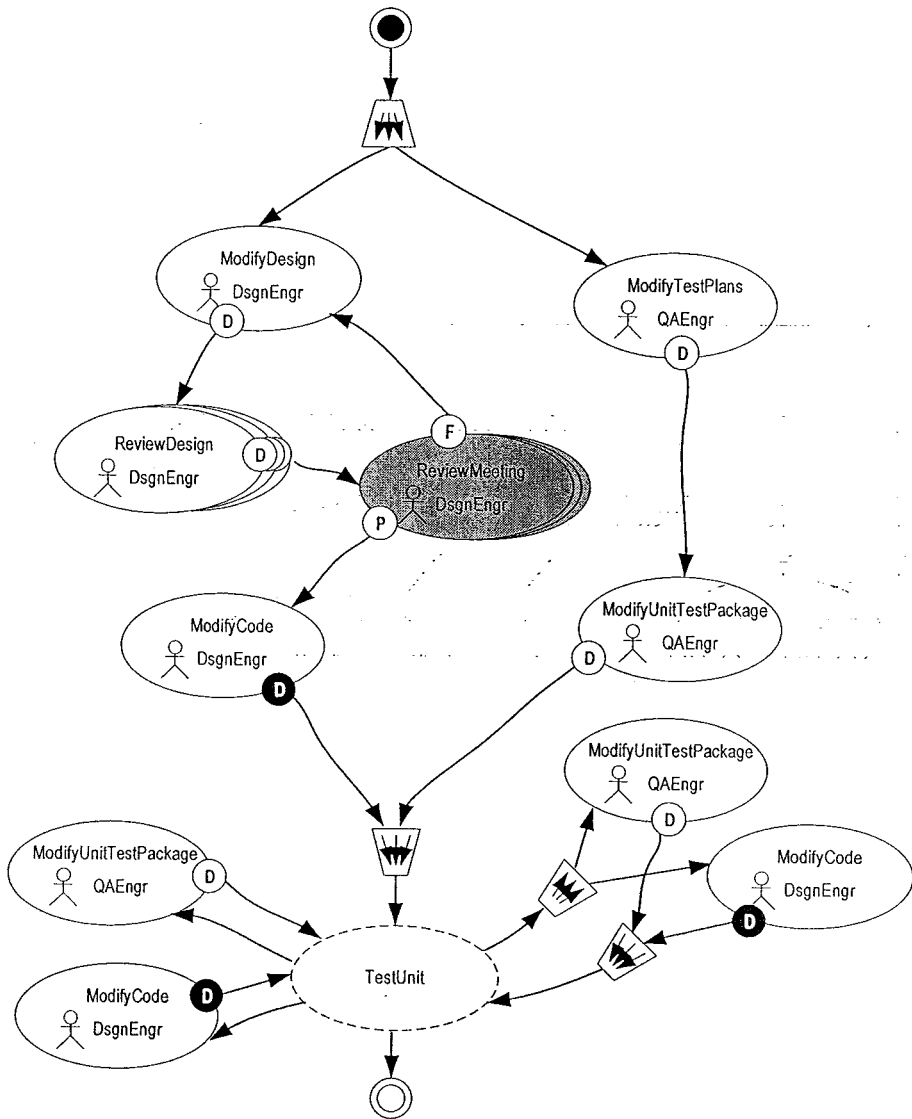


Fig. 1: Excerpt of the VRPML Graph

Software processes are specified in VRPML as graphs, by interconnecting nodes from top to bottom using arcs that carry run-time control-flow signals. The complete description of the syntax and semantics of VRPML can be found in [16].

As an illustration, Figure 1 presents an excerpt of the VRPML solution to the ISPW-6 problem. Similar to JIL [11] and Little JIL [13], software processes in VRPML are described using process step abstractions, which represent the most atomic representation of a software process (i.e. the actual activity that software engineers are expected to perform). These activities are represented as nodes, called activity nodes (shown as small ovals with stick figures).

As depicted in Figure 1, VRPML supports many different kinds of activity nodes. They include: *general-purpose activity nodes* (shown as individual small ovals with stick figures); *multi-instance activity nodes* (shown as overlapping small ovals with stick figures); and *meeting activity node* (shown as small and shaded overlapping ovals with stick figures). Both multi-instance activity nodes and meeting activity nodes have associated depths, indicating the actual number of engineers involved (and also the number of identical activities in the case of multi-instance activity).

The firing of activity nodes is controlled by the arrival of a control flow signal. In VRPML, an initial control flow signal is always generated from a *start node* (a white circle enclosing a small black circle). A *stop node* (a white circle enclosing a nother white circle) does not generate any control flow signals. Control flow signals may also be generated at the completion of a node, often from special completion events called *transitions* (shown as small white circles with a capital letter, attached to an activity node) or *decomposable transitions* (small black circles with a capital letter). Decomposable transitions enable automation scripts or sub-graphs to be specified (and executed if selected) as post-conditions before allowing transition to generate a control flow signal. The sub-graph associated with the decomposable transition representing Done (labeled D) for the activity node called Modify Code is given in Figure 2.

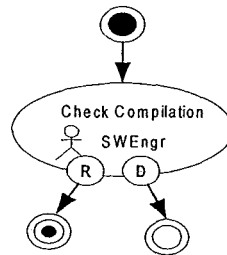


Fig. 2: Sub-graph for Decomposable Transition labeled D in Modify Code

When Check Compilation fails, the assigned software engineer can select the transition R (for re-do). As a result, a control-flow signal will be generated to re-enact its parent node (i.e. Modify Code) through a *re-enabled node* (shown as two white circles enclosing black circle). Otherwise, if the compilation is successful, the assigned engineer can select the transition D (for Done). In this case, the control-flow signal will be generated and propagated back to the main graph to enable the subsequent connected node.

In VRPML, activity nodes can also be enacted in parallel using combinations of language elements called *merger* and *replicator* nodes (shown as trapezoidal boxes with arrows inside). To improve readability, a set of VRPML nodes can be grouped together and replaced by a *macro node* (shown as dotted line ovals), with the macro expansion appearing on a separate graph. For example, referring to Figure 1, Test Unit is a macro node. The macro expansion of Test Unit is given in Figure 3.

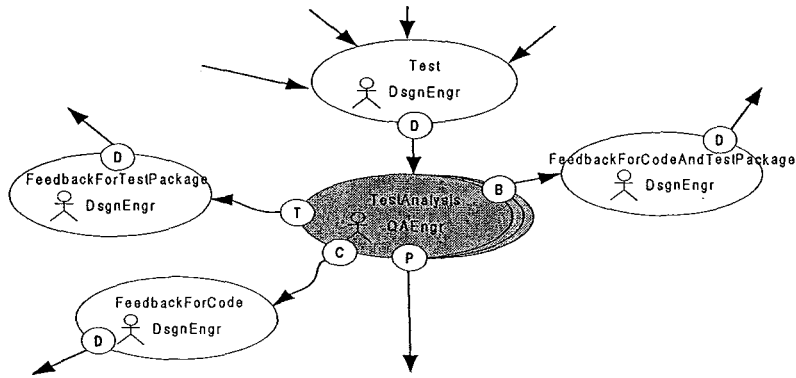


Fig. 3: Macro Expansion for Test Unit

For every activity node, VRPML provides a separate *workspace*, the concept borrowed from ADELE-TEMPO [2], APEL [3] and MERLIN [6]. Figure 4 depicts the sample workspace for the activity node called Review Meeting in Figure 1. A workspace typically gives a *work context* of an activity as it hosts resources needed for enacting the activity: transitions, artifacts (shown as overlapping two overlapping documents with arrows for depicting access rights), communication tools (shown as a microphone, and an envelope), and any task descriptions (shown as a question mark). Effectively, when an activity is undertaken, the workspace is mapped into a virtual room, transitions into buttons, and artifacts, communication tools (i.e. for synchronous and asynchronous forms of communications) and task description into objects which can be manipulated by software engineers to complete the particular task at hand.

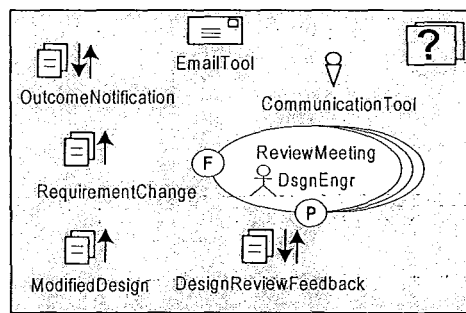
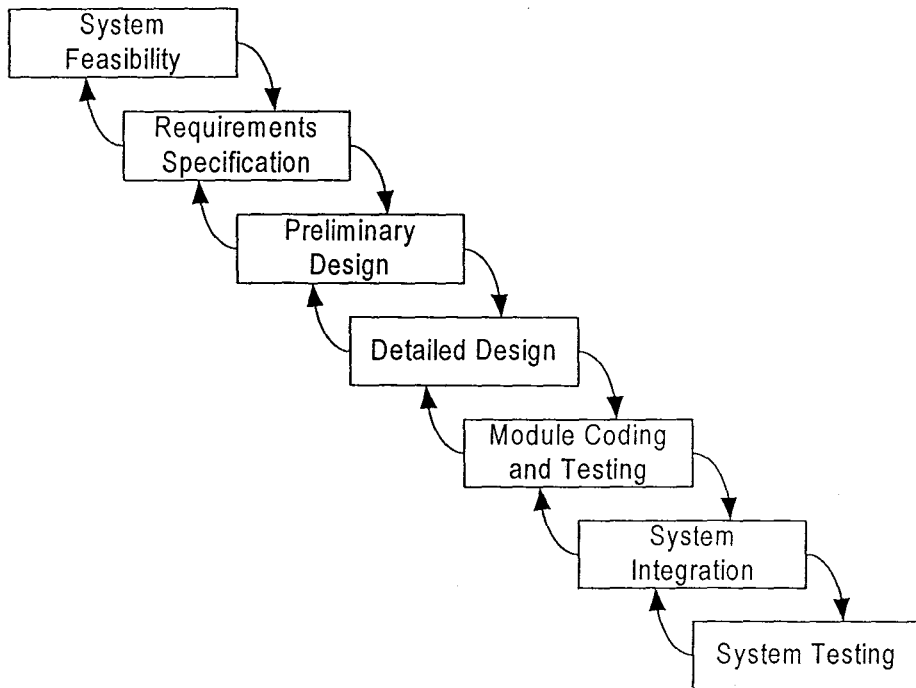


Fig. 4. Sample Workspace for Activity Node Review Meeting

As part of its enactment model, VRPML relies on its resource exception handling mechanism. In VRPML, resources include roles assignment, artifacts and tools (including communication tools) in a workspace as well as the depths of multi-instance activity nodes and meeting activity nodes. Depending on the needs of a particular software development project, these resources can either be allocated during graph instantiation or dynamically during graph enactment.

### 3.0 OVERVIEW OF THE WATERFALL DEVELOPMENT MODEL

The earliest form of the software processes based on the waterfall model was introduced by Royce (1970). Since then, many variants from the original waterfall model have been proposed. One of its variants [4] is shown in Figure 5.



**Fig. 5: The Waterfall Model**

Among the characteristics of the waterfall model are:

- The model is divided into a number of separate stages from system feasibility to maintenance.
- Each stage has a clearly delineated activity which is performed in a linear and sequential manner.
- Each stage is also independent that is, there is no overlap amongst stages.
- Feedback is usually provided to the preceding stage.
- The completion of a stage is determined by a review either formally or informally and conducted at the end of each stage so that development can proceed to the next stage. This is important because the output of the current stage often becomes the input of the next stage.

In order to support the modeling and enacting of software processes implementing the waterfall model, each stage of the model must be precisely defined in detail. Building on the work by Sommerville [12] and the waterfall model given earlier, Table 1 in the next page summarizes the possible activities along with their inputs and outputs. It must be stressed that this is only one of the possible list of activities as there are a number of variations to the waterfall model.



Waterfall Stages	Activities	Inputs	Outputs
System Feasibility	Analyse and Define Requirements	Customer Requirements	Draft Feasibility Study Draft Requirement Documents
	Review System Feasibility	Draft Feasibility Study Draft Requirement Documents	Feasibility Study Requirement Documents
Requirements Specification	Prepare Functional Specification	Requirement Documents	Draft Functional Specification
	Prepare Acceptance Test Plan	Requirement Documents	Draft Acceptance Test Plan
	Prepare Draft User Manual	Draft Functional Specification	Draft Preliminary User Manual
	Review Specification	Draft Functional Specification Draft Acceptance Test Plan Draft Preliminary User Manual	Functional Specification Acceptance Test Plan Preliminary User Manual
Preliminary Design	Prepare Architectural Specification	Requirement Documents Functional Specification	Draft Architectural Specification
	Prepare System Test Plan	Requirement Documents Functional Specification	Draft System Test Plan
	Review Preliminary Design	Draft Architectural Specification Draft System Test Plan	Architectural Specification System Test Plan
Detailed Design	Prepare Interface Specification	Functional Specification Architectural Specification Requirement Documents	Draft Interface Specification
	Prepare Integration Test Plan	Functional Specification Architectural Specification Requirement Documents	Draft Integration Test Plan
	Prepare Design Specification	Functional Specification Architectural Specification Draft Interface Specification Requirement Documents	Draft Design Specification
	Prepare Unit Test Plan	Functional Specification Architectural Specification Draft Interface Specification Requirement Documents	Draft Unit Test Plan
	Review Detailed Design	Draft Interface Specification Draft Design Specification Draft Integration Test Plan Draft Unit Test Plan	Interface Specification Design Specification Integration Test Plan Unit Test Plan
Module Coding and Testing	Perform Coding	Requirement Documents Design Specification	Draft Program Code
	Perform Unit and Module Testing	Unit Test Plan Draft Program Code	Draft Unit Test Report
	Review Coding and Testing	Draft Program Code Draft Unit Test Report	Program Code Unit Test Report
System Integration	Perform Integration Testing	Integration Test Plan Program Code	Draft Integration Test Report
	Prepare Final User Manual	Preliminary User Manual Functional Specification Program Code	Draft User Manual
	Review Integration Testing	Draft Integration Test Report Draft User Manual	Integration Test Report User Manual
System Testing	Perform System Testing	System Test Plan Acceptance Test Plan Program Code	Draft System Test Report
	Review System	Program Code User Manual Draft System Test Report	Final Release

**Table 1: Waterfall Model Activities, Inputs, and Outputs**

Referring to Table 1, the summary of activities involved in each stage of the waterfall model raises a number of issues. Firstly, the roles associated with each defined activity have not been identified. In this case, it is assumed that all of the software engineers involved have the required skills to perform the activities assigned to them. Hence, the role for each activity will be simply software engineers.

Secondly, although the ordering of activities in each stage has not been defined, they can be indirectly inferred from their input dependencies. In fact, activities in a stage can also be enacted in parallel when they are independent, that is, they do not require any input from each other. This will be reflected in the VRPML solution given below.

Finally, in order to highlight only the key aspects of VRPML, only a partial solution of the waterfall model from Table 1 will be presented here. The complete solution can be found in Zamli [17].

#### 4.0 VRPML SOLUTION OF THE WATERFALL DEVELOPMENT MODEL

The main graph of the VRPML solution for the software processes based on the waterfall model is given in Figure 6 consisting of 7 macros namely: System Feasibility; Requirements Specification; Preliminary Design; Detailed Design; Module Coding and Testing; System Integration; and System Testing.

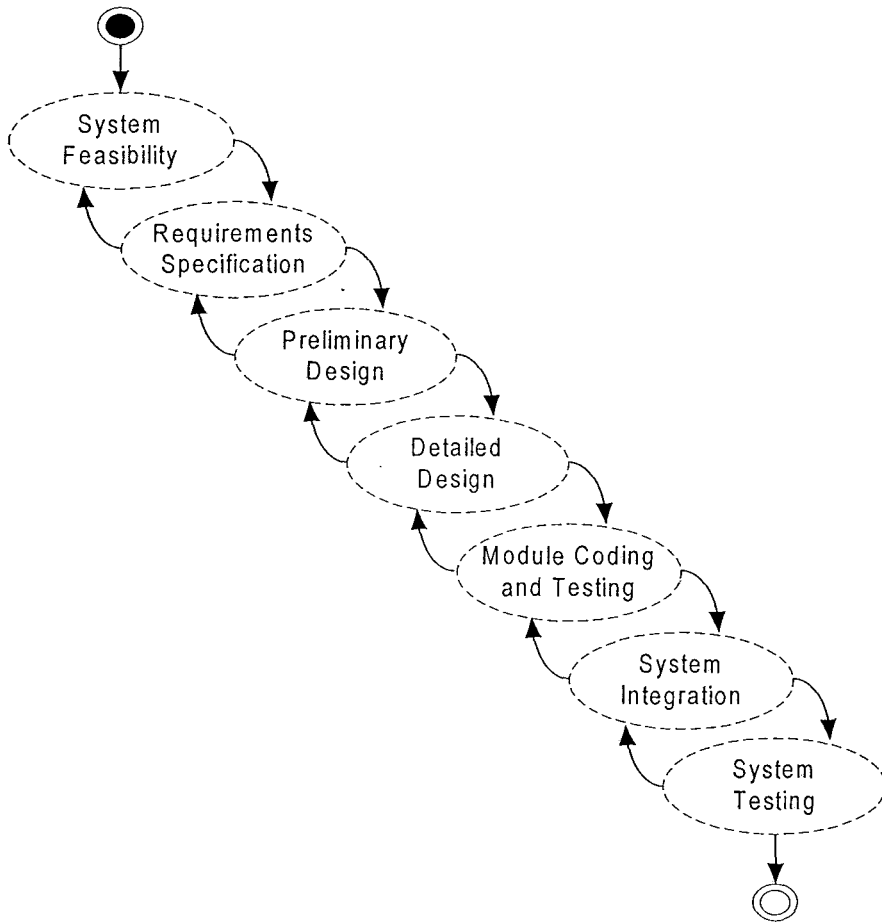
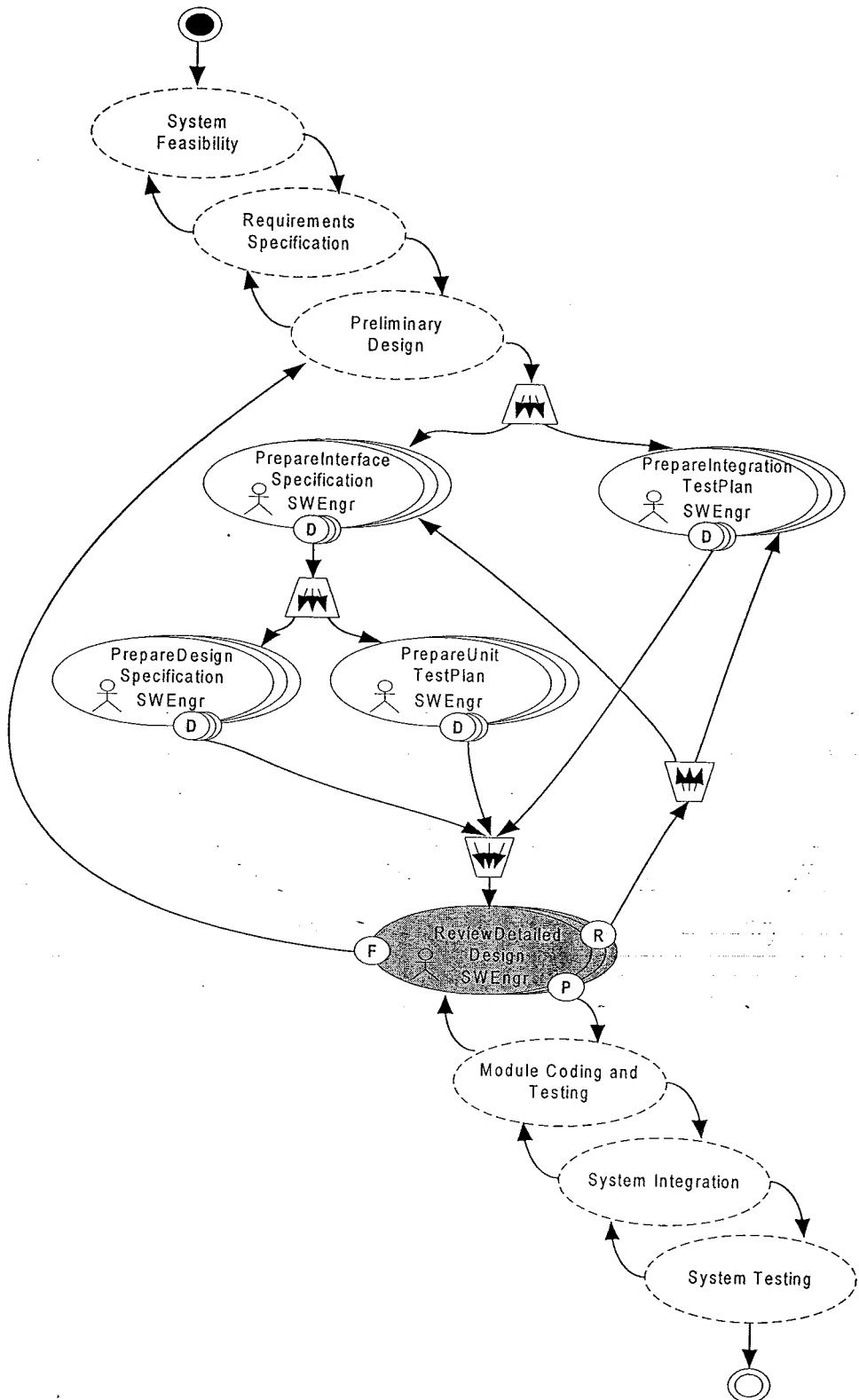


Fig. 6: Main VRPML Graph for the Waterfall Model

To further illustrate the VRPML notation, one of the macros, called Detailed Design, is shown in Figure 7. The presence of macros related to other stages in the Figure is merely to give focus and context to the expansion.



**Fig. 7: Macro Expansion for Detailed Design**

The macro expansion for Detailed Design consists of four multi-instance activity nodes (Prepare Interface Specification, Prepare Integration Test Plan, Prepare Design Specification, and Prepare Unit Test Plan) and

one meeting node (Review Detailed Design). Because Prepare Interface Specification and Prepare Integration Test Plan are independent of each other, they can be enacted in parallel. However, Prepare Design Specification and Prepare Unit Test Plan can only be enacted after Prepare Interface Specification has been completed. This is because both Prepare Design Specification and Prepare Unit Test Plan require an artifact from Prepare Interface Specification, called the Interface Specification, as one of their inputs (see Table 1). Actually, once Prepare Interface Specification has been completed, both Prepare Design Specification and Prepare Unit Test Plan can be enacted in parallel. Lastly, Review Detailed Design is enacted when all the above activities have been completed.

In terms of transitions, Prepare Interface Specification, Prepare Integration Test Plan, Prepare Design Specification and Prepare Unit Test Plan each have only one defined transition for Done (labeled D) to allow their completion. However, Review Detailed Design has three defined transitions: Redo (labeled R) in order to allow loop back to the previous activities; Passed (labeled P) in order to move to the next stage; and Feedback (labeled F) in order to permit feedback to the previous stage.

In terms of workspaces, they can be straightforwardly defined by analyzing the inputs for each activity as described in Table 1. As an illustration, Figure 8 depicts all the respective workspaces for Detailed Design.

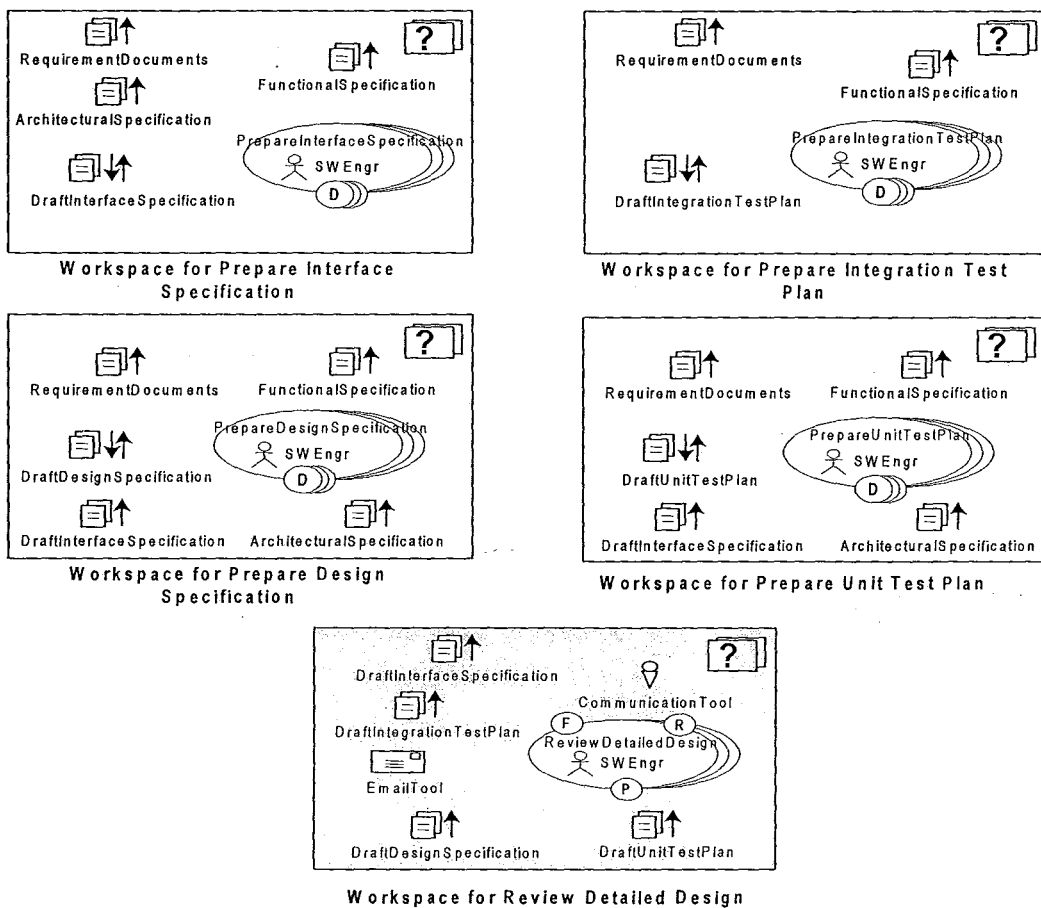
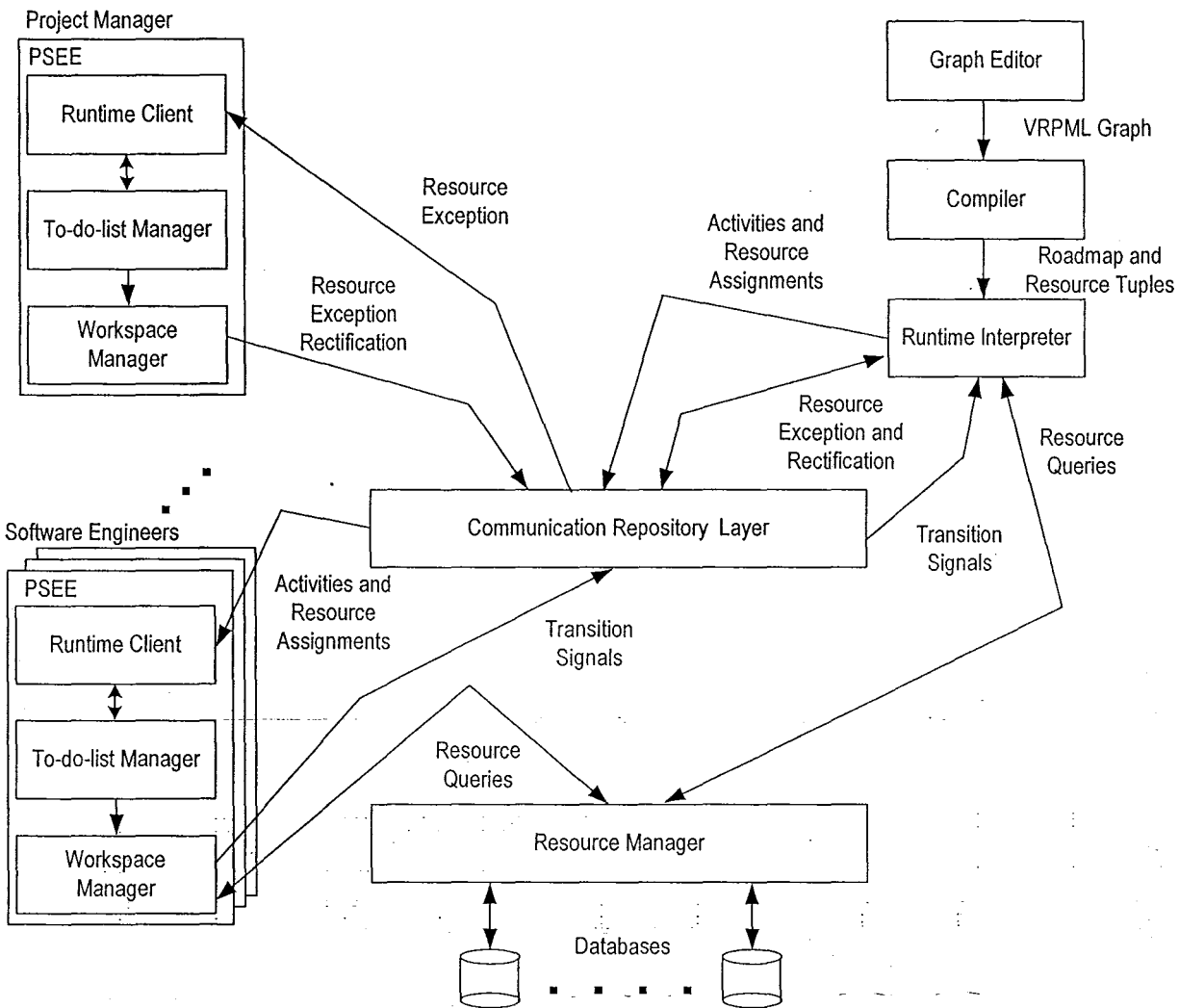


Fig. 8: Workspaces for Detailed Design

Although not shown in this paper (see [17]), all of the macros given in Figure 6 expand into a combination of multi-instance activity nodes and meeting activity nodes connected by arcs. The reason for using multi-instance activity nodes and meeting activity nodes is to demonstrate that VRPML supports the dynamic creation of tasks, that is, no prior assumption is made when constructing the model in terms of how many engineers have to be assigned to any of the activities represented by these nodes.

As far as enactment is concerned, the VRPML support system is responsible to allow the process model to be enacted. The overall structure of the VRPML support system is shown in Figure 9.



**Fig. 9: VRPML Support System**

The main components of the VRPML support system consist of:

- **Graph Editor** – allows the VRPML graphs to be specified.
- **Compiler** – compiles the VRPML graphs into an immediate format for enactment.
- **Runtime Interpreter** – interprets the compiled VRPML graph.
- **Communication Repository Layer** – allows communication between the runtime interpreter, runtime client, and workspace manager.
- **Resource Manager** – queries the databases for artifacts.
- **Process Centered Environment (PSEE)** – encapsulates three main sub-components: the runtime client, the to-do-list manager, and the workspace manager. The runtime client retrieves activities and resource assignments from the communication repository layer. The to-do-list manager manages the activities assigned to a particular software engineer whilst the workspace manager manages activity workspace in a virtual environment, manages activity transition, and forward queries to the resource manager.

The complete description of the VRPML support system, however, is beyond the scope of the paper. Interested readers are referred to Zamli [17].

To illustrate enactment, Figure 10 shows a sample snapshot of the to-do-list GUI for a software engineer name Kamal where the current activity in the to-do-list queue is Review Detailed Design whilst Figure 11 depicts the snapshot of resource allocation activity performed by the project manager.

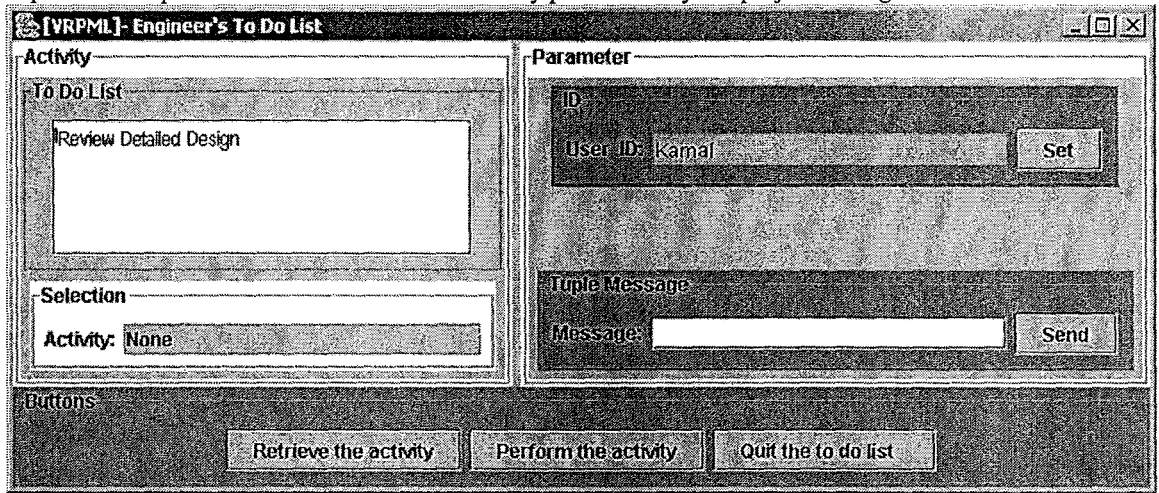


Fig. 10: Snapshot of the engineer's to-do-list

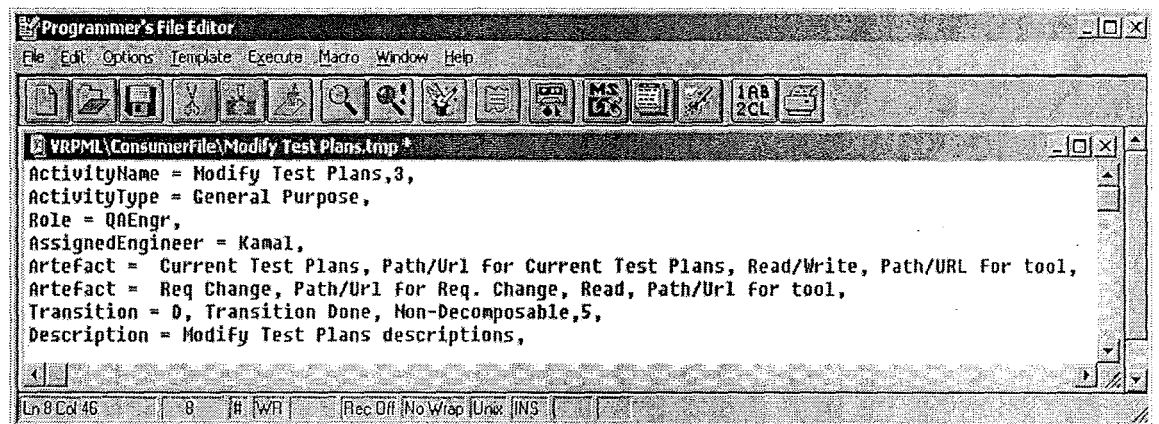
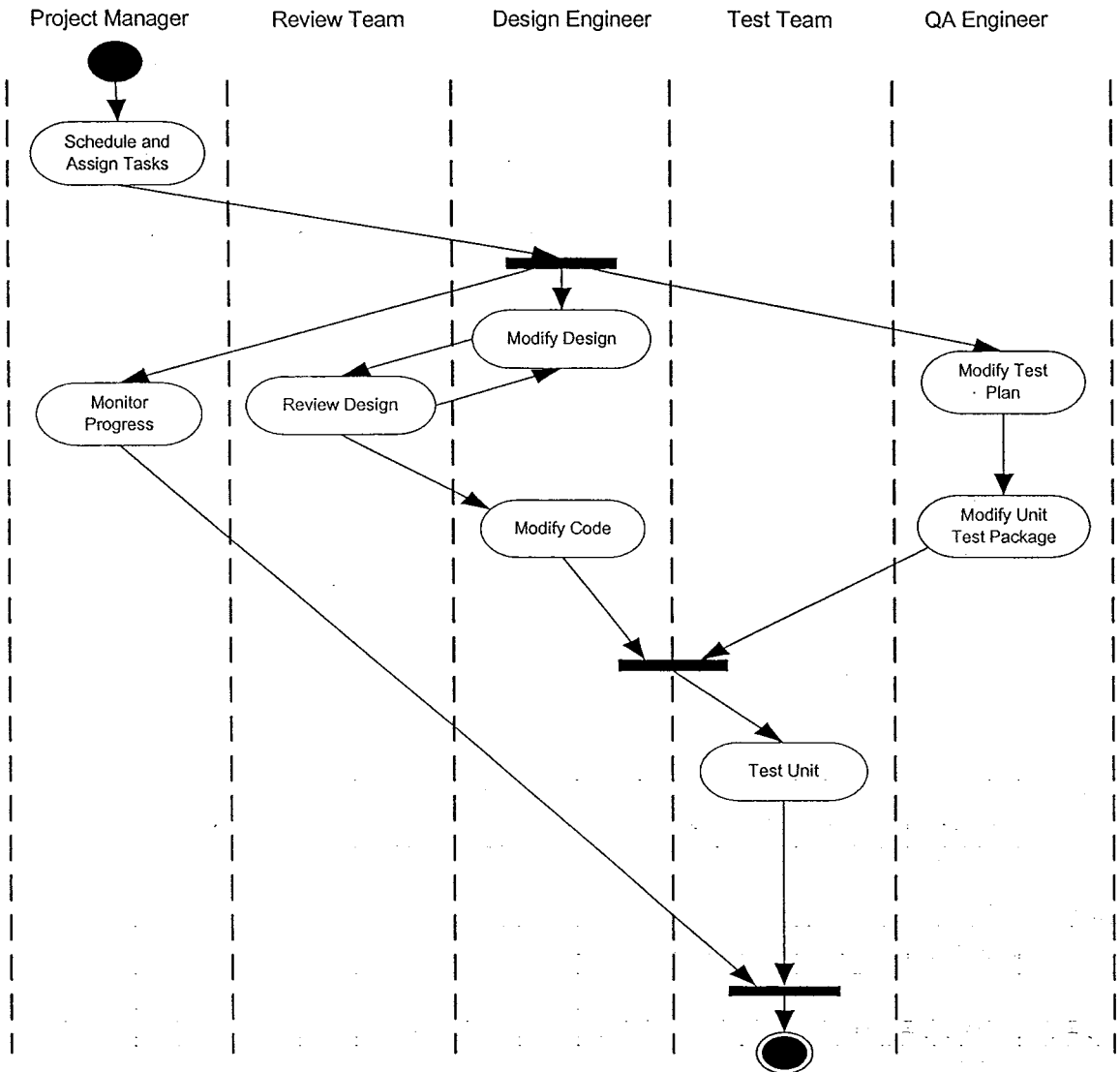


Fig. 10: Snapshot of resource allocation activity

## 5.0 DISCUSSION

The fact that VRPML provides a sound solution to the waterfall development model as well as its enactment gives an encouraging indication of the expressiveness of VRPML. This can be further supported from the fact that the VRPML solution itself can be arranged like the waterfall development model. The ability of VRPML to support such arrangement may be useful to facilitate process understanding. In fact, similar arrangement may not be possible in other visual PMLs such as Slang [1], Promenade [8], and APEL [3].

Although the UML activity diagram [10] is non-enactable, it can be compared to VRPML in terms of its graph representation. Figure 11 depicts an example of a software process expressed using the UML activity diagram.



**Fig. 11: Example of the UML Activity Diagram**

The UML activity diagram representation of a software process is simple and intuitive. Nonetheless, while the UML activity diagram can be used to express activities in a software process, it lacks features to express the individual role, resources, work contexts, and the completion of activities. Furthermore, UML activity diagrams do not have a well-defined executable semantics (i.e. as in VRPML). A known experience of using UML as a PML can be seen in the design of PROMENADE [8]. Here, the authors of PROMENADE dismiss the use of activity diagram as a PML, as PROMENADE mainly relies on class diagrams and object constraint language for supporting the modeling and enacting of software processes. Furthermore, in doing so, the authors of PROMENADE extensively extend the UML meta-models, hence, affecting the standardization of UML. For these reasons, we believe that UML is not particularly suitable as a PML.

Referring to the VRPML solution to the waterfall development model discussed in the previous section, multi-instance and meeting activity nodes were sufficient to construct that process model. Thus, at a glance, removing the general purpose activity node from the VRPML notation seems beneficial to reduce the language complexity. Nevertheless, eliminating the general purpose activity node from the notation can be disadvantageous. As far as readability of a VRPML graph is concerned, it can be difficult to distinguish whether an activity will be solely performed by one person or collaboratively by more than one person

[18]. Therefore, it is suggested that both general purpose activity nodes and multi-instance activity nodes are kept as part of the notation.

Concerning enactment, the fact that VRPML can produce an enactable model is helpful to facilitate coordination of activities involved in a particular development cycle. In addition, the support for enactment in VRPML can also be helpful for the following reasons:

- It provides guidance through the steps to be taken. Such guidance is particularly useful for junior software engineers.
- It can enforce strict procedures and policies. Enforcement of strict procedures is sometimes important in cases such as developing critical systems where human lives depend on a piece of software. An example of such a system would be a car auto-cruise control system. In this case, the software development team in charge of developing such a system may require its defined steps to be followed precisely. For example, evolution of the software in such a system must be strictly controlled. *Ad hoc* changes must not be permitted because such changes may introduce bugs which may not be tested and accounted for. Such bugs could be dangerous especially if they affect the mechanism to control the speed of the car in auto-cruise.
- It permits the automation of tasks. In software engineering, there are many tasks which can benefit from automation. For example, although tasks such as compiling and linking source codes look simple, they can be painstakingly dull especially if the source codes are very large and involving multiple modules. Such mundane tasks, if automated, can relieve software engineers from tedious routine work (and reduce potential human errors), and consequently, improve software engineer's productivity.

## 6.0 CONCLUSION

In conclusion, this paper has demonstrated the use of the VRPML for modeling and enacting of software processes. As for future work, we are planning to use VRPML to model a more realistic software process problem such as the spiral and the extreme programming model.

### Acknowledgement

The work undertaken in this research is partially funded by the USM Short Term Grants – “The Design and Implementation of the VRPML Runtime Environment”.

## REFERENCES

- [1] Bandinelli, S., Fuggetta, A., Ghezzi, C., and Lavazza, L. 1994. SPADE: An Environment for Software Process Analysis, Design and Enactment. In Finkelstein, A., Kramer, J. and Nuseibeh, B. (Eds.), *Software Process Modelling and Technology*, Research Studies Press, Taunton, England: 223-247.
- [2] Belkhatir, N., Estublier, J., and Melo, W. 1994. ADELE-TEMPO: An Environment to Support Process Modelling and Enaction. In Finkelstein, A., Kramer, J. and Nuseibeh, B. (Eds.), *Software Process Modelling and Technology*, Research Studies Press, Taunton, England: 187-122.
- [3] Dami, S., Estublier, J., and Amieur, M. 1998. APEL: A Graphical Yet Executable Formalism for Process Modeling”. *Automated Software Engineering*, 5(1):61-96.
- [4] DeBellis, M., and Haapala, C. 1995. User-Centric Software Engineering. *IEEE Expert*, February 1995: 34-41.
- [5] Jaccheri, M.J., Conradi, R., and Drynes, B.H.. Software Process Technology and Software Organisations. *Proceedings of the 7th European Workshop on Software Process (EWSPT 2000)*, Kaprun, Austria, 2000, Springer: 96-108.



- [6] Junkermann, G., Peuschel, B., Schafer, W., and Wolf, S. 1994. MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment. In Finkelstein, A., Kramer, J. and Nuseibeh, B. (Eds.), *Software Process Modelling and Technology*, Research Studies Press, Taunton, England: 103-129.
- [7] Kellner, M.I., Feiler, P.H., Finkelstein, A., Katayama, T., Osterweil, L.J., Penedo, M.H., and Rombach, H.D. 1990. Software Process Modeling Example Problem. *Proceedings of the 6th International Software Process Workshop*, Hakodate, Japan, IEEE CS Press.
- [8] Ribo, J.M., and Franch, X. 2000. PROMENADE: A PML Intended to Enhance Standardization, Expressiveness and Modularity in Software Process Modelling. Research Report LSI-34-R., Lluenguatges I Sistemes Informatics, Politechnical of Catalonia, Spain.
- [9] Royce, W.W. 1970. Managing the Development of Large Software Systems *Proceedings of IEEE WESCON*: 1-9.
- [10] Rumbaugh, J., Jacobson, I. and Booch, G. 1999. The UML Reference Manual. Addison Wesley.
- [11] Sutton, S. Jr., and Osterweil, L.J. 1997. The Design of a Next-Generation Process Language. *Proceedings of the Joint 6th European Software Engineering Conference and the 5th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, Lecture Notes in Computer Science Volume 1301, Springer: 142-158.
- [12] Sommerville, I. 2001 *Software Engineering (Sixth Edition)*. Addison Wesley.
- [13] Wise., A. 1998. Little JIL 1.0 Language Report - Technical Report 98-24, Department of Computer Science, University of Massachusetts at Amherst, USA.
- [14] Zamli, K.Z. and Lee, P.A. 2001. Taxonomy of Process Modeling Languages, *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, Beirut, Lebanon IEEE CS Press: 435-437.
- [15] Zamli, K.Z. 2001. Process Modeling Languages: A Literature Review. *Malaysia Journal of Computer Science* 14, 2: 26-37.
- [16] Zamli, K.Z. and Lee, P.A. 2002. Exploiting a Virtual Environment in a Visual PML. *Proceedings of the 4th International Conference on Product Focused Software Process Improvements (PROFES02)*, In Oivo, M. and Komi-Sirvio, S. (Eds.). Lecture Notes in Computer Science Volume 2559, Rovaniemi, Finland, Springer: 49-62.
- [17] Zamli, K.Z. 2003. Supporting Software Processes for Distributed Software Engineering Teams. PhD Thesis, School of Computing Science, University of Newcastle upon Tyne, United Kingdom
- [18] Zamli, K.Z. and Lee, P.A. 2003. Modeling and Enacting Software Processes Using VRPML. *Proceedings of the 10th IEEE Asia-Pacific Conference on Software Engineering*, Chiang Mai, Thailand, IEEE CS Press: 243-252.
- [19] Zamli, K.Z., and N.A. Mat Isa 2004. A Survey and Analysis of Process Modeling Languages. *Malaysia Journal of Computer Science* 17, 2: 68-89.
- [20] Zamli, K.Z., and N.A. Mat Isa. 2005. The Computational Model for a flow-based PML. *Proceedings of the AIDIS International Conference on Applied Computing 2005*, Algarve, Portugal, pp. 217-224.

# COORDINATING BUSINESS PROCESSES USING A PML

Kamal Zuhairi Zamli, Nor Ashidi Mat Isa, Ahmad Nazri Ali  
School of Electrical and Electronics  
USM Engineering Campus  
14300 Nibong Tebal, Penang, Malaysia

## **Abstract**

*Software processes relate to the sequences of steps that must be performed by software engineers in order to pursue the goal of software engineering. In order to have an accurate representation and implementation of what the actual steps are, software processes may be modeled and enacted by a process modeling language (PML).*

*In this paper, we investigate the use of a PML in a different scope, that is, to model a general form of processes involving workflow activities (termed business process). In doing so, we have adopted a new visual PML, called VRPML, developed as part of our on-going research. Our work is based on the hypothesis which suggests that business processes and software processes are similar in nature, thus, a PML may also be applicable to support business process activities.*

## **1. Introduction**

Software development is a process of change, refinement, transformation or addition to existing software product. In order to ensure a quality software product, it is often necessary to observe and control the processes that are used to produce that software product. Based on the aforementioned premise, much research has been conducted in the field of software engineering to develop ways of controlling software processes. One active area of research in the area is on the development of process modeling languages (PML). With a PML, a model of the process can be developed (termed *process model*). Through enactment (i.e. execution) of the process model, coordination, automation, guidance, and enforcement of policies embedded in the model can be achieved to support the activities of software engineers developing a software product.

Although a natural language can be used for defining a software process, it exhibits a number of difficulties. In general, the description of software process using a natural language is often imprecise, ambiguous, inconsistent and open to user interpretation. Typically, such characteristics may lead to discrepancies in the software process undertaken by software engineers – for example, what is performed may not be what is required in the description of the software process (i.e. as illustrated in Figure 1). With a PML such discrepancies may be alleviated.

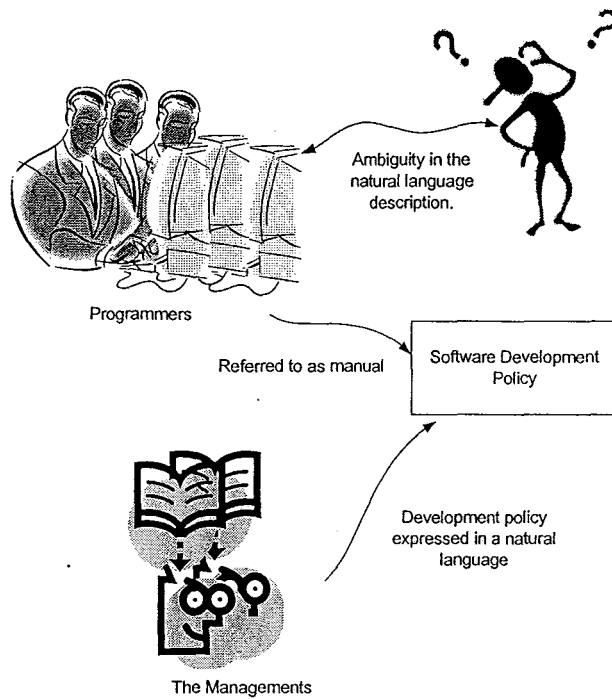


Figure 1. Problem with a natural language description

In this paper, we investigate the use of a PML to model a general form of processes involving workflow activities (termed *business process*). In doing so, we have adopted a new visual PML, called VRPML, developed as part of our on-going research [8-13]. Our work is based on the hypothesis which suggests that business processes and software processes are similar in nature, thus, a PML may also be applicable to support business process activities.

This paper is organized as follows. Section 2 gives an overview of VRPML. Section 3 discusses the case study problem used in this paper. Section 4 discusses some of the lessons learned. Finally, section 5 presents the conclusions of the paper.

## 2. Overview of VRPML

VRPML is a control-flow based visual PML for supporting the modeling and enacting of software processes. In VRPML, software processes are generically modeled. Resources (in terms of software engineers, artifacts and tools) can be dynamically assigned and customized for specific projects from a generic model.

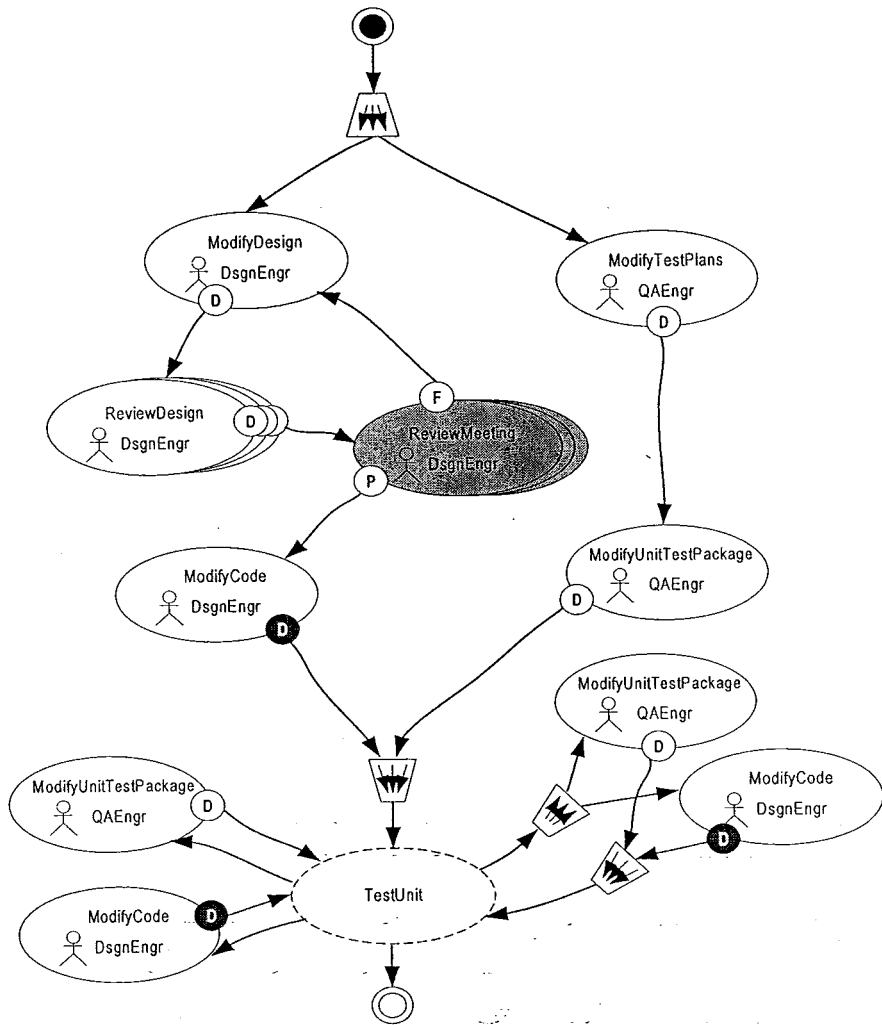


Figure 2. Excerpt from the VRPML Graph for the ISPW-6 Problem

Software processes are specified in VRPML as graphs, by interconnecting nodes from top to bottom using arcs that carry run-time control-flow signals. The complete description of the syntax and semantics of VRPML can be found in [10].

As an illustration, Figure 2 presents an excerpt of the VRPML graphs expressing the ISPW-6 problem [4]. Similar to Little JIL [5-7], software processes in VRPML are described using process step abstractions, which represent the most atomic representation of a software process (i.e. the actual activity that software engineers are expected to perform). These activities are represented as nodes, called activity nodes (shown as small ovals with stick figures).

As depicted in Figure 2, VRPML supports many different kinds of activity nodes. They include: *general-purpose activity nodes* (shown as individual small ovals with stick figures); *multi-instance activity nodes* (shown as overlapping small ovals with stick figures); and *meeting activity node* (shown as small and shaded overlapping ovals with stick figures). Both multi-instance activity nodes and meeting activity nodes have associated depths, indicating the actual number of engineers involved (and also the number of identical activities in the case of multi-instance activity).

The firing of activity nodes is controlled by the arrival of a control flow signal. In VRPML, an initial control flow signal is always generated from a *start node* (a white circle enclosing a small black circle). A *stop node* (a white circle enclosing another white circle) does not generate any control flow signals. Control flow signals may also be generated at the completion of a node, often from special completion events called *transitions* (shown as small white circles with a capital letter, attached to an activity node) or *decomposable transitions* (small black circles with a capital letter). Decomposable transitions enable automation scripts or sub-graphs to be specified (and executed if selected) as post-conditions before allowing transition to generate a control flow signal. The sub-graph associated with the decomposable transition representing Done (labeled D) for the activity node called Modify Code is given in Figure 3.

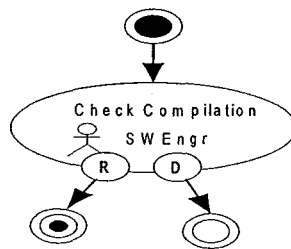


Figure 3. Sub-graph for Decomposable Transition labeled D in Modify Code

When Check Compilation fails, the assigned software engineer can select the transition R (for re-do). As a result, a control-flow signal will be generated to re-enact its parent node (i.e. Modify Code) through a *re-enabled node* (shown as two white circles enclosing black circle). Otherwise, if the compilation is successful, the assigned engineer can select the transition D (for Done). In this case, the control-flow signal will be generated and propagated back to the main graph to enable the subsequent connected node.

In VRPML, activity nodes can also be enacted in parallel using combinations of language elements called *merger* and *replicator* nodes (shown as trapezoidal boxes with arrows inside). To improve readability, a set of VRPML nodes can be grouped together and replaced by a *macro node* (shown as dotted line ovals), with the macro expansion appearing on a separate graph. For example, referring to Figure 2, Test Unit is a macro node. The macro expansion of Test Unit is given in Figure 4.

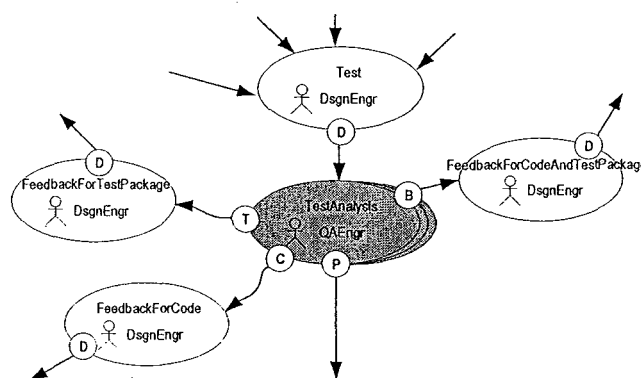


Figure 4. Macro Expansion for Test Unit in Figure 1

For every activity node, VRPML provides a separate *workspace*. Figure 5 depicts the sample workspace for the activity node called Review Meeting in Figure 2. A workspace typically gives a *work context* of an activity as it hosts resources needed for enacting the activity: transitions, artifacts (shown as overlapping two overlapping documents with arrows for depicting access rights), communication tools (shown as a microphone, and an envelope), and any task descriptions (shown as a question mark). Effectively, when an activity is undertaken, the workspace is mapped into a virtual room, transitions into buttons, and artifacts, communication tools and task description into objects which can be manipulated by software engineers to complete the particular task at hand. This mapping is based on Doppke's task-centered mapping described in [3].

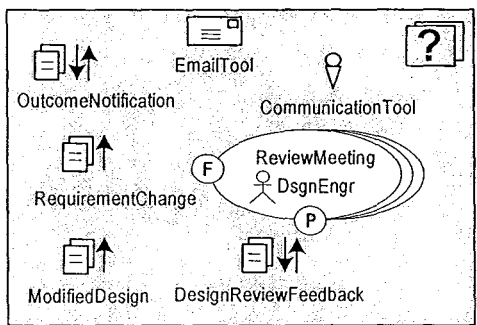


Figure 5. Sample Workspace for Activity Node Review Meeting from Figure 1

As part of its enactment model, VRPML relies on its resource exception handling mechanism. In VRPML, resources include roles assignment, artifacts and tools (including communication tools) in a workspace as well as the depths of multi-instance activity nodes and meeting activity nodes. Depending on the needs of a particular software development project, these resources can either be allocated during graph instantiation or dynamically during graph enactment. The VRPML's enactment model is summarized in Figure 6.

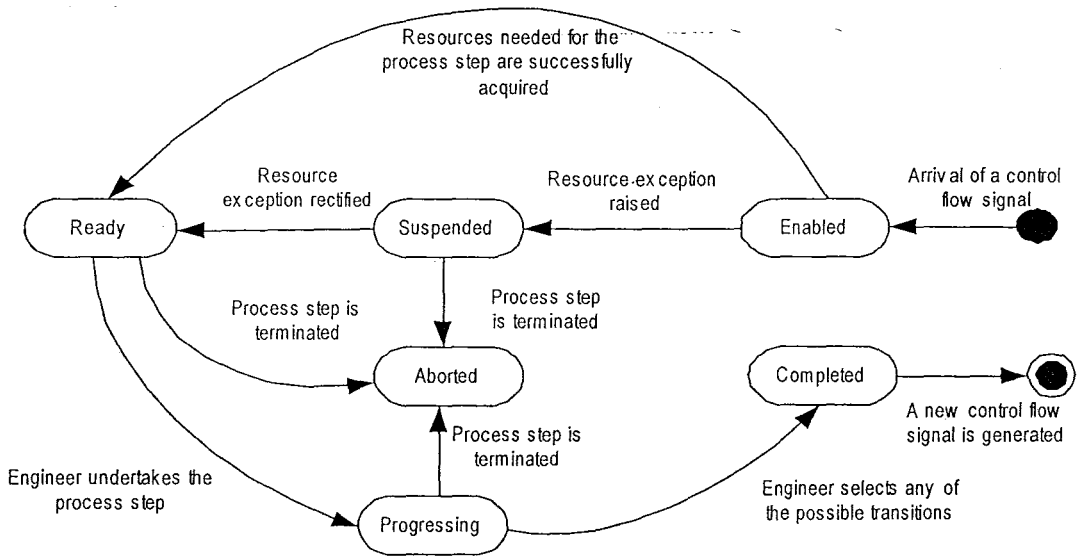


Figure 6. VRPML Enactment Model

Upon the arrival of the control-flow signal, an activity node will be enabled. Here, the VRPML interpreter attempts to acquire resources that the activity node needs. If resources are successfully acquired, the VRPML interpreter then instantiates the activity corresponding to that activity node. If for any reason VRPML fails to acquire the resources, enactment will be blocked until such resources are made available (e.g. an engineer has not been assigned to the activity). In this way, the VRPML's resource exception handling mechanism is similar to blocking primitives (e.g. in, read) in Linda [5]. Once enactment is blocked, the VRPML interpreter automatically produces an activity for the administrator (e.g. process engineer) to rectify the resource exception or completely terminate the current activity. If that activity is terminated, the administrator may optionally terminate the overall enactment of the particular VRPML graph in question or manually re-enact connecting nodes by providing the necessary control-flow signals that they need to fire. If the resource exception is rectified, normal enactment of the particular VRPML graph can be resumed resulting in the activity being assigned to the appropriate software engineer. When that engineer selects that particular activity, a workspace for that activity will appear as a virtual room with artifacts, transitions and communication tools as objects which software engineer can manipulate to complete the task. Finally, the activity completes when the software engineer selects one of the possible transitions (e.g. passed, failed, done, or aborted).

### **3. Case Study Problem**

In order to investigate the expressiveness of the VRPML notation to model and enact general workflow activities, we partially present in Figure 7, 8, 9, and 10 a trip planning problem, modeled in VRPML. The planning a trip problem is based on the one presented in [2] and its extension discussed in [6-7].

Briefly, the trip planning problem involves four people: the traveler, a travel agent, and two secretaries. The problem involves making an airline reservation, trying United first, and USAir. If (after making the plane reservation) the traveler has gone over budget and a Saturday stay over was not included, these dates should be changed. Another attempt should be made to include a Saturday stay over. After the airline reservation is made and travel dates and times are set, car and hotel reservation should also be made. The hotel reservation may be made at either a Days Inn, or if the budget is not tight, a Hyatt, whilst the car reservation be made with either Avis or Hertz.

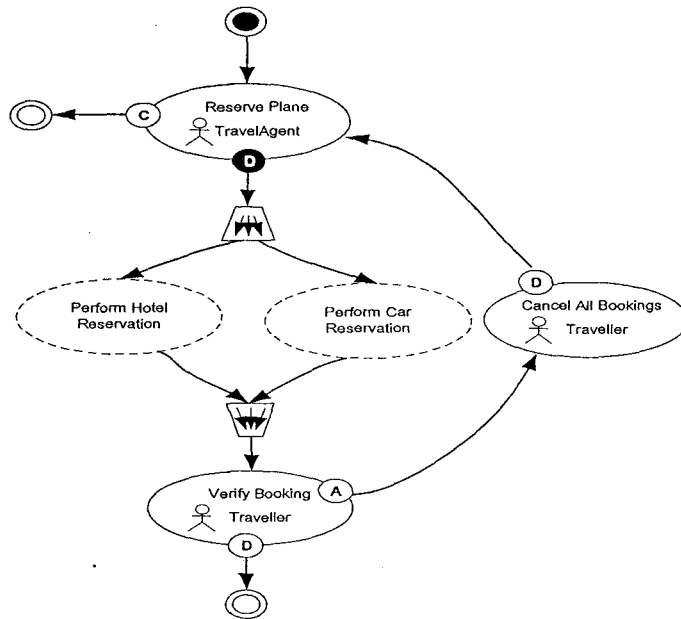


Figure 7. Main VRPML Graph for Planning a Trip Problem

The VRPML graph expressing the planning a trip problem can be seen in Figure 7. Here, the plane booking activities are implemented using decomposable transition labeled D (for Done) of the activity Reserve a Plane. The sub-graph associated with that transition is given in Figure 8.

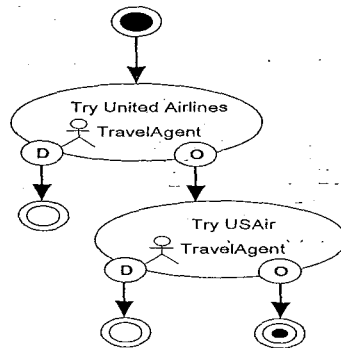


Figure 8. Sub-graph for Decomposable Transition labeled D in Reserve Plane

Here, the activity Try United Airlines is started when its parent activity (i.e. Reserve Plane) completes. Here, the travel agent may decide to choose transition D if the booking is within budget for the selected date and time. Otherwise, the traveler may select the transition O (for Over budget) in order to start the Try USAir activity. If the Try USAir is started, the travel agent may again select the transition D if the booking is within budget. Otherwise, the travel agent must select the transition O in order to re-enable its parent activity (see the semantics of re-enable node in Section 2). In this case, the parent activity (Reserve Plane) permits the travel agent to re-select the date and time for the plane reservation.

Concerning the activities involving hotel reservation and car reservation, they are performed in parallel (see Figure 7). Because both activities consist of a number of sub-activities, they are



abstracted using macro nodes Perform Hotel Reservation and Perform Car Reservation respectively. The macro expansion for Perform Hotel Reservation activity is given in Figure 9.

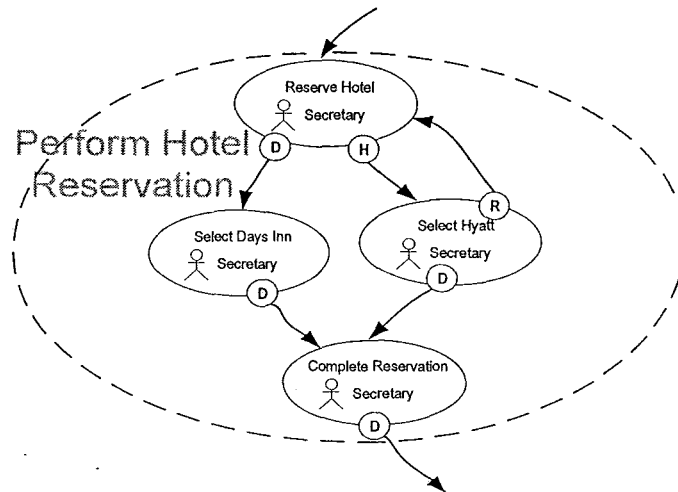


Figure 9. Macro Expansion for Perform Hotel Reservation

In this case, when the Reserve Hotel activity is started, the secretary may decide to select either Days Inn or Hyatt. The assumption in the planning a trip problem is that Hyatts always depletes more of the traveller’s budget than Days Inn [7]. Thus, if the secretary chooses to select Hyatt and the booking is over budget, the secretary must select the transition R in order to re-enable the Reserve Hotel activity. In the Reserve Hotel activity, the secretary may now choose Days Inn instead. Once reservation is completed, Complete Reservation activity will be started. Complete Reservation activity completes when the secretary selects the transition D (for Done).

Next, the macro expansion for Perform Car Reservation activity is given in Figure 10.

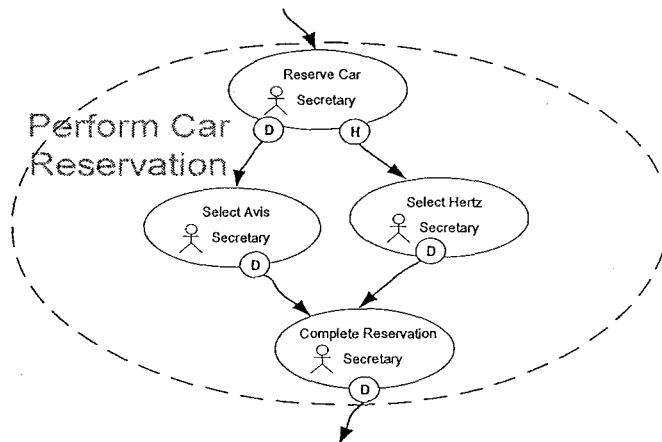


Figure 10. Macro Expansion for Perform Car Reservation

Unlike Reserve Hotel activity, Reserve Car activity does not put constraints on the car reservation. It is up to the secretary to choose either Avis or Hertz. Similar to the Reserve Hotel activity, Complete Reservation activity will be started when the secretary has completed the car reservation. The Complete Reservation activity finishes when the secretary selects the transition D (for Done).

Going back to Figure 7, when the car and hotel reservation have been successfully made, Verify Booking activity will be enabled. While the booking may not be over budget, the traveler may still choose to cancel all booking if the Saturday stay over is not included in the itinerary. To cancel, he may choose the transition A (for Abort) of the Verify Booking activity to initiate the Cancel All Bookings activity. Once completed (i.e. after the traveler chooses transition D of the Cancel All Bookings activity), the Reserve Plane activity will be re-enabled. In this case, hotel and car bookings activities may be re-started for different dates and time to ensure a Saturday stay over is included in the itinerary or the overall hotel and car booking activities may be completely terminated.

As far as data flows are concerned, they can be straightforwardly express in each of the workspaces of each activity. Because the focus of this paper is on coordination of activities, data flow issues will not be discussed further here.

#### 4. Discussion

Our earlier work indicates that VRPML can be successfully adopted to model and enact a number of case study problems involving software processes including the ISPW-6 [4] and the variation of the waterfall development model [1]. Here, the fact that VRPML can model and enact a general business process problem (i.e. planning a trip) gives further indication of the expressiveness of the VRPML notation. Nevertheless, since planning a trip problem is a rather small problem, it might be difficult to generalize the applicability of VRPML to any general business process. In fact, we felt that more experiences will be needed before the true value of VRPML can be established.

For comparison purposes, Figure 11 reproduces the partial Little JIL solution [7] to planning a trip problem. While Little JIL provides a rich set of notations for modeling of processes, it might be difficult at a glance to make sense out of Little JIL representation. One reason is that process activities are structured as a tree structure making it difficult to follow the sequencing of activities. In VRPML, process models are expressed as graphs resembling flowcharts. As such, process models expressed in VRPML should be straightforward to comprehend.

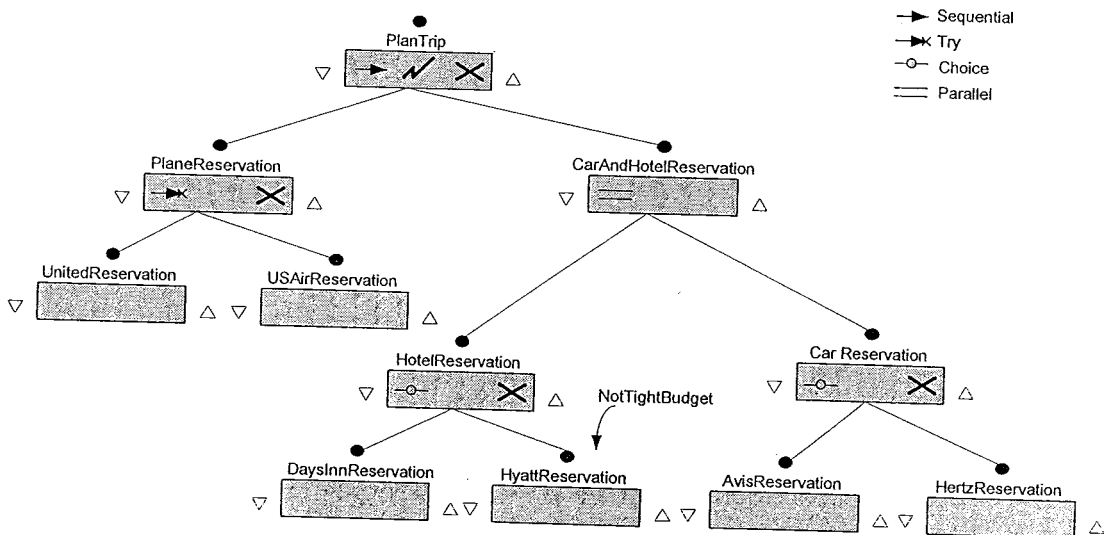


Figure 11. Excerpt from the Little JIL solution

Concerning modularity, VRPML permits the use of a macro node in order to improve the graph complexity. JIL, on the other hand, does not provide such a feature. For this reason, readability of the process model expressed in Little JIL might be significantly affected if a process involved large number of activities.

As far as our experience with PML is concerned, we believe that software processes and workflow activities (i.e. business processes) share many characteristics. In fact, there is an active and parallel research area looking into workflow languages as opposed to PMLs. In order to understand the differences between a PML and a workflow language, consider the analogy of a fighter plane and a commercial plane. While both a fighter plane and a commercial plane can be used to transport passengers from one location to another, a fighter plane might not be able to provide a comfortable journey as it is not well equipped with such a task. Similarly, a commercial plane can not also function as a fighter plane. While an *ad hoc* modification can be introduced in the commercial plane to function as a fighter plane (e.g. by adding missile launching capabilities), it might not be as good as the fighter plane itself in terms of performance.

Observing from a different perspective, one can always develop a general purpose plane (i.e. as a fighter and commercial plane). However, it is expected that the design of such a plane might be inefficient and bulky as it includes significant functionalities that are not always in use depending on where the plane is deployed (i.e. either as a commercial plane or a fighter plane). As the analogy illustrates, it seems appropriate that the scope of coverage of PMLs and workflow languages be kept separate. Nevertheless, while the requirements of PMLs and workflow languages might be domain specific and context dependent, synergies and experiences from either domain might still be useful for the betterment of both.

## 5. Conclusion

This paper has demonstrated the use of VRPML in a different domain that it is design for, that is, to model and enact workflow activities involving planning a trip problem. While VRPML appears to have sufficiently rich notations to model such a problem, we felt that more experimentation will be needed to establish its suitability as a workflow language. For this reason, research into the applicability of VRPML as a workflow language is still worthwhile.

## Acknowledgement

The work undertaken in this research is sponsored by the USM Short Term Grants – “The Design and Implementation of the VRPML Runtime Environment”.

## 7. References

- [1] DE BELLIS, M., HAAPALA, C., “User-Centric Software Engineering” *IEEE Expert*, February 1995, pp. 34-41.
- [2] BERTINO, E., JAJODIA, S., MANCINI, L., and RAY, I., “Multiform Transactional Model for Workflow Management”. In *Proc. of the NSF Workshop on Workflow and Process Automation in Information Systems*, May 1996.
- [3] DOPPKE, J.C., HEIMBIGNER, D., WOLF, A.L., “Software Process Modeling and Execution within Virtual Environments”. *ACM Transactions on Software Engineering and Methodology*, 7 (1), 1998, pp. 1-40.

- [4] KELLNER, M.I., FIELER, P.H., FINKELSTEIN, A., KATAYAMA, T., OSTERWEIL, L.J., PENEDO, M.H., ROMBACH, H.D., "Software Process Modeling Example Problem". In *Proc. of the 6th Intl. Software Process Workshop*, Hakodate, Hokkaido, Japan, October 1990), IEEE Computer Society Press.
- [5] WISE, A., "Little JIL 1.0 Language Report - Technical Report UM-CS-1998-024", Dept. of Computer Science, Univ. of Massachusetts at Amherst, April 1998.
- [6] WISE, A., LERNER, B.S., MCCALL, E.K., OSTERWEIL, L.J., SUTTON JR, S.M., "Specifying Coordination in Processes Using Little-JIL - Technical Report UM-CS-1999-071", Dept. of Computer Science, Univ. of Massachusetts, Amherst, November 1999.
- [7] WISE, A., CASS, G., LERNER B.S., MCCALL, E.K., OSTERWEIL, L.J., SUTTON JR., S.M., "Using Little-JIL to Coordinate Agents in Software Engineering", In *Proc. of the Automated Software Engineering Conf. (ASE 2000)*, Grenoble, France, pp. 155-163, September 2000, IEEE CS Press.
- [8] ZAMLI, K.Z., LEE, P.A., "Taxonomy of Process Modeling Languages". In *Proc. of the ACS/IEEE Intl. Conf. on Computer Systems and Applications*, pp. 435-437, 2001. IEEE CS Press.
- [9] ZAMLI, K.Z., "Process Modeling Languages: A Literature Review". *Malaysian Journal of Computer Science*, 14 (2), December 2002, pp. 26-37.
- [10] ZAMLI, K.Z., LEE, P.A., "Exploiting a Virtual Environment in a Visual PML". In *Proc. of the 4th Intl. Conf. on Product Focused Software Process Improvements (PROFES02)*, pp. 49-62, No. 2559, LNCS, Rovaniemi, Finland, December 2002, Springer.
- [11] ZAMLI, K.Z., LEE, P.A., "Modeling and Enacting Software Processes Using VRPML". In *Proc. of the 10th IEEE Asia-Pacific Conf. on Software Engineering*, pp. 243-252, December 2003, IEEE CS Press.
- [12] ZAMLI, K.Z., MAT ISA, N.A., "A Survey and Analysis of Process Modeling Languages". *Malaysian Journal of Computer Science*, 17 (2), December 2004, pp. 68-89.
- [13] ZAMLI, K.Z., MAT ISA, N.A., "The Computational Model for a Flow-based Visual Languages". In *Proc. of AIDIS Intl. Conf. in Applied Computing 2005*, Algarve, Portugal, pp.217-224.

# The Applicability of VRPML for Supporting Distributed Software Engineering Teams

Kamal Zuhairi Zamli, Nor Ashidi Mat Isa

*Software Engineering Research Group  
School of Electrical and Electronic Engineering,  
Universiti Sains Malaysia, Engineering Campus,  
14300 Nibong Tebal, Seberang Perai Selatan, Pulau Pinang, Malaysia  
Tel: 604-5937788 ext 6079, Fax: 604-5941023,  
E-mail: { eekamal, ashidi@eng.usm.my }*

## Abstract

This paper evaluates the applicability of a new visual process modelling language (PML), called the Virtual Reality Process Modelling Language (VRPML), for supporting the modelling and enacting of software processes in a distributed environment. VRPML serves as a research vehicle to address our main research hypothesis which suggests that a visual PML which exploits a virtual environment is useful for supporting software processes for distributed software engineering teams.

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

## 1. Introduction

Engineering as a discipline relates to the creative application of mathematical and scientific principles to devise and implement solutions to problems in our everyday lives in an economic and timely fashion. To provide a quality solution, it is not usually sufficient to focus only on the final product. Often, it is also necessary to consider the *processes* involved in producing that product [8, 9]. For example, consider an assembly of a car. From the customer's perspective, it is the final product that matters (i.e. a quality car). From an engineering perspective, such quality could not be achieved if some of the processes (e.g. assembly lines) are faulty. Although additional rework can fix the problems caused by the faulty assembly lines, this tends to raise the overall costs because it deals only with symptoms of the problem. In contrast, going to the cause of the problem and improving the process (e.g. the faulty assembly lines) avoids the introduction of quality defects in the first place and leads to better results with lower costs. As this example illustrates, it is through the processes that engineers can observe and improve quality, control production costs and possibly reduce the time to market their products.

Similar analogies can be applied in the case of software engineering. To produce quality software, it is also necessary to place emphasis on the processes by which the software is produced. In software engineering, these processes are usually called software processes.

A software process can be viewed as a partially ordered set of activities that must be undertaken by software engineers to manage, develop, maintain and evolve software systems. To allow better control of a particular software process, a model of that process (called a process model) can be created using a PML making the process explicit and open to examination. Through enactment (or execution) of the process model, automation, guidance and enforcement of the policy embedded in a particular process model can be usefully achieved.

Although there has been much fruitful research into PMLs, their adoption by industry has not been widespread [5]. While the reasons for this lack of success may be many and varied, our earlier work [12-16] identified two areas in which PMLs may have been deficient: human dimension issues in terms of the support for awareness and visualization as well as the support for addressing management and resource issues that might arise dynamically when a process model is being enacted. In order to address some of these issues, a new visual PML called Virtual Reality Process Modeling Language (VRPML) has been developed and evaluated [14-16]. VRPML serves as a research vehicle for addressing our main research hypothesis that a PML, which exploits a virtual environment, is useful to support software processes for distributed software engineering teams.

This paper is organized as follows. Section 2 gives highlights of the main syntax and semantics of VRPML. Section 3 demonstrates the main features of VRPML through the experimental setup. Section 4 discusses the evaluation of VRPML in the context of supporting

distributed software engineering teams. Section 5 outlines the conclusion of the paper.

## 2. Overview of VRPML

VRPML is a control-flow based visual PML for supporting the modeling and enacting of software processes. The main novel features of VRPML are:

- It considers virtual environments as a fundamental constituent, manipulatable as part of the construction of the process model (i.e. via features in the language) as well as being part of the runtime environment.
- It supports dynamic allocation of resources through its enactment model [14-16].

In VRPML, software processes are generically modeled. Resources (in terms of software engineers, artifacts and tools) can be dynamically assigned and customized for specific projects from a generic model.

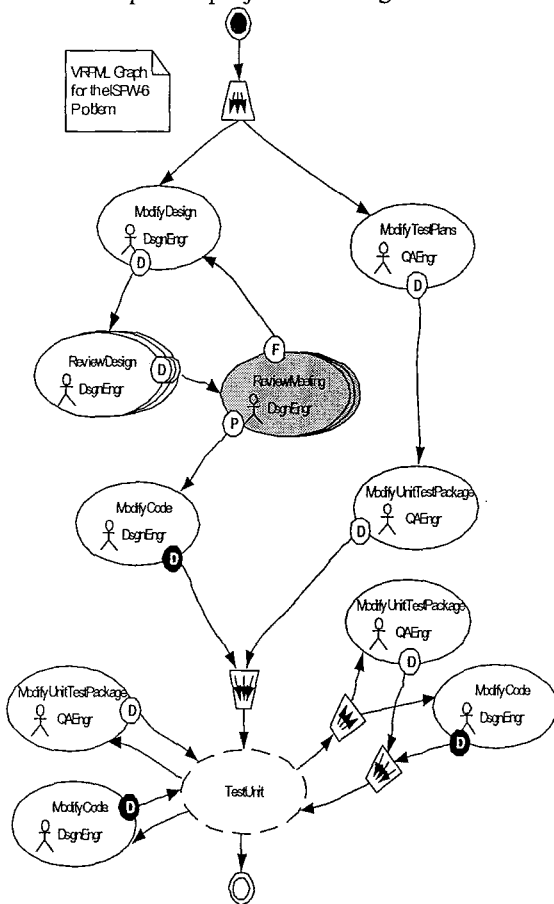


Figure 1. Excerpt from the VRPML Graph for the ISPW-6 Problem

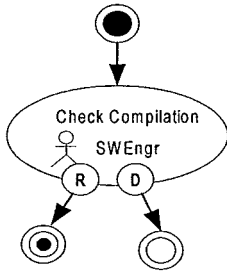
using arcs that carry run-time control-flow signals. The complete description of the syntax and semantics of VRPML can be found in [14].

As an illustration, Figure 1 presents an excerpt of the VRPML solution to a benchmark process, i.e. the ISPW-6 problem [7]. Similar to JIL [10] and Little JIL [11], software processes in VRPML are described using process step abstractions, which represent the most atomic representation of a software process (i.e. the actual activity that software engineers are expected to perform). These activities are represented as nodes, called activity nodes (shown as small ovals with stick figures).

As depicted in Figure 1, VRPML supports many different kinds of activity nodes. They include: *general-purpose activity nodes* (shown as individual small ovals with stick figures); *multi-instance activity nodes* (shown as overlapping small ovals with stick figures); and *meeting activity node* (shown as small and shaded overlapping ovals with stick figures). Both multi-instance activity nodes and meeting activity nodes have associated depths, indicating the actual number of engineers involved (and also the number of identical activities in the case of multi-instance activity).

The firing of activity nodes is controlled by the arrival of a control flow signal. In VRPML, an initial control flow signal is always generated from a *start node* (a white circle enclosing a small black circle). A *stop node* (a white circle enclosing another white circle) does not generate any control flow signals. Control flow signals may also be generated at the completion of a node, often from special completion events called *transitions* (shown as small white circles with a capital letter, attached to an activity node) or *decomposable transitions* (small black circles with a capital letter). Decomposable transitions enable automation scripts or sub-graphs to be specified (and executed if selected) as post-conditions before allowing transition to generate a control flow signal. The sub-graph associated with the decomposable transition representing Done (labeled D) for the activity node called Modify Code is given in Figure 2.

Software processes are specified in VRPML as graphs, by interconnecting nodes from top to bottom

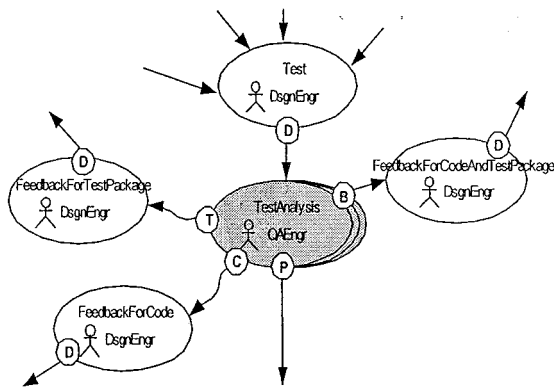


**Figure 2. Sub-graph for Decomposable Transition labeled D in Modify Code**

When Check Compilation fails, the assigned software engineer can select the transition R (for redo). As a result, a control-flow signal will be generated to re-enact its parent node (i.e. Modify Code) through a *re-enabled node* (shown as two white circles enclosing black circle). Otherwise, if the compilation is successful, the assigned engineer can select the transition D (for Done). In this case, the control-flow signal will be generated and propagated back to the main graph to enable the subsequent connected node.

In VRPML, activity nodes can also be enacted in parallel using combinations of language elements called *merger* and *replicator* nodes (shown as trapezoidal boxes with arrows inside).

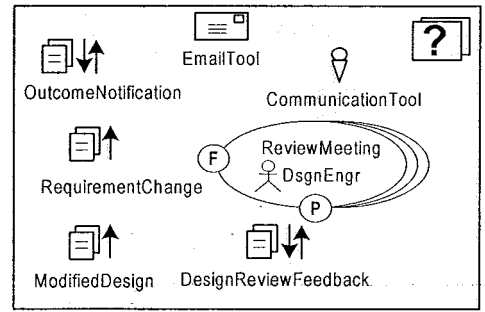
To improve readability, a set of VRPML nodes can be grouped together and replaced by a *macro node* (shown as dotted line ovals), with the macro expansion appearing on a separate graph. For example, referring to Figure 1, Test Unit is a macro node. The macro expansion of Test Unit is given in Figure 3.



**Figure 3. Macro Expansion for Test Unit in Figure 1**

For every activity node, VRPML provides a separate *workspace*, the concept borrowed from ADELE-TEMPO [1], APEL [2] and MERLIN [6].

Figure 4 depicts the sample workspace for the activity node called Review Meeting in Figure 1. A workspace typically gives a *work context* of an activity as it hosts resources needed for enacting the activity: transitions, artifacts (shown as overlapping two overlapping documents with arrows for depicting access rights), communication tools (shown as a microphone, and an envelope), and any task descriptions (shown as a question mark). Effectively, when an activity is undertaken, the workspace is mapped into a virtual room, transitions into buttons, and artifacts, communication tools (i.e. for synchronous and asynchronous forms of communications) and task description into objects which can be manipulated by software engineers to complete the particular task at hand. This mapping is based on Doppke's task-centered mapping described in [3].



**Figure 4. Sample Workspace for Activity Node Review Meeting from Figure 1**

As part of its enactment model, VRPML relies on its resource exception handling mechanism. In VRPML, resources include roles assignment, artifacts and tools (including communication tools) in a workspace as well as the depths of multi-instance activity nodes and meeting activity nodes. Depending on the needs of a particular software development project, these resources can either be allocated during graph instantiation or dynamically during graph enactment.

Upon the arrival of the control-flow signal, an activity node will be enabled. Here, the VRPML interpreter attempts to acquire resources that the activity node needs. If resources are successfully acquired, the VRPML interpreter then instantiates the activity corresponding to that activity node. If for any reason VRPML fails to acquire the resources, enactment will be blocked until such resources are made available (e.g. an engineer has not been assigned to the activity). In this way, the VRPML's resource exception handling mechanism is similar to blocking primitives (e.g. in, read) in Linda [4]. Once enactment

is blocked, the VRPML interpreter automatically produces an activity for the administrator (e.g. process engineer) to rectify the resource exception or completely terminate the current activity. If that activity is terminated, the administrator may optionally terminate the overall enactment of the particular VRPML graph in question or manually re-enact connecting nodes by providing the necessary control-flow signals that they need to fire. If the resource exception is rectified, normal enactment of the particular VRPML graph can be resumed resulting in the activity being assigned to the appropriate software engineer. When that engineer selects that particular activity, a workspace for that activity will appear as a virtual room with artifacts, transitions and communication tools as objects which software engineer can manipulate to complete the task. Finally, the activity completes when the software engineer selects one of the possible transitions (e.g. passed,

failed, done, or aborted).

### 3. Demonstration

The main aim of this section is to demonstrate that faithful enactment of the process model expressed in VRPML can be achieved. Considering this aim, the objectives are:

- To demonstrate that enactment of the process model expressed in VRPML can be achieved in a distributed environment
- To demonstrate that dynamic creation of tasks and allocation of resources can be supported by exploiting the enactment model
- To demonstrate that integration with a virtual environment is possible at the PML enactment level. Thus, awareness and visualisation issues can be supported.

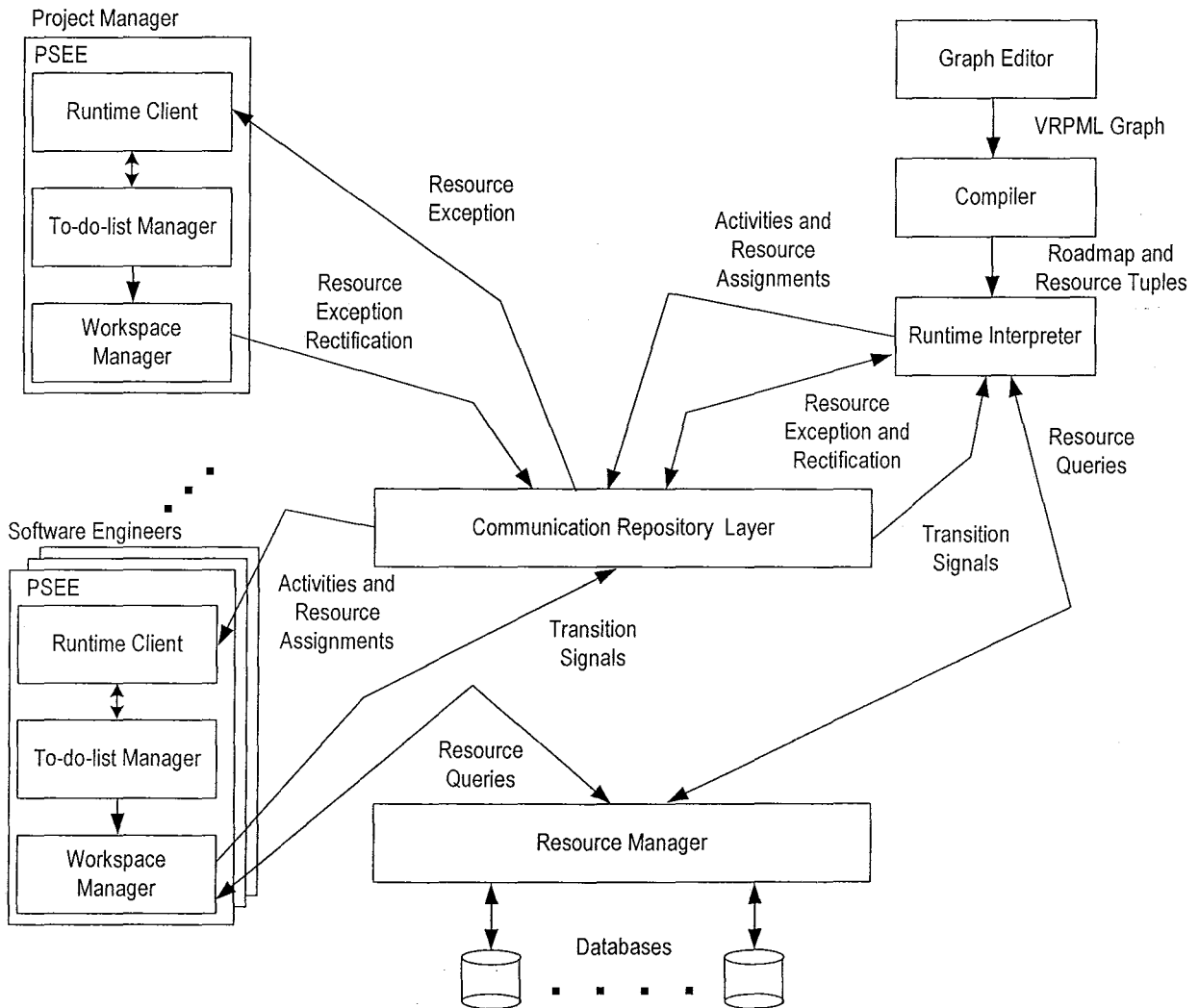


Figure 5. The VRPML Support Environment



In order to implement VRPML, a number of components for the support environment can be identified. These components and their interactions are shown earlier in Figure 5.

Briefly, the main components of the complete VRPML support environment are:

- Graph Editor – allows the VRPML graphs to be specified.
- Compiler – compiles the VRPML graphs into an immediate format for enactment.
- Runtime Interpreter – interprets the compiled VRPML graph.
- Runtime Client – retrieve activities and resource assignments from the communication repository layer.
- To-do-list Manager – manages the activities assigned to a particular software engineer.
- Workspace Manager – manages activity workspace in a virtual environment, manages activity transition, and forward queries to the resource manager.
- Communication Repository Layer – allows communication between the runtime interpreter, runtime client, and workspace manager.
- Resource Manager – queries the databases for artifacts.

A more detail description of the implementations and functionalities of these components are beyond the scope of this paper and are discussed elsewhere in [16]. It must also be noted that the prototype VRPML support environment in its current form is not suitable to support the real-world software engineering activities, and its sole purpose is to gain insights into the actual enactment.

As far as the experimental setup is concerned, the VRPML process model for the ISPW-6 problem was adopted for enactment, although only partial enactment will be demonstrated here.

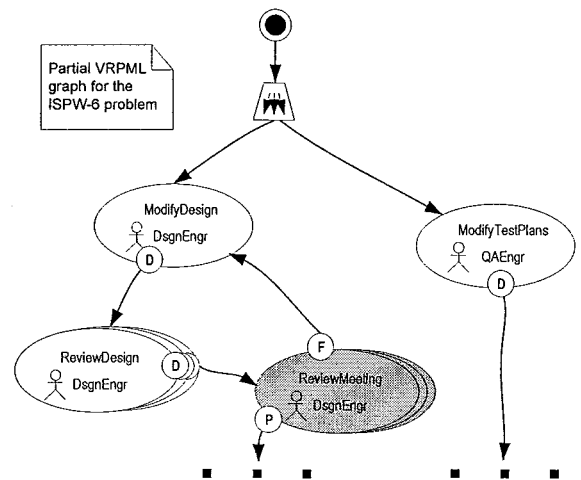


Figure 6. Partial Enactment of the ISPW-6 Problem

Referring to Figure 6, enactment will be demonstrated for the following activities: Modify Design; Modify Test Plans; Review Design; and Review Meeting. Furthermore, only resource allocation via role assignment will be considered.

It is assumed that the above activities involve three software engineers who will be taking on different roles – they are: Kamal, Pete, and Jon. Kamal and Jon will be taking the role of design engineers (abbreviated as DsgnEngr). Pete will be taking the role of quality assurance engineer (abbreviated as QAEngr) and administrator (or process engineer). It is also assumed that Modify Design is pre-assigned to Jon whilst Modify Test Plans, Review Design, and Review Meeting are dynamically assigned. Finally, Kamal, Pete, and Jon are physically isolated, that is, each of them has access to their to-do-list from a separate machine in a distributed environment.

Enactment starts when the start node produces the necessary control-flow signal. In turn, this control-flow signal will cause the replicator node to produce two more control-flow signals. Upon receiving these two control-flow signals, the interpreter queries the resources assignments for Modify Design, and Modify Test Plans in order to put them in the communication repository layer. Modify Design has already been assigned to Jon, but a resource exception will be thrown for Modify Test Plans. As a result, Modify Test Plans will be automatically assigned to the administrator (i.e. Pete) so that the resource exception can be rectified. Modify Design and Modify Test Plans will appear on Jon's to-do-list and the administrator's to-do-list respectively as soon as they made the request to retrieve the activity in the communication repository

layer. As an illustration, Figure 7 depicts Jon's to-do-list.

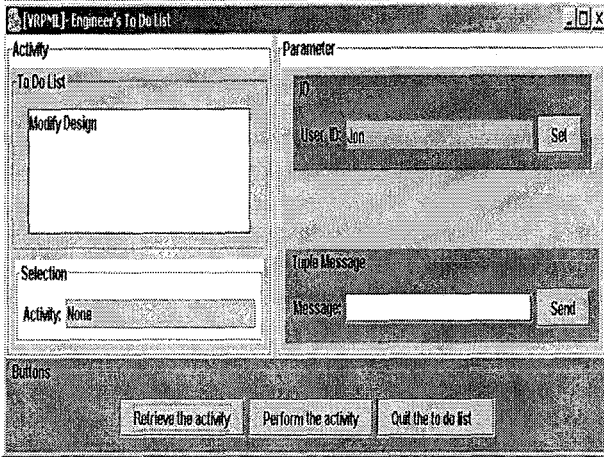


Figure 6. Jon's To-Do-List

When Jon selects and undertakes Modify Design from his to-do-list, a workspace for Modify Design is automatically opened in a virtual environment as shown in Figure 7.

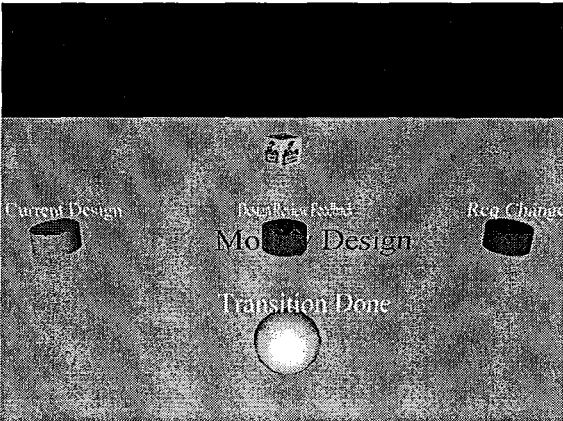


Figure 7. Workspace for Modify Design

In this case, the workspace defines transitions, tools, and artifacts for performing the activity Modify Design. Because the mapping of transitions, tools, and artifacts in the workspace is straightforward, it will not be discussed further here.

However, the workspace for Modify Test Plans requires further discussion to illustrate how the rectification of resource exception achieves dynamic allocations of resources. When the administrator selects and undertakes Modify Test Plans from his to-do-list, a workspace to rectify the resource exception

can be opened in a virtual environment. As shown in Figure 8 below, this experimental setup simply uses a text editor to facilitate the updating of resources assignment. Here, the assigned engineer has been allocated to Kamal.

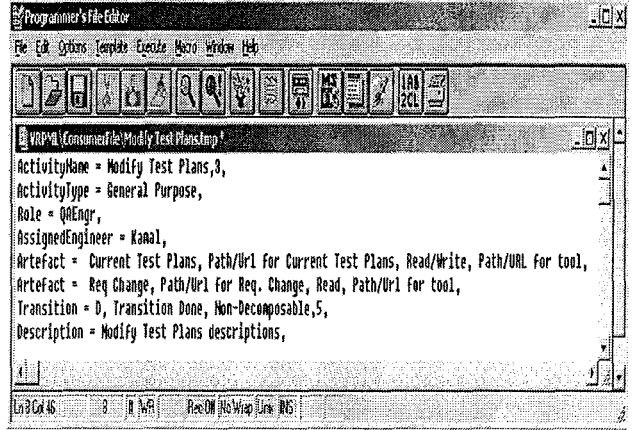


Figure 8. Resource Allocations for Modify Test Plans

Once resource allocation has been completed, Modify Test Plans will be put back into the communication repository layer. This is achieved via the administrator's to-do-list shown in Figure 9. Ideally, the administrator's to-do-list would be no different to an ordinary to-do-list, as the resource update should be done automatically in the background by the workspace manager. However, the administrator's to-do-list GUI is tailored to allow the resource update to be put back manually into the communication repository layer through the update button.

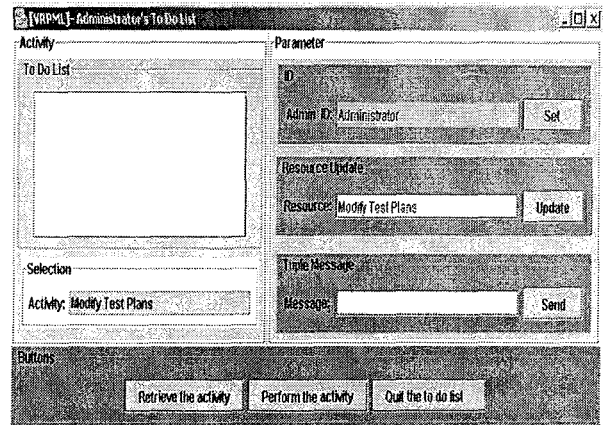


Figure 9. Administrator's To-Do-List

Once the resource allocation for Modify Test Plans has been updated, Modify Test Plans is now assigned to Kamal. Consequently, when Kamal makes the

request to retrieve the activity from the communication repository layer, Modify Test Plans will appear in Kamal's to-do-list.

Going back to Jon, once he has completed Modify Design and selects the done transition (simulated by the Send button), another control-flow signal will be generated in the communication repository layer. Upon receiving this control-flow signal, the interpreter queries the resources assignments for Review Design. As no resource assignments have been made, a resource exception will be thrown causing Review Design to be assigned to the administrator. In turn, when the administrator makes the request to retrieve the activity from the communication repository layer, Review Design will appear in the administrator's to-do-list. Similar to the case of Modify Test Plans discussed earlier, in order to rectify this resource exception there is a need to update the resource tuple for Review Design. As Review Design is a multi-instance activity node, it can be assigned to more than one software engineer through changing its depth, as illustrated in Figure 10.

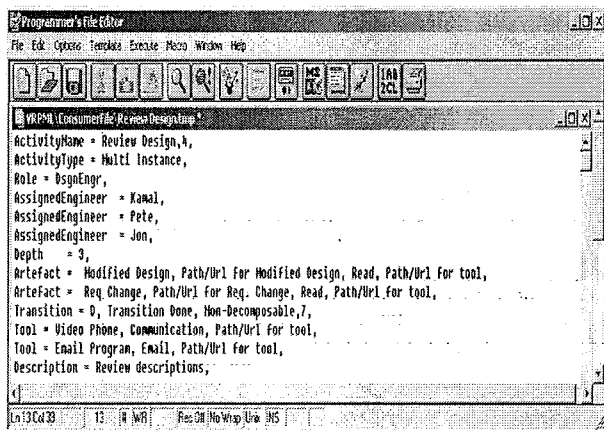


Figure 10. Resource Allocations for Review Design

By manipulating the Review Design's depth, the number of software engineers required to review the design (or how many Review Design activities are created) can be tailored according to the current needs of the project. In this example, Review Design is assigned to three software engineers: Pete, Kamal, and Jon. Once resource allocation for Review Design has been completed, it will be put back into the communication repository layer through the administrator's to-do-list. As a result, Review Design will appear in the to-do-list for Pete, Kamal, and Jon when they make the requests to retrieve the activity from the communication repository layer.

In order to inculcate the sense of process awareness, the virtual environment representing workspaces for a multi-instance activity node such as Review Design has a different appearance to a workspace for a general purpose activity node (see Figure 7) in terms of the background of the workspace.

As far as the completion of Review Design is concerned, because it is a multi-instance activity node assigned to Pete, Kamal, and Jon, all of them must complete the review by selecting the Done transition in their own separate workspaces. Only after all of the done transitions have been selected can a new control-flow signal be generated in the communication repository layer to enable the subsequent activity Review Meeting.

Finally, upon receiving the control flow signal generated above, the interpreter queries the resources assignments for Review Meeting (see Figure 6). As no resource allocations have been made, a resource exception will be thrown causing Review Meeting to be assigned to the administrator. Consequently, when the administrator makes the request to retrieve the activity from the communication repository layer, Review Meeting will appear in his to-do-list.

Being a meeting activity node, Review Meeting also has an associated depth which can be manipulated in order to allocate engineers dynamically based on the needs of the activity. Using the allocation mechanism described above, the administrator can assign Review Meeting to Pete and Jon, with Pete being the moderator.

As far as the workspace is concerned, being a meeting activity node, Review Meeting can have a different appearance as compared to other types of activity nodes, again, to inculcate the sense of process awareness. In terms of the transitions associated with Review Meeting, these are only accessible to Pete as he is the moderator. Therefore, it is Pete who has the final say of whether the modified design is endorsed or rejected, and only one control-flow signal will be generated as a result.

If Pete decides to choose the Failed transition, the interpreter reassigns Modify Design to Jon. Assuming Jon is still part of the development team, Modify Design will appear in Jon's to-do-list after he makes the request to retrieve the activity from the communication repository layer. Otherwise, the resource exception will be thrown. After Jon completes Modify Design, Review Design will now be reassigned to Pete, Kamal, and Jon. After Pete, Jon, and Kamal

complete Review Design, Review Meeting will be reassigned to Pete and Jon. This “looping” sequence of activities will continue until Pete, being the moderator, selects the Passed transition after completing Review Meeting.

Overall, the experiment has successfully achieved its objective of demonstrating enactment in a distributed environment of a process model expressed in VRPML. Furthermore, the experiment has also demonstrated the VRPML support for the dynamic allocation of resources as well as highlighted the possible support for visualization and awareness issues. Hence, it is believed that this experiment has demonstrated that VRPML can be used in practice to express a process model and support its enactment.

#### 4. Discussion

In line with the main research hypothesis discussed earlier, this section debates the applicability of VRPML for supporting distributed software engineering teams. In doing so, this section identifies some of the difficulties associated with distributed software processes, and analyses whether or not the features provided in VRPML addresses those difficulties.

Due to the lack of face-to-face contact, coordination of activities involved in a software process is often difficult when the development teams are not physically collocated. The fact that VRPML supports the construction of the process model as well as its enactment in a distributed environment is helpful in this situation. One reason is that the coordination of activities can be fully automated through enactment.

Another common problem arising from the lack of face-to-face contact relates to communication breakdown amongst the team members. Generally, communication breakdown has a negative effect on the developed software, resulting in bugs and unnecessary rework. As a consequence, the probability of development project success can be significantly reduced. Although not fully implemented in the current prototype, the support for awareness in VRPML may be helpful to address some of these issues. This is because through awareness, group cohesion may be improved, and hence encourage informal communication amongst team members.

Nevertheless, communication amongst team members can often be difficult when the development teams are distributed in multiple sites. Asynchronous communication tools (e.g. email tools) address this

issue to a certain degree, but do not allow software engineers to hold the rich discussions possible when they are physically collocated. Thus, the feature of VRPML that permits the specification of synchronous communication tools (e.g. a tele-conferencing program) as part of the workspace definition can be helpful to address this issue. Furthermore, VRPML also provides a special node for virtual meetings. The support for virtual meetings is beneficial since meetings are an important characteristic of software engineering. Additionally, when development teams are distributed over multiple sites, virtual meetings could help reduce costs if a meeting would otherwise have to be held face-to-face.

In the context of distributed software engineering teams affected by both geographical and temporal distribution, collaboration on a shared activity can also be difficult to achieve. This is because there may be only a small window of overlap in term of times when the team members can work together. In some cases, there could also be absolutely no window of overlap at all. The fact that VRPML permits dynamic allocation of resources might be convenient to address some of the above issues. One reason is that the assignment of engineers as resources to a shared activity can be made dynamically not only based on the availability of engineers but also depending on whether or not there is a temporal overlap for team members to collaborate.

#### 5. Conclusion

As has been shown, some of the features of VRPML can be usefully exploited to address some of the problems associated with distributed software processes. Therefore, it can be concluded that VRPML is useful for supporting software processes for distributed software engineering teams.

#### Acknowledgement

The work undertaken in this research is partially funded by the USM Short Term Grants – “The Design and Implementation of the VRPML Runtime Environment”.

#### References

- [1] N. Belkhatir, J. Estublier, and W. Melo. ADELE-TEMPO: An Environment to Support Process Modelling and Enaction. in Nuseibeh, B. ed. *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, 187-222.

- [2] S. Dami, J. Estublier, and M. Amiour. "APEL: A Graphical Yet Executable Formalism for Process Modeling". *Automated Software Engineering*, 5 (1), 1998, 61-96.
- [3] J.C. Doppke, D. Heimbigner, and A.L. Wolf. "Software Process Modeling and Execution within Virtual Environments". *ACM Transactions on Software Engineering and Methodology*, 7 (1), 1-40.
- [4] D. Gelernter. "Generative Communication in Linda". *ACM Transactions on Programming Languages and Systems*, 7 (1), 1985, pp. 80-112.
- [5] 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 (EWSPT 2000)*, Kaprun, Austria, February 2000, Lecture Notes in Computer Science Volume 1780, Springer, 96-108.
- [6] 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, 103-129.
- [7] M.I. Kellner, P.H. Feiler, A. Finkelstein, T. Katayama, L.J. Osterweil, M.H. Penedo, and H.D. Rombach. "Software Process Modeling Example Problem". In *Proc. of the 6th Intl. Software Process Workshop*, Hakodate, Hokkaido, Japan, October 1990. IEEE CS Press.
- [8] L.J. Osterweil. Software Processes are software too, revisited. In *Proc. of the 19th IEEE Intl. Conf. on Software Engineering*, Boston, USA, 1997, IEEE CS Press, 540-548.
- [9] L.J. Osterweil. Software Processes are software too. In *Proc. of the 9th IEEE Intl. Conf. on Software Engineering*, Monterey, USA, 1987, IEEE CS Press, 2-13.
- [10] S. Sutton Jr., and L.J. Osterweil. The Design of a Next-Generation Process Language. in *Proc. of the Joint 6th European Software Engineering Conference and the 5th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, (1997), Lecture Notes in Computer Science Volume 1301, Springer, 142-158.
- [11] A. Wise. "Little JIL 1.0 Language Report - Technical Report 98-24", Dept. of Computer Science, Univ. of Massachusetts at Amherst, April 1998.
- [12] 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, 435-437.
- [13] K.Z. Zamli. "Process Modeling Languages: A Literature Review". *Malaysia Journal of Computer Science* 14, 2 (December 2001)
- [14] 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 (PROFES02)*, Lecture Notes in Computer Science Volume 2559, Rovaniemi, Finland, 2002, Springer, 49-62.
- [15] 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, December 2003, IEEE CS Press, 243-252.
- [16] K.Z. Zamli. "Supporting Software Processes for Distributed Software Engineering Teams", School of Computing Science, Univ. of Newcastle upon Tyne, PhD Thesis, October 2003.

# THE COMPUTATIONAL MODEL FOR A FLOW-BASED VISUAL PML

Kamal Zuhairi Zamli

*School of Electrical and Electronics  
USM Engineering Campus, 14300 Nibong Tebal  
Pulau Pinang, Malaysia  
Email: eekamal@eng.usm.my*

Nor Ashidi Mat Isa

*School of Electrical and Electronics  
USM Engineering Campus, 14300 Nibong Tebal  
Pulau Pinang, Malaysia  
Email: ashidi@eng.usm.my*

## ABSTRACT

This paper discusses the three possible computational models for a flow-based visual process modeling language (PML): the data-flow model; the control-flow model; and the combined model. In doing so, the suitable computational model is chosen, evaluated, and applied into the design of a new visual PML, called Virtual Reality Process Modeling Language (VRPML), taking into accounts the need to support the modeling and enacting of software processes.

## KEYWORDS

Software Engineering, Software Processes, Process Modeling Languages, Visual Languages

## 1. INTRODUCTION

Visual programming languages have been around for quite some time now. The basic idea behind a visual programming language is that computer graphics (e.g. graphs consisting of icons, nodes, and arcs) are used instead of a textual representation. In fact, the central argument for a visual programming language is based on an observation that picture is better than text (i.e. a picture is worth a thousand words – (Whitley, 1997)).

While a visual programming language may not be able to provide a silver bullet to solve every problem related to engineering a software system, a carefully chosen level of abstractions (e.g. by working at the same level of abstraction as the problem domain) coupled with easy to understand notations may help alleviate the low-level complexities offered by the textual counterpart. For this reason, visual programming language has been considered as one of the important issues in the design of VRPML, a domain specific visual PML developed as part of our on-going research (Zamli and Lee, 2001; Zamli, 2002; Zamli and Lee, 2002; Zamli and Lee, 2003; Zamli and Mat Isa, 2004).

Motivated by a number of existing visual PMLs, for example, FUNSOFT Nets (Dami et al, 1998), Dynamic Task Nets (Heiman et al, 1996), APEL (Dami et al, 1998), EVPL (Grundy and Hosking, 1998), PROMENADE (Ribo and Franch, 2000; Franch and Ribo, 2003), and Little JIL (Wise, 1998), a flow-based visual language paradigm seems to be a suitable choice for VRPML. Based on this choice, this paper discusses the three possible computational models for VRPML: the data-flow model; the control-flow model; and the combined model. In doing so, the suitable computational model is chosen, evaluated, and applied into the design of VRPML, taking into accounts the need to support the modeling and enacting of software processes.

This paper is organized as follows. Section 2 gives an overview of flow-based visual languages. Section 3 debates the suitable computational model for a flow-based visual language such as VRPML. Section 4 gives

an overview of VRPML. Section 5 discusses the lessons learned. Finally, section 6 presents the conclusions of the paper.

## 2. FLOW BASED VISUAL LANGUAGES

Normally, flow-based visual languages are based on directed graphs. Graphs typically consist of nodes, arcs and sub-graphs. Nodes represent function or actions, arcs carry data or control-flow signals, and sub-graphs provide abstraction and modularization. Operations in graphs follow a *firing rule* which defines the conditions under which execution of node occurs.

In the control-flow based model, a visual program consists of nodes connected by arcs carrying control-flow signals. Arcs depict the control-flow dependencies amongst connected nodes. The firing rule is based solely on the availability of the control-flow signals on the node's input arcs – that is, data availability does not play any part at all.

Conceptually, in the control-flow based model, every program can be thought of as having an instruction counter and a globally addressable memory which holds programs and data objects whose contents are updated by program instructions during execution (Ackerman, 1982). As far as a visual language associated with the control-flow model is concerned, for simplicity, it may be viewed as supporting executable flowcharts.

In the data-flow based model, a visual program consists of nodes connected by arcs carrying data. Arcs depict data dependencies amongst nodes. The firing rule is based on the availability of data on the node's input arcs, and may be *data-driven* or *demand-driven*. With a data-driven firing rule, an arc is used as a supply route to transmit data from the source node to the destination node. A destination node is executed as soon as data is available on all input arcs. With a demand-driven firing rule, an arc is used as a demand route to request data from the source node. A source node is executed only if there is a demand for its result. For either firing rule, arcs are conduits for data. In turn, data on an arc is consumed by the executing node to perform its computation (although some variations of the data-flow based model also allow an arc to retain data).

According to (Agerwala and Arvind, 1982), the data-flow based model can be distinguished from the control-flow based model in that it has neither a globally addressable memory nor a single instruction counter. As the data-flow based model possesses no global memory, the only data available to a node for its operation is that from its inputs. In addition, because of the lack of any shared data amongst nodes, there can be no *side effects* (one node interfering with other node's data, potentially causing unexpected results).

As the data-flow firing rule depends solely on the availability of data, nodes whose data is available can potentially be executed in parallel. The sequencing of the execution of nodes, for example in terms of the assignment of runtime processes to processors, is determined solely at runtime by the runtime system. Thus, a data-flow based model supports parallelism naturally.

Apart from the control-flow or the data-flow based models, one less popular paradigm is the computational model based on both models. Here, there are two kinds of arcs with different semantics: the data-flow and the control-flow arc. The firing rule for this paradigm can be complex because it is based on the combination of both the data-flow and the control-flow signals. Furthermore, while the problem of arcs crossing each other and resulting in a cluttered view is inherent in a flow based visual language based on directed graphs, the fact that two arcs are used here means that the crossover problem can be even greater. Generally, if there are too many arc crossovers, the overall program understanding may be compromised. Therefore, the option of combining the control and data-flow models will not be considered as the computational model for VRPML.

Having disregarded the option of adopting both the data-flow and the control-flow as the computational model for VRPML, there are now only two choices of the computational model: the control-flow model; or the data-flow based model. The selection of the suitable computational model for VRPML will be discussed next.

## 3. DATA FLOW VERSUS CONTROL FLOW

In order to support the modeling and enacting of software processes, there is a need for VRPML to support cyclic behavior. This need can be seen, for example, in the case where software design fails its review. In the case of the control-flow model, only one control-flow signal is needed to enable the previous node regardless of the number of data items required for that node. However, in the case of the data-flow based model, to enable the previous node requires all the data for that node to be available. Thus, while both the control-flow and the data-flow based model can address this need in a straightforward manner, it seems somewhat easier in terms of implementation to use the control-flow rather than the data-flow model.

In addition to being able to support cyclic behavior, there is also a need for VRPML to facilitate reasoning about its execution semantics, hence making enactment of a software process traceable. It is useful to be able to trace the enactment of the software process to facilitate process awareness and process understanding which could lead to improved process support. Because the ordering and enactment of tasks in the data-flow based model strictly depends on the availability of data at runtime, enactment of a software process may be non-deterministic. Hence, it is difficult to reason about and to trace enactment in the data-flow model compared to the control-flow model.

Because some activities in a software process need to be undertaken by more than one software engineer, there is also a need for VRPML to support shared artifacts. For example, in the case of an activity such as modify code, it may be that more than one software engineer is assigned to change the same source code at the same time. Although both the control-flow and the data-flow based models can address this need, it seems preferable to use the control-flow rather than the data-flow model. Because the data-flow model inherently avoids the problem of side effects – that is, one node interfering with other node's data, there is a need for copies of data to be replicated across nodes to enable shared data. In fact, this need may put an extra burden for process engineers who construct the process model.

Finally, although the data-flow based model is helpful in the sense that parallelism can be achieved automatically; such a feature may not be a major benefit for enacting software processes in VRPML. As far as enactment of software processes are concerned, because software engineers are the “processors” which perform the computation, it is desirable to have their assignments under human control (e.g. under the discretion of the project manager). This is because software engineers have different skills which need to be considered before task assignments can be made. Thus, in this respect, the data-flow based model has no clear advantage over the control-flow based model.

While the data-flow based model has some merits, it is the control-flow based model that has been chosen for VRPML. This choice will be reflected in the design of VRPML discussed next.

#### 4. OVERVIEW OF VRPML

VRPML is a control-flow based visual PML for supporting the modeling and enacting of software processes. The main novel features of VRPML are:

- It considers virtual environments as a fundamental constituent, manipulatable as part of the construction of the process model (i.e. via features in the language) as well as being part of the runtime environment.
- It supports dynamic allocation of resources through its enactment model.

In VRPML, software processes are generically modeled. Resources (in terms of software engineers, artifacts and tools) can be dynamically assigned and customized for specific projects from a generic model.

Software processes are specified in VRPML as graphs, by interconnecting nodes from top to bottom using arcs that carry run-time control-flow signals. The complete description of the syntax and semantics of VRPML can be found in (Zamli and Lee, 2002).

As an illustration, Figure 1 presents an excerpt of the VRPML solution to a benchmark process, i.e. the ISPW-6 problem (Kellner et al, 1990). Similar to Little JIL (Wise, 1998), software processes in VRPML are described using process step abstractions, which represent the most atomic representation of a software process (i.e. the actual activity that software engineers are expected to perform). These activities are represented as nodes, called activity nodes (shown as small ovals with stick figures).

As depicted in Figure 1, VRPML supports many different kinds of activity nodes. They include: *general-purpose activity nodes* (shown as individual small ovals with stick figures); *multi-instance activity nodes* (shown as overlapping small ovals with stick figures); and *meeting activity node* (shown as small and shaded



overlapping ovals with stick figures). Both multi-instance activity nodes and meeting activity nodes have associated depths, indicating the actual number of engineers involved (and also the number of identical activities in the case of multi-instance activity).

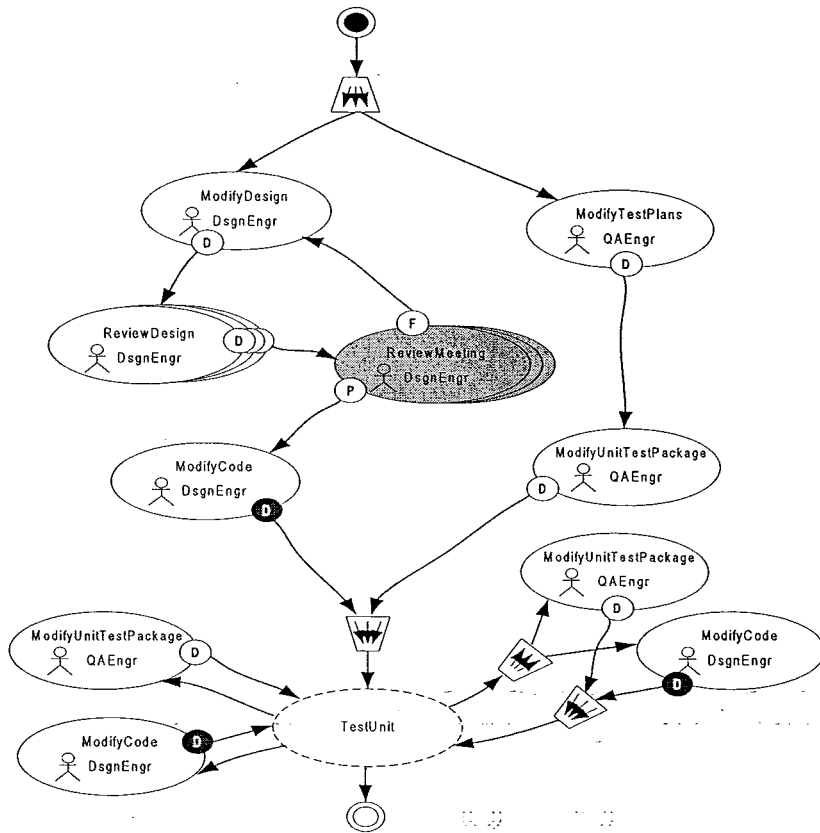


Figure 1. Excerpt from the VRPML Graph for the ISPW-6 Problem

The firing of activity nodes is controlled by the arrival of a control flow signal. In VRPML, an initial control flow signal is always generated from a *start node* (a white circle enclosing a small black circle). A *stop node* (a white circle enclosing another white circle) does not generate any control flow signals. Control flow signals may also be generated at the completion of a node, often from special completion events called *transitions* (shown as small white circles with a capital letter, attached to an activity node) or *decomposable transitions* (small black circles with a capital letter). Decomposable transitions enable automation scripts or sub-graphs to be specified (and executed if selected) as post-conditions before allowing transition to generate a control flow signal. The sub-graph associated with the decomposable transition representing Done (labeled D) for the activity node called Modify Code is given in Figure 2.

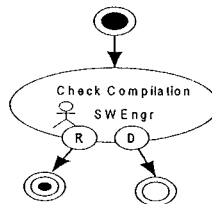


Figure 2. Sub-graph for Decomposable Transition labeled D in Modify Code

When Check Compilation fails, the assigned software engineer can select the transition R (for re-do). As a result, a control-flow signal will be generated to re-enact its parent node (i.e. Modify Code) through a *re-enabled node* (shown as two white circles enclosing black circle). Otherwise, if the compilation is successful, the assigned engineer can select the transition D (for Done). In this case, the control-flow signal will be generated and propagated back to the main graph to enable the subsequent connected node.

In VRPML, activity nodes can also be enacted in parallel using combinations of language elements called *merger* and *replicator* nodes (shown as trapezoidal boxes with arrows inside). To improve readability, a set of VRPML nodes can be grouped together and replaced by a *macro node* (shown as dotted line ovals), with the macro expansion appearing on a separate graph. For example, referring to Figure 1, Test Unit is a macro node. The macro expansion of Test Unit is given in Figure 3.

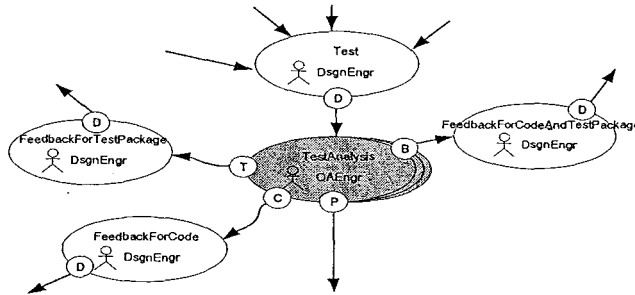


Figure 3. Macro Expansion for Test Unit in Figure 1

For every activity node, VRPML provides a separate *workspace*. Figure 4 depicts the sample workspace for the activity node called Review Meeting in Figure 1. A workspace typically gives a *work context* of an activity as it hosts resources needed for enacting the activity: transitions, artifacts (shown as overlapping two overlapping documents with arrows for depicting access rights), communication tools (shown as a microphone, and an envelope), and any task descriptions (shown as a question mark). Effectively, when an activity is undertaken, the workspace is mapped into a virtual room, transitions into buttons, and artifacts, communication tools and task description into objects which can be manipulated by software engineers to complete the particular task at hand. This mapping is based on Doppke’s task-centered mapping described in (Doppke et al, 1998).

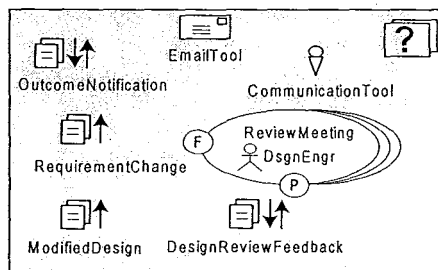


Figure 4. Sample Workspace for Activity Node Review Meeting from Figure 1

As part of its enactment model, VRPML relies on its resource exception handling mechanism. In VRPML, resources include roles assignment, artifacts and tools (including communication tools) in a workspace as well as the depths of multi-instance activity nodes and meeting activity nodes. Depending on the needs of a particular software development project, these resources can either be allocated during graph instantiation or dynamically during graph enactment. Upon the arrival of the control-flow signal, an activity node will be enabled. Here, the VRPML interpreter attempts to acquire resources that the activity node

needs. If resources are successfully acquired, the VRPML interpreter then instantiates the activity corresponding to that activity node. If for any reason VRPML fails to acquire the resources, enactment will be blocked until such resources are made available (e.g. an engineer has not been assigned to the activity). In this way, the VRPML's resource exception handling mechanism is similar to blocking primitives (e.g. in, read) in Linda (Gelernter, 1985). Once enactment is blocked, the VRPML interpreter automatically produces an activity for the administrator (e.g. process engineer) to rectify the resource exception or completely terminate the current activity. If that activity is terminated, the administrator may optionally terminate the overall enactment of the particular VRPML graph in question or manually re-enact connecting nodes by providing the necessary control-flow signals that they need to fire. If the resource exception is rectified, normal enactment of the particular VRPML graph can be resumed resulting in the activity being assigned to the appropriate software engineer. When that engineer selects that particular activity, a workspace for that activity will appear as a virtual room with artifacts, transitions and communication tools as objects which software engineer can manipulate to complete the task. Finally, the activity completes when the software engineer selects one of the possible transitions (e.g. passed, failed, done, or aborted) regardless of the outcome.

### 5. LESSONS LEARNED

Although the UML activity diagram (Rumbaugh et al, 1999) is not a PML, it can be compared to VRPML. The UML activity diagram representation of a software process is simple and intuitive. Nonetheless, while the UML activity diagram can be used to express activities in a software process, it lacks features to express the individual role, resources, work contexts, and the completion of activities. Furthermore, UML activity diagrams do not have a well-defined executable semantics (i.e. as in VRPML). A known experience of using UML as a PML can be seen in the design of PROMENADE (Ribo and Franch, 2000; Franch and Ribo, 2003). Here, the authors of PROMENADE dismiss the use of activity diagram as a PML, as PROMENADE mainly relies on class diagrams and object constraint language for supporting the modeling and enacting of software processes. Furthermore, in doing so, the authors of PROMENADE extensively extend the UML meta-models, hence, affecting the standardization of UML. For these reasons, we believe that UML is not particularly suitable as a PML.

As far as the computational model is concerned, although a computational model based on control-flow was initially chosen instead of a data-flow or a combined model, the author's experience from developing VRPML reveals the need to employ both control-flow and data-flow semantics in order to support the modeling and enacting of software processes. This is because the concepts of control-flow and data-flow are interrelated: the program control as well as the execution of an activity depends on the availability of data, and the flow of data is often governed by some notion of control.

While having control-flow firing rule at the graph level, the VRPML enactment model also relies on data dependency semantics in order to enact a particular activity. Therefore, similar to Little JIL (Wise, 1998), VRPML can also be viewed as a PML which adopts a combined model. As a comparison, Figure 5 depicts excerpt of the Little JIL solution to the ISPW-6 problem.

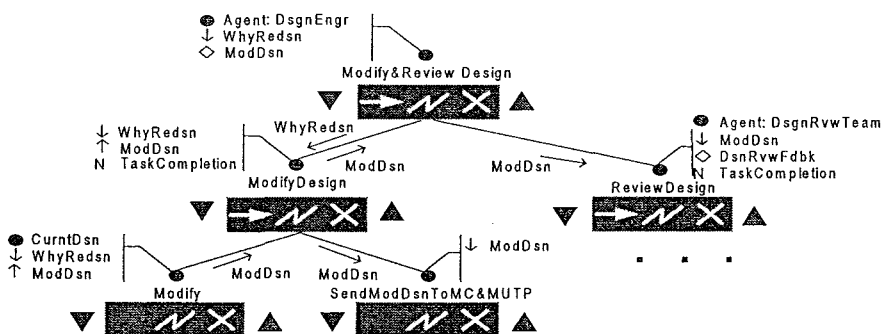


Figure 5. Excerpt from Little JIL

Unlike Little JIL, the flow of data in VRPML does not appear directly as part of the process graphs (see excerpt in Figure 1 and Figure 5). The fact that the flow of data does not appear directly as part of process graphs is significant to reduce the visual complexity of the process models expressed in VRPML as compared to Little JIL.

Furthermore, VRPML can also be compared to Little JIL in terms of the separation of concerns (Zamli and Lee, 2003). Unlike VRPML, Little JIL depicts both the control-flow and the data-flow in the same view. Although depicting both flows might be useful to give the overall context about a particular activity, a counter argument suggests that it results in a cluttered view. In VRPML, control-flow and data-flow are completely separated through the concept of workspaces, essentially giving two different views about the activities: the flow of activities and the individual activity's work context. The flow-of-activities view gives the overall dependencies amongst different activities whilst the activity's work context view gives focus of an activity in question. These views are advantageous to assist reasoning about a software process.

Demonstrated by the excerpt description of the ISPW-6 problem in shown in Figure 1, software processes tend to be imperative in nature. Therefore, the computational model based on control-flow at the graph level appears to be appropriate to express such behavior.

As the VRPML firing rule at the graph level is based on control-flow, the author's experience in the modeling of activities for the ISPW-6 problem (discussed in (Zamli and Lee, 2003)) showed the need to manually analyze the input and output dependencies in order to properly sequence those activities. If VRPML were purely based on the data-flow model, the sequencing of activities would be achieved automatically based on the availability of data at runtime. In fact, parallelism among activities can be opportunistically achieved in the same manner. Thus, in this respect, the pure data-flow model can be helpful to facilitate the construction of process models as the process engineers need not be concerned about parallelism.

Because activities specified in a process model are often performed by many engineers, there is a need for a PML to address issues relating to artifact sharing. In VRPML, artifact sharing can be flexibly achieved in the sense that activities could be tailored to deal with shared or copies of data. Obviously, there are pros and cons for either choice. If the process engineer chooses to allocate resources based on shared data, VRPML ensures that data consistency is maintained via access control mechanisms associated with artifacts. However, the disadvantage of working with shared data is that parallelism may be restricted due to runtime data "locking" imposed by access controls. If the process engineer chooses to allocate resources based on copies of data (i.e. like most pure data-flow models would), parallelism may be maximized. Nonetheless, there is a need to utilize a special tool such as the Concurrent Versions System (CVS, 2003) in order to ensure that data consistency is maintained if changes to copies of data have to be merged together.

Overall, as the above discussion illustrates, the VRPML computational model usefully balances control-flow firing rule and data dependency semantics without adding visual complexity to the language. For this reason, the VRPML computational model seems appropriate for supporting the modeling and enacting of software processes.

## 6. CONCLUSION

In conclusion, this paper discussed the three possible computational models for a flow-based visual process modeling language (PML): the data-flow model; the control-flow model; and the combined model. The suitable computational model based on the control-flow model is chosen, applied, and evaluated into the design of VRPML, and compared to other PMLs. In addition to highlighting lessons learned, this evaluation can be useful as guidance for the design of next-generation visual PMLs.

## ACKNOWLEDGEMENT

This project has been undertaken under the generous funding of the USM Short Term Grants – "The Design and Implementation of the VRPML Support Environment".

## REFERENCES

- CVS, 2003. Concurrent Versions System, <http://www.cvshome.org>.
- Ackerman, W.B., 1982. Data Flow Languages. In *IEEE Computer*, pp 15-23.
- Agerwala, T. and Arvind, 1982. Data Flow Languages. In *IEEE Computer*, pp 10-13.
- Bandinelli S. et al, 1994. SPADE: An Environment for Software Process Analysis, Design and Enactment. In A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.), *Software Process Modelling and Technology*, pp. 223-247, Research Studies Press, Taunton, England.
- Dami S. et al, 1998. APEL: A Graphical Yet Executable Formalism for Process Modelling. In *Automated Software Engineering*, Vol. 5, No. 1, pp. 61-96.
- Doppke J.C. et al, 1998. Software Process Modeling and Execution within Virtual Environments. In *ACM Transactions on Software Engineering and Methodology*, Vol. 7, No. 1, pp. 1-40.
- Emmerich, W. and Como, V.G., 1991. FUNSOFT Nets: A Petri-Net based Software Process Modeling Language. *Proc. of the 6th Intl. Workshop on Software Specification and Design*, Italy. pp. 175-184.
- Gelernter, D., 1985. Generative Communication in Linda. In *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80-112.
- Grundy, J.C. and Hosking, J.G., 1998. Serendipity: Integrated Environment Support for Process Modeling, Enactment and Work Coordination. In *Automated Software Engineering*, Vol 5, No. 1, pp. 27-60.
- Heiman P. et al, 1996. DYNAMITE: Dynamic Task Nets for Software Process Management. *Proc. of the 18th Intl. Conf. on Software Engineering*, pp. 331-341.
- Kellner M.I. et al, 1990. Software Process Modeling Example Problem. *Proc. of the 6th Intl. Software Process Workshop*, IEEE CS Press.
- Ribo J.M. and Franch, X., 2000. PROMENADE: A PML Intended to Enhance Standardization, Expressiveness and Modularity in Software Process Modelling. Research Report LSI-34-R., Lluenguatges I Sistemes Informatics, Politechnical of Catalonia, Spain.
- Rumbaugh J. et al, 1999, *The UML User Guide*. Addison Wesley.
- Whitley, K.N., 1997. Visual Programming Languages and the Empirical Evidence For and Against. In *Journal of Visual Language and Computing*, Vol. 8, No. 1, pp. 109-142.
- Wise, A., 1998. Little JIL 1.0 Language Report - Technical Report 98-24, Dept. of Computer Science, Univ. of Massachusetts at Amherst.
- Franch, X. and Ribo, J.M., 2003. A UML-Based Approach to Enhance Reuse within Process Technology. *Proc. of the 9th European Workshop on Software Process Technology*, LNCS Vol. 2786, Helsinki, Finland, pp. 74-93.
- Zamli, K.Z. and Lee, P.A., 2001. Taxonomy of Process Modeling Languages. *Proc. of the ACS/IEEE Intl. Conf. on Computer Systems and Applications*, pp. 435-437, IEEE CS Press.
- Zamli, K.Z., 2002. Process Modeling Languages: A Literature Review. In *Malaysian Journal of Computer Science*, Vol. 14, No. 2, pp. 26-37.
- Zamli, K.Z. and Lee, P.A., 2002. Exploiting a Virtual Environment in a Visual PML. *Proc. of the 4th Intl. Conf. on Product Focused Software Process Improvements (PROFES02)*, LNCS Vol. 2559, Rovaniemi, Finland, pp. 49-62.
- Zamli, K.Z. and Lee, P.A., 2003. Modeling and Enacting Software Processes Using VRPML. *Proc. of the 10th IEEE Asia-Pacific Conf. on Software Engineering*, Chiang Mai, Thailand, pp. 243-252.
- Zamli, K.Z. and Mat Isa, N.A., 2004. A Survey and Analysis of Process Modeling Languages. In *Malaysian Journal of Computer Science*, Vol. 17, No. 2, (forthcoming).

## THE DESIGN AND IMPLEMENTATION OF THE VRPML SUPPORT ENVIRONMENTS

<sup>1</sup>Kamal Zuhairi Zamli, <sup>2</sup>Nor Ashidi Mat Isa and <sup>3</sup>Norazlina Khamis

<sup>1,2</sup>Software Engineering Research Group,  
School of Electrical and Electronics,  
Universiti Sains Malaysia Engineering Campus,  
14300 Nibong Tebal, Pulau Pinang, Malaysia  
Tel: +604-5937788 ext 6079, Fax: +604-5941023  
Email: {eekamal,ashidi}@eng.usm.my

<sup>3</sup>Faculty of Computer Science and Information Technology,  
Universiti Malaya,  
50603 Lembah Pantai, Kuala Lumpur, Malaysia  
Tel: +603-76976402, Fax: +603-79676339  
Email: azlina@um.edu.my

### ABSTRACT

Software processes relate to the sequences of steps that must be performed by software engineers in order to pursue the goals of software engineering. In order to have an accurate representation and implementation of what the actual steps are, software processes may be modeled and enacted by a process modeling language (PML) and its process support system (*called the Process Centered Environments i.e. PSEE*). Although there has been much fruitful research into PMLs, their adoption by industry has not been widespread. Furthermore, no single PML and PSEE have assumed dominance and accepted as the *de facto standard*. For these reasons, research into PMLs and PSEEs are still necessary.

This paper discusses the design of the process support environment for a new process modeling language, called the Virtual Reality Process Modeling Language (VRPML). In doing so, this paper identifies the main components of the VRPML process support environments as well as summarizes the current implementation prototypes. Our experience highlights some lesson learned and offers insights into the design of next-generation PMLs and PSEEs.

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

### 1. Introduction

Engineering as a discipline relates to the creative application of mathematical and scientific principles to devise and implement solutions to problems in our everyday lives in an economic and timely fashion. To provide a quality solution, it is not usually sufficient to focus only on the final product. Often, it is also necessary to consider the *processes* involve in producing that product. For example, consider an assembly of a car. From the customer's perspective, it is the final product that matters (i.e. a quality car). From an engineering perspective, such quality could not be achieved if some of the processes (e.g. assembly lines) are faulty. Although additional rework can fix the problems caused by the faulty assembly lines, this tends to raise the overall costs because it deals only with symptoms of the problem. In contrast, going to the cause of the problem and improving the process (e.g. the faulty assembly lines) avoids the introduction of quality defects in the first place and leads to better results with lower costs. As this example illustrates, it is through the processes that engineers can observe and improve quality, control productions costs and possibly reduce the time to market their products.

Similar analogies can be applied in the case of software engineering. To produce quality software, it is also necessary to place emphasis on the processes by which the software is produced. In software engineering, these processes are usually called software processes. Software processes relate to the sequences of steps that must be performed by software engineers in order to pursue the goal of software engineering. In order to have an accurate representation and implementation of what the actual steps are, software processes may be modeled and enacted by a process modeling language (PML) and its process support system (*called the Process Centered Environments i.e.*

*PSEE*). Although there has been much fruitful research into PMLs and their corresponding PSEEs, their adoption by industry has not been widespread [6]. Furthermore, no single PML and PSEE have assumed dominance and accepted as the *de facto standard*.

For these reasons, research into PMLs and PSEEs are still necessary. In this paper, we discuss our experiences developing the support environments for the Virtual Reality Process Modeling Language (VRPML) [13-19]. We also discuss how the environment can be used to realize some of the main novel features of VRPML, that is, in terms of the integration with a virtual environment as well as the support for dynamic creation of tasks and allocation of resources [15, 16].

This paper is organized as follows. Section 2 gives an overview of VRPML. Section 3 summarises the main components of the VRPML support environments. Section 4 discusses the experience using the support environment. Section 5 discusses the prospect future work. Finally, section 6 presents the conclusion of the paper.

## 2. Overview of VRPML

Software processes are specified in VRPML as graphs, by interconnecting nodes from top to bottom using arcs that carry run-time control-flow signals. In VRPML, software processes are generically modeled. Resources (in terms of software engineers, artifacts and tools) can be dynamically assigned and customized for specific projects from a generic model.

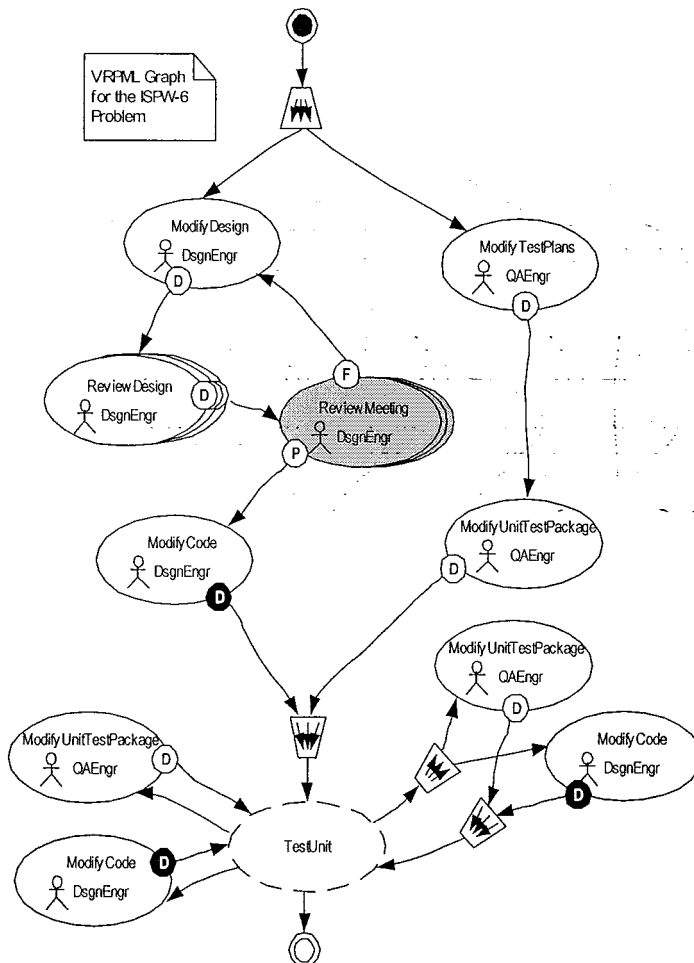


Figure 1. Excerpt from the VRPML Graph for the ISPW-6 Problem

As an illustration, Figure 1 presents an excerpt of the VRPML solution to a benchmark process, i.e. the ISPW-6 problem [8]. Briefly, the ISPW-6 problem involves a software requirement change request occurring either towards the end of the development phase or during the maintenance and enhancement phase of the software lifecycle. When a software change request is received, the project manager assigns and schedules specific tasks to a number of participating software engineers. These tasks includes: Modify Design; Review Design; Modify Code; Modify Test Plans; Modify Unit Test Package; and Test Unit. Some tasks may be executed in parallel, while others have to be executed in a sequential manner. In each task, there are defined roles, tools, source files, task ordering constraints, and pre-conditions and post-conditions which must be respected by the software engineers to complete the task.

Similar to JIL [10] and Little JIL [12], software processes in VRPML are described using process step abstractions, which represent the most atomic representation of a software process (i.e. the actual activity that software engineers are expected to perform). These activities are represented as nodes, called activity nodes (shown as small ovals with stick figures).

As depicted in Figure 1, VRPML supports many different kinds of activity nodes. They include: *general-purpose activity nodes* (shown as individual small ovals with stick figures); *multi-instance activity nodes* (shown as overlapping small ovals with stick figures); and *meeting activity node* (shown as small and shaded overlapping ovals with stick figures). Both multi-instance activity nodes and meeting activity nodes have associated depths, indicating the actual number of engineers involved (and also the number of identical activities in the case of multi-instance activity).

The firing of activity nodes is controlled by the arrival of a control flow signal. In VRPML, an initial control flow signal is always be generated from a *start node* (a white circle enclosing a small black circle). A *stop node* (a white circle enclosing another white circle) does not generate any control flow signals. Control flow signals may also be generated at the completion of a node, often from special completion events called *transitions* (shown as small white circles with a capital letter, attached to an activity node) or *decomposable transitions* (small black circles with a capital letter). Decomposable transitions enable automation scripts or sub-graphs to be specified (and executed if selected) as post-conditions before allowing transition to generate a control flow signal. The sub-graph associated with the decomposable transition representing Done (labeled D) for the activity node called Modify Code is given in Figure 2.

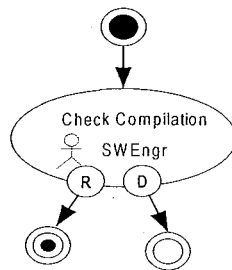


Figure 2. Sub-graph for Decomposable Transition labeled D in Modify Code

When Check Compilation fails, the assigned software engineer can select the transition R (for re-do). As a result, a control-flow signal will be generated to re-enact its parent node (i.e. Modify Code) through a *re-enabled node* (shown as two white circles enclosing black circle). Otherwise, if the compilation is successful, the assigned engineer can select the transition D (for Done). In this case, the control-flow signal will be generated and propagated back to the main graph to enable the subsequent connected node.

In VRPML, activity nodes can also be enacted in parallel using combinations of language elements called *merger* and *replicator* nodes (shown as trapezoidal boxes with arrows inside). To improve readability, a set of VRPML nodes can be grouped together and replaced by a *macro node* (shown as dotted line ovals), with the macro expansion appearing on a separate graph. For example, referring to Figure 1, Test Unit is a macro node. The macro expansion of Test Unit is given in Figure 3.



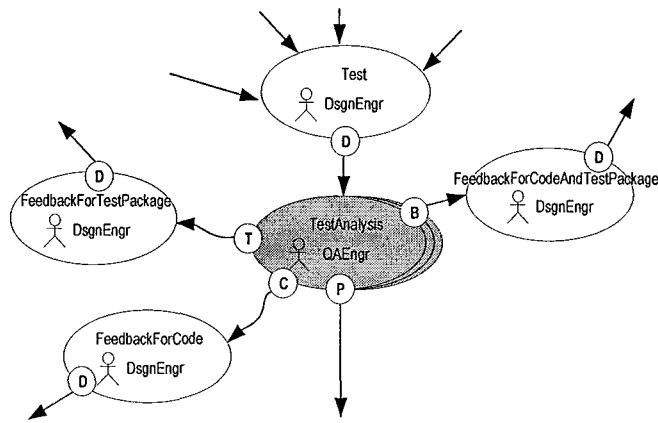


Figure 3. Macro Expansion for Test Unit in Figure 1

For every activity node, VRPML provides a separate *workspace*, the concept borrowed from ADELE-TEMPO [1], APEL [3] and MERLIN [7]. Figure 4 depicts the sample workspace for the activity node called Review Meeting in Figure 1. A workspace typically gives a *work context* of an activity as it hosts resources needed for enacting the activity: transitions, artifacts (shown as overlapping two overlapping documents with arrows for depicting access rights), communication tools (shown as a microphone, and an envelope), and any task descriptions (shown as a question mark). Effectively, when an activity is undertaken, the workspace is mapped into a virtual room, transitions into buttons, and artifacts, communication tools (i.e. for synchronous and asynchronous forms of communications) and task description into objects which can be manipulated by software engineers to complete the particular task at hand. This mapping is based on Doppke's task-centered mapping described in [4].

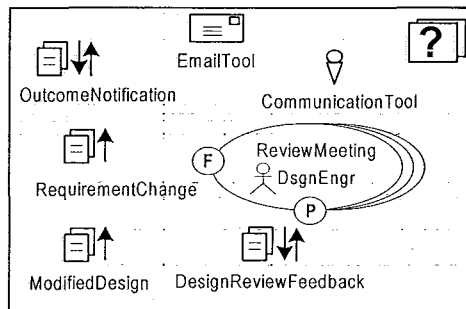


Figure 4. Sample Workspace for Activity Node Review Meeting from Figure 1

As part of its enactment model, VRPML relies on its resource exception handling mechanism. In VRPML, resources include roles assignment, artifacts and tools (including communication tools) in a workspace as well as the depths of multi-instance activity nodes and meeting activity nodes. Depending on the needs of a particular software development project, these resources can either be allocated during graph instantiation or dynamically during graph enactment. Upon the arrival of the control-flow signal, an activity node will be enabled. Here, the VRPML interpreter attempts to acquire resources that the activity node needs. If resources are successfully acquired, the VRPML interpreter then instantiates the activity corresponding to that activity node. If for any reason VRPML fails to acquire the resources, enactment will be blocked until such resources are made available (e.g. an engineer has not been assigned to the activity). In this way, the VRPML's resource exception handling mechanism is similar to blocking primitives (e.g. *in*, *read*) in Linda [5]. Once enactment is blocked, the VRPML interpreter automatically produces an activity for the administrator (e.g. process engineer) to rectify the resource exception or completely terminate the current activity. If that activity is terminated, the administrator may optionally terminate the overall enactment of the particular VRPML graph in question or manually re-enact connecting nodes by providing the necessary control-flow signals that they need to fire. If the resource exception is rectified, normal enactment of the

particular VRPML graph can be resumed resulting in the activity being assigned to the appropriate software engineer. When that engineer selects that particular activity, a workspace for that activity will appear as a virtual room with artifacts, transitions and communication tools as objects which software engineer can manipulate to complete the task. Finally, the activity completes when the software engineer selects one of the possible transitions (e.g. passed, failed, done, or aborted).

### 3. The VRPML Support Environment

In order to implement VRPML, a number of components for the support environment can be identified. These components and their interactions are shown in Figure 5.

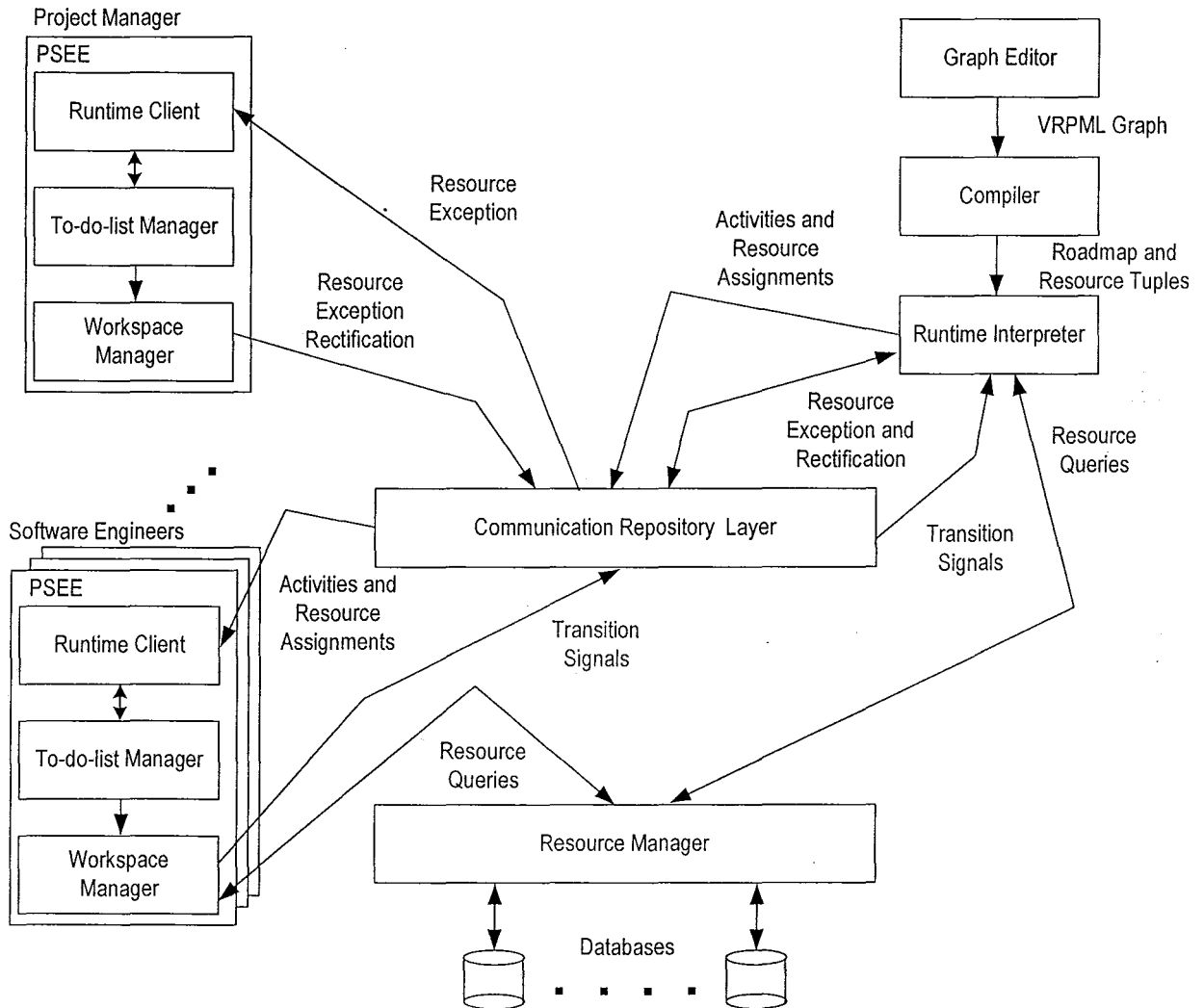


Figure 5. The VRPML Support Environment

Referring to Figure 5, the main components of the complete VRPML support environment are:

- Graph Editor – allows the VRPML graphs to be specified.
- Compiler – compiles the VRPML graphs into an immediate format for enactment.
- Runtime Interpreter – interprets the compiled VRPML graph.

- Runtime Client – retrieve activities and resource assignments from the communication repository layer.
- To-do-list Manager – manages the activities assigned to a particular software engineer.
- Workspace Manager – manages activity workspace in a virtual environment, manages activity transition, and forward queries to the resource manager.
- Communication Repository Layer – allows communication between the runtime interpreter, runtime client, and workspace manager.
- Resource Manager – queries the databases for artifacts.

The description of each of the components of the environment will be discussed next.

## Graph Editor

Much like a programmer's editor in a textual programming language, the graph editor would allow VRPML graphs to be drawn and changed. It would also allow browsing through graphs, and would support examination of every level of the graph (for instance, by opening further graph-editor windows onto workspaces and macros) in order to assist awareness issues (i.e. in terms of the readability of the VRPML graph). Although having a dedicated graph editor for VRPML would be vital for a complete system, it warrants no further discussion because the technology of graph editors is essentially already well-established. Consequently, the graph editor for VRPML has not been developed, and VRPML graphs were drawn manually.

## Compiler

A compiler would perform syntax checking and translate a VRPML graph into an intermediate format known to the runtime interpreter. In this research work, the compiler for VRPML has not been developed and the compilation of the VRPML graphs was performed manually. However, in order to ensure that a compiler could be written, research into the information needed for enactment was felt to be necessary. In particular, an important consideration for compiling VRPML is the information stored in the intermediate format. Clearly, the topology of the VRPML graph in terms of the ordering and sequencing of activities together with their resource assignments (if any) needs to be preserved.

One solution shown in Figure 6 is to produce a *roadmap* of how activities are interconnected and to generate the resource assignments in all the workspaces separately as *resource tuples*.

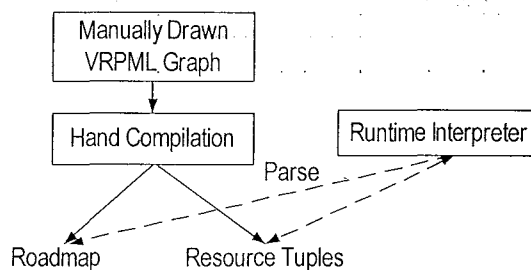


Figure 6. Compilation of the VRPML Graph

A format for the roadmap and the resource tuples has been identified (described below) and used to facilitate the hand-compilation of the VRPML graph, and hence permit enactment. To illustrate this technique, Figure 7 shows an example graph to be compiled.

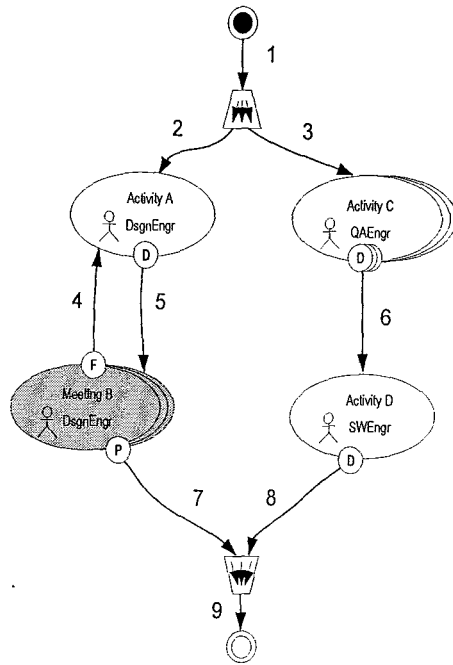


Figure 7. Simple VRPML Graph

The roadmap that is generated for the above VRPML graph is as follows (where the number shown on each arc is the internally generated control-flow signal id which is allowed to flow through the arc):

- 1, *Master 2] Master] 3]*)
- 2, *Administrator] Activity A]*)
- 3, *Administrator] Activity C]*)
- 4, *Administrator] Activity A]*)
- 5, *Administrator] Meeting B]*)
- 6, *Administrator] Activity D]*)
- 7, 8, *Master] 9]*)
- 9, *Master] Terminate]*)

A number of items in the roadmap need clarification and several terms in the roadmap need to be defined. “Master” refers to the runtime interpreter itself whilst “Administrator” refers to the role in charge of rectifying resource exceptions (e.g. when resource assignments are not specified or not available). The characters ,, ] and ) merely serve as separators which are used by the runtime interpreter to parse the enabling control-flow signal id (shown as a unique number for clarity), the defined activities and the target activity assignment (e.g. master or administrator).

Resource tuples must be generated for each activity defined in the graph. To illustrate the contents of a resource tuple, assume that the workspace for activity A shown in Figure 8 is defined below:

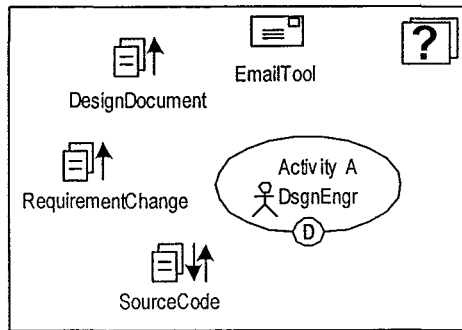


Figure 8. Example Workspace

The resource tuple for activity A is generated as follows with keywords (shown in bold) used for clarity and human readability.

*ActivityName* = Activity A, 2,  
*ActivityType* = General Purpose,  
*Role* = DsgnEngr  
*AssignedEngineer* = Unspecified,  
*Artifact* = Design Document, Path/Url for Modified Design, Read, Path/Url for tool,  
*Artifact* = Requirement Change, Path/Url for Req. Change, Read, Path/Url for tool,  
*Artifact* = Source Code, Path/Url for Source Code, Read/Write, Path/Url for tool,  
*Tool* = Email Program, Email, Path/Url for tool,  
*Transition* = D, Transition Done, Non-Decomposable, 5,  
*Descriptions* = Put the description of the activity here.

A resource tuple carries runtime information about the workspace consisting of: activity name and type; resource assignments including access rights for artifacts; tool assignments; and the defined transitions as well as the id of each control flow that will be generated if a particular transition is selected. One important aspect to observe in order to generate the resource tuple is that the id of each control-flow signal to be generated must be consistent with that defined in the roadmap. For example, transition Done must generate the control-signal id 5 in order to enable activity B.

Using the roadmap and resource tuples, enactment can easily be achieved. In fact, by adopting the Linda tuple space as the communication repository layer, enactment can be achieved in a distributed environment.

### Runtime Interpreter

Having considered the graph editor and the compiler, the next component is the runtime interpreter. Much of the functionality has already been implied in the earlier discussion. The full list of the runtime interpreter's functions is as follows:

- parse, maintain, and interpret the runtime information held in the roadmap and the resource tuples
- check for the arrival of control-flow signals in the communication repository layer, and decide when activities are able to fire
- interact with the resource manager to check for resource availability before enabling activities and return exceptions accordingly
- detect the termination of enactment and shut down gracefully

### Runtime Client

To support enactment in a distributed environment, there is the need to implement a runtime client which works on behalf of the to-do-list manager (described below) in order to retrieve the activities and their resource allocations

from the communication repository layer according to a software engineer's assignments. There are two types of runtime client which can be considered. In the first type, the runtime client automatically retrieves activities and their resource allocations as soon as they are assigned. In this case, there is a need for a dedicated channel which maintains a connection between the runtime client and the communication repository layer at all times. In the second type, the runtime client only retrieves activities and their resource allocations when there is an explicit request from the software engineer. As far as this research work is concerned, both types are equally useful. However, the second type has been chosen because it simplifies the implementation in the sense that there is no need to setup a dedicated communication channel between the runtime client and the communication repository layer.

### To-do-list Manager

In order to manage the assigned activities, there is also a need for a to-do-list manager. The full list of the to-do-list manager's functions is as follows:

- create a to-do-list for a particular software engineer
- interact with the runtime client upon request to retrieve the assigned activities from the communication repository layer
- provide an internal to-do-list queue to store the assigned activities received from the runtime client
- manage the graphical user interface (GUI) to allow a software engineer to select, browse, or undertake activities from the to-do-list queue as well as retrieve the assigned activities from the communication repository layer
- forward an activity and its resource allocations to the workspace manager (described below) when the activity is selected to be undertaken
- provide an interface to quit the to-do-list

As an illustration, Figure 9 shows a sample snapshot of the to-do-list GUI for a software engineer name Kamal where the current activity in the to-do-list queue is Review Design. Five buttons are also shown: Retrieve the activity; Perform the activity; Quit the to-do-list; Set; and Send. These buttons implement the functions described in the bulleted list above with the exception of the Send button. The Send button will be described below when the discussion on the workspace manager has been developed. It must be stressed that implementation efforts have concentrated on functionality rather than the aesthetics of the interface.

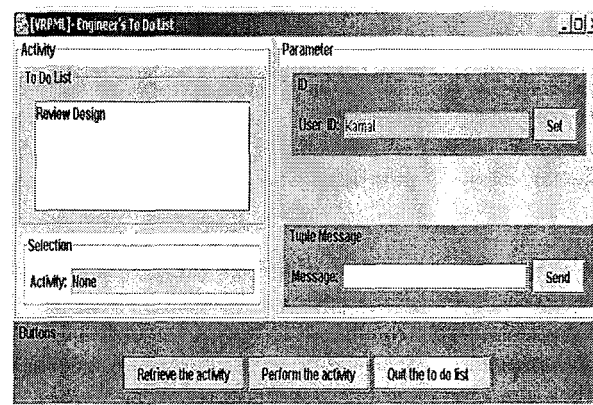


Figure 9. The To-Do-List Graphical User Interface

### Workspace Manager

The workspace manager generates and maintains each activity's workspace according to its work context, that is, in terms of the artifacts and tools required to complete that activity. Maintaining work context is important to give the software engineers a sense of awareness about the activity that they are currently undertaking.

At a first glance, generating and maintaining each activity's workspace according to its work context seems like a difficult task. However, a closer look reveals that this can easily be achieved by the workspace manager parsing the runtime information stored in the resource tuple. Using this runtime information, each activity's workspace can be generated when it is enabled.

As far as the implementation is concerned, the chosen approach to generate the workspace for each activity in a virtual environment was through the use of the Virtual Reality Modelling Language (VRML), a language for specifying three dimensional scenes with rich sets of object primitives and events [11]. With VRML, the workspace can be translated to a VRML scene, and the actual translation may be facilitated by a freely available CyberVRML97 library package [9] integrated as part of the workspace manager itself. In this work, efforts have been directed towards providing functionality and ignoring the aesthetics of the virtual environment.

Figure 10 below depicts the possible translation of the workspace and representation in a virtual environment for an activity called Modify Design. Utilizing Doppke's task-centered mapping [4], the activity's workspace maps into a virtual room with artifacts and tools corresponding to objects in that virtual room. Artifacts are represented as cylinders, and their access rights are distinguished by colours. Task descriptions are represented as help boxes. Transitions are represented as spheres.

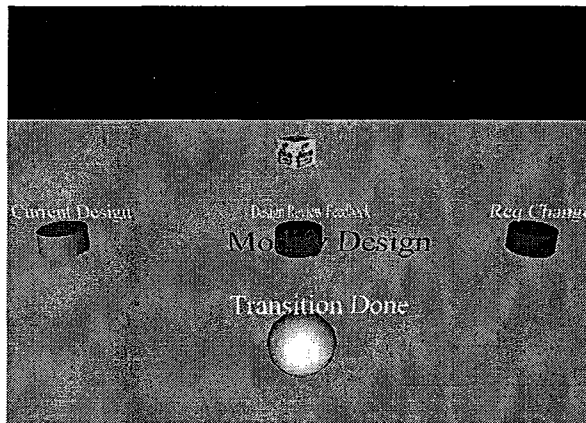


Figure 10. Sample Workspace in a Virtual Environment

Following a request from the software engineer, the workspace manager needs to interact with the resource manager in order to query the databases for artifacts and tools in the workspace to perform the activity. Because VRML supports event handling, it would be possible for the software engineer to access the objects (e.g. artifacts, tools, and transitions) in a scene, although this has not been investigated further as the objective here is to demonstrate that it is feasible to support integration with a virtual environment at the PML enactment level.

However, the fact that in the prototype the representation of transitions in the virtual environment are not active means that a mechanism is needed to simulate transitions and hence indicate the completion (or cancellation) of an activity. Thus, a Send button and a message box are provided in the to-do-list GUI (see Figure 10) to achieve the sending of a control flow signal to the communication repository layer by the workspace manager.

### Communication Repository Layer

The main function of the communication repository layer is to act as an intermediate mailbox for keeping the assigned activities and their resource tuples as well as the control-flow signals. There are three main components which interact with the communication repository layer: the runtime client to allow query of activity assignments and their resource tuples; the runtime interpreter to allow assignment of activities and their resources allocations to be made; and the workspace manager to allow the control-flow signals generated from transitions to be sent.

As far as implementation is concerned, the distributed shared memory model based on the Linda tuple space seems to be a suitable choice for the communication repository layer. The main reason for choosing the Linda tuple space stemmed from the fact that the VRPML enactment model is based on Linda, the base language from which the Linda tuple space is derived. In addition, Linda provides several pre-defined primitives which facilitate pattern matching of tuples in the tuple space and they can be used to simplify the implementation. While there are many Linda implementations available, Jada [2], the Linda implementation based on Java, has been chosen for this research work. Jada permits the user to setup a client-server based Linda tuple space that uses Java Remote Method Invocation. It is this tuple space that facilitates enactment in a distributed environment.

### Resource Manager

The last component that needs to be considered for supporting enactment is the resource manager. The main function of the resource manager is to handle the queries received from the runtime interpreter and the workspace manager. As discussed earlier, the queries received from the interpreter mainly involve checking for resource allocation and availability before allowing that activity to be assigned. The queries received from the workspace manager mainly involve requests to manipulate existing artifacts and tools or to create new artifacts. In addition to handling queries, the resource manager also enforces the required access rights involving artifacts, and supports changes and updates of the shared artifacts by multiple software engineers.

In terms of implementation, the resource manager can be straightforwardly realised by a database server with the capability of accessing more than one database at a time in a distributed environment. Because the technology to access the database server is well-established, it warrants no further discussions.

### 4. Discussion

A number of implementation issues relating to VRPML have been discussed in this paper. In particular, the main components of the VRPML support environment have been identified. As far as implementation is concerned, a working prototype has been developed (refer to [17]). Figure 11 summarized the overall class diagrams for the VRPML support environments based on the Unified Modeling Language notation.

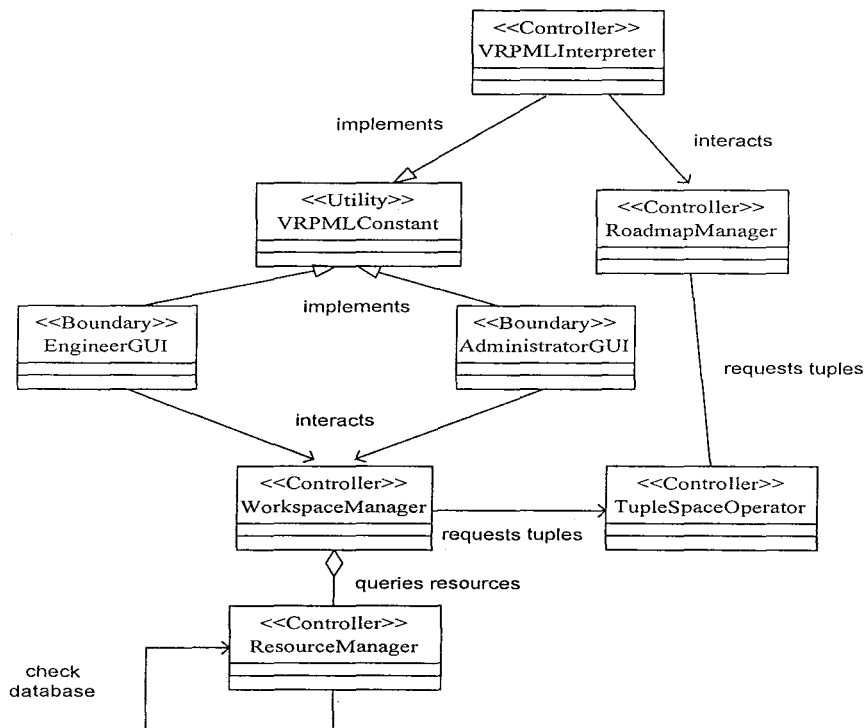


Figure 11. The UML Class Diagram for the VRPML Support Environment



Here, the VRPML interpreter class plays the role of interacting with all the roadmap generated by the compilation process. In order to do so, VRPML interpreter class needs to interact with the roadmap manager class. The roadmap manager class sequences the execution of the VRPML model based on the roadmap description. The exchange of information on the roadmap execution (i.e. tuples within a distributed environment) is done through the tuple space operator class which is responsible for managing the tuple operations on the communication layer. As their name suggest, the engineer GUI class and the administrator GUI class provides interface to the users. The construction of a virtual environment space for both the engineer's GUI and the administrator's GUI are done dynamically by the workspace manager class based on the workspace description of the activity assigned to them. Finally, the resource manager class is responsible for accessing databases for a particular task assigned to the engineers by the workspace manager.

As part of the evaluation process on the VRPML support environment discussed above (i.e. in terms of whether or not the environment is suitable for implementing the novel features of VRPML), an experiment has been successfully conducted which demonstrated enactment in a distributed environment by utilising the ISPW-6 problem. The complete discussion on the experiment is, however, beyond the scope of this paper. Interested readers are referred to our earlier work described in [16] and [17].

Given that the ISPW-6 problem has been formulated by experts in the field of software engineering, it should contain different types of (subtle) process issues seen in the real world. Thus, the fact that the support environment was able to assist and facilitate enactment is a positive indication of its applicability.

While the conversion from the VRPML graph to an intermediate format known to the interpreter is done manually in the current prototype, the translation process is done as a compiler would have. No expert knowledge, that is making use of information not available to the compiler, is applied during the translation process. Therefore, a compiler support for VRPML is implementable. In fact, as far as an automated compiler is concerned, it is currently under development along with the complete support environments.

## 5. Future Work

As the current implementation of VRPML is still in a prototype form, an obvious starting point for future work would be to complete the implementation. For example, implementing a automated compiler and a complete workspace manager would be a useful endeavour.

A number of other research avenues could also be investigated. The fact that VRPML and its support environments supports integration with a virtual environment opens up many possibilities for using visualization to provide multiple views of the same process model from different perspectives, and hence potentially improving process understanding. Instead of using a straightforward mapping of workspaces, that is, a workspace, artifacts, tools and task descriptions map one-to-one to a virtual room and objects, other meaningful visualisations could also be explored by defining or using other types of mapping. For example, artifact-centred mapping as defined by Doppke *et al* [4] could be used where artifacts are represented as virtual rooms and their dependency relationships are expressed as part of the arrangement of the rooms. If sub-products of the artifacts are defined then they are represented as separate rooms connected to the parent product room either by exits or by containment, whilst tools and task description can be defined as objects inside the room. Clearly, by manipulating the types of mapping used, multiple views of the same process can be achieved. In turn, such views may enhance the support for awareness.

Another possible area of research is to find other ways of addressing awareness in VRPML, for example, supporting user awareness by representing software engineers as avatars in the workspaces during enactment or using live video. Using avatars or live video can perhaps improve the sense of realism and further encourage informal communication as engineers can "see" each other. This is especially useful if the workspaces involve more than one person, and the software engineering teams are physically distributed.

Although useful by giving focus on a particular activity, it is believed that workspaces defined by VRPML (see Figure 11) give insufficient working context particularly about the overall activities, that is, in terms of how the pieces fit together into the whole picture. In the current VRPML implementation, questions such as what the

previous task was, what the next task is, and what needs to be done to move along cannot be easily answered. Therefore, it would be useful to find ways for VRPML to also give context of the overall activity, for example, by giving the software engineers access to the enacted VRPML graph in the forms of animated flow of control during enactment.

Finally, because a software development project often involves hard deadlines, another possibility for further work is to investigate the inclusion of "timing" criteria as part of the VRML notation as well its runtime environment. Perhaps, the timing criteria could be exploited as part of the workspace definitions, raising an "alarm" when an activity is due to be completed. As a result, the software engineers can be reminded about the deadlines of the activities that they are undertaking.

## 6. Conclusion

Because of the potential benefits in terms of being able to provide automation, guidance and enforcement of software engineering practices and policies, through modeling and enactment (i.e. execution), a PML and its PSEEs could form an important feature of future software engineering environments. Moving toward a goal of a practicable PML and PSEE, this paper has highlighted the main components of the VRPML support environments. The fact that the VRPML support environments complement the VRPML novel features (e.g. in terms of supporting distributed enactment within a virtual environment) gives a positive indication of its applicability. As such, we believe that our work offers valuable insights into the design of next-generation PMLs and PSEEs.

## Acknowledgement

The work undertaken in this research is partially funded by the USM Short Term Grants – "The Design and Implementation of the VRPML Runtime Environment".

## References

- [1] N. Belkhatir, J. Estublier, and W. Melo. ADELE-TEMPO: An Environment to Support Process Modelling and Enaction. in Nuseibeh, B. ed. *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, 187-222.
- [2] P. Ciancarini, and D. Rossi. Jada: A Coordination Toolkit for Java - Technical Report UBLCS-96-15, Department of Computer Science, University of Bologna, Italy, 1997.
- [3] S. Dami, J. Estublier, and M. Amiour. "APEL: A Graphical Yet Executable Formalism for Process Modeling". *Automated Software Engineering*, 5 (1), 1998, 61-96.
- [4] J.C. Doppke, D. Heimbigner, and A.L. Wolf. "Software Process Modeling and Execution within Virtual Environments". *ACM Transactions on Software Engineering and Methodology*, 7 (1), 1998, 1-40.
- [5] D. Gelernter. "Generative Communication in Linda". *ACM Transactions on Programming Languages and Systems*, 7 (1), 1985, 80-112.
- [6] M.J. Jaccheri, R. Conradi, B.H. Drynes, Software Process Technology and Software Organisations. In: Conradi, R. (ed.): *Proc. of 7th European Workshop on Software Process Technology (EWSPT 2000)*, Kaprun, Austria, Springer-Verlag, 96-108
- [7] 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, 103-129.
- [8] M.I. Kellner, P.H. Feiler, A. Finkelstein, T. Katayama, L.J. Osterweil, M.H. Penedo, and H.D. Rombach. "Software Process Modeling Example Problem". In *Proc. of the 6th Intl. Software Process Workshop*, Hakodate, Hokkaido, Japan, October 1990, IEEE CS Press.

- [9] S. Konno, *CyberVRML97 - Virtual Reality Modelling Language Development Library*, 2002.
- [10] S. Sutton Jr., and L.J. Osterweil. The Design of a Next-Generation Process Language. in *Proc. of the Joint 6th European Software Engineering Conference and the 5th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, (1997), Lecture Notes in Computer Science Volume 1301, Springer, 142-158.
- [11] The VRML Consortium. *VRML97 International Standard Specification (ISO/IEC 14772-1:1997)*
- [12] A. Wise. "Little JIL 1.0 Language Report - Technical Report 98-24", Dept. of Computer Science, Univ. of Massachusetts at Amherst, April 1998.
- [13] 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, 435-437.
- [14] K.Z. Zamli. "Process Modeling Languages: A Literature Review". *Malaysian Journal of Computer Science* 14, (2), December 2001.
- [15] 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 (PROFES02)*, Lecture Notes in Computer Science Volume 2559, Rovaniemi, Finland, 2002, Springer, 49-62.
- [16] 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, December 2003, IEEE CS Press, 243-252.
- [17] K.Z. Zamli. "Supporting Software Processes for Distributed Software Engineering Teams", School of Computing Science, Univ. of Newcastle upon Tyne, PhD Thesis, October 2003.
- [18] K.Z. Zamli and N.A. Mat Isa, "A Survey and Analysis of Process Modeling Languages". *Malaysian Journal of Computer Science*, 17.(2), December 2004, 68-89.
- [19] K.Z. Zamli, and N.A. Mat Isa, "The Computational Model for a Flow-based Visual Languages". In *Proc. of the AIDIS International Conference in Applied Computing 2005*, Algarve, Portugal, 217-224.

**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, University Technology-Malaysia in 2000, and PhD in Software Engineering from the University of Newcastle upon Tyne, UK in 2003. He is currently attached to 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 University Science Malaysia in 2000 and PhD in Image Processing and Neural Networks from the same university in 2003. He is currently attached to the School of Electrical and Electronics Engineering, USM Engineering Campus in Transkrian. He specializes in the area of image processing, neural networks for medical applications, and software engineering.

**Norazlina Khamis** obtained her BSc in Information Technology from the University of Malaya in 1999 and her MSc in Real Time Software Engineering from CASE, University of Technology Malaysia in 2001. She is currently attached to the Department of Software Engineering, Faculty of Computer Science & Information Technology, University of Malaya. Her academic activities include teaching various undergraduate computer science courses ranging from software engineering, database, operating system, software quality and software requirements engineering.

# Implementing Executable Graph Based Visual Language in a Distributed Environment

<sup>1</sup>Kamal Zubairi Zamli, <sup>2</sup>Nor Ashidi Mat Isa and <sup>3</sup>Norazlina Khamis

<sup>1,2</sup>Software Engineering Research Group,  
School of Electrical and Electronics,  
Universiti Sains Malaysia Engineering Campus,  
14300 Nibong Tebal, Pulau Pinang, Malaysia  
Tel: +604-5937788 ext 6079, Fax: +604-5941023  
Email: {eekamal,ashidi}@eng.usm.my

<sup>3</sup>Faculty of Computer Science and Information  
Technology,  
Universiti Malaya,  
50603 Lembah Pantai, Kuala Lumpur, Malaysia  
Tel: +603-76976402, Fax: +603-79676339  
Email: azlina@um.edu.my

**Abstract**—One of the common difficulties in a graph based visual language is to develop its executable semantics and achieved its execution in a distributed environment. In order to address some of these issues, this paper outlines the general control flow semantics of a graph based visual language. In doing so, this paper also discusses a sound technique implementing such semantics permitting execution in a distributed environment. An implementation is sketched for a domain specific graph based visual language, called VRPML.

## I. INTRODUCTION

Visual programming languages have been around for quite some time now. The basic idea behind a visual programming language is that computer graphics (e.g. graphs consisting of icons, nodes, and arcs) are used instead of a textual representation. In fact, the central argument for a visual programming language is based on an observation that picture is better than text (i.e. a picture is worth a thousand words [15]).

While a visual programming language may not be able to provide a silver bullet to solve every problem related to engineering a software system, a carefully chosen level of abstractions (e.g. by working at the same level of abstraction as the problem domain) coupled with easy to understand notations may help alleviate the low-level complexities offered by the textual counterpart.

There have been many visual languages developed in the domains of computer science. In this paper, we discuss a common subset of visual language, that is, the graph based visual language. One of the common difficulties in a graph based visual language is to develop its executable semantics and achieved its execution (i.e. enactment) in a distributed environment. In order to address some of these issues, this paper outlines the general control flow semantics of a graph based visual language. In doing so, this paper also discusses a sound technique implementing such semantics permitting execution in a distributed environment. An implementation is sketched for a domain specific graph based visual language, called VRPML [10].

This paper is organized as follows. Section 2 gives an overview of graph based visual languages. Section 3 introduces the syntax and semantics of VRPML.

Section 4 identifies the possible runtime components supporting execution (or enactment) of the VRPML graph. Section 5 outlines our prototype implementation. Finally, section 6 presents the conclusions of the paper.

## II. GRAPH BASED VISUAL LANGUAGE

Graph based visual language has been around since the early days of computers. Whether we realize or not, flow chart can be seen as a form of graph based visual languages.

Typically, graphs consist of nodes, arcs and sub-graphs. Nodes represent function or actions, arcs carry data or control-flow signals, and sub-graphs provide abstraction and modularization. Operations in graphs follow a *firing rule* which defines the conditions under which execution of node occurs.

In the control-flow based model, a visual program consists of nodes connected by arcs carrying control-flow signals. Arcs depict the control-flow dependencies amongst connected nodes. The firing rule is based solely on the availability of the control-flow signals on the node's input arcs – that is, data availability does not play any part at all.

Conceptually, in the control-flow based model, every program can be thought of as having an instruction counter and a globally addressable memory which holds programs and data objects whose contents are updated by program instructions during execution [1].

As far as a visual language associated with the control-flow model is concerned, for simplicity, it may be viewed as supporting executable flowcharts. In the data-flow based model, a visual program consists of nodes connected by arcs carrying data. Arcs depict data dependencies amongst nodes. The firing rule is based on the availability of data on the node's input arcs, and may be *data-driven* or *demand-driven*.

With a data-driven firing rule, an arc is used as a supply route to transmit data from the source node to the destination node. A destination node is executed as soon as data is available on all input arcs. With a

demand-driven firing rule, an arc is used as a demand route to request data from the source node. A source node is executed only if there is a demand for its result. For either firing rule, arcs are conduits for data. In turn, data on an arc is consumed by the executing node to perform its computation (although some variations of the data-flow based model also allow an arc to retain data).

According to Agerwala and Arvind [2], the data-flow based model can be distinguished from the control-flow based model in that it has neither a globally addressable memory nor a single instruction counter. As the data-flow based model possesses no global memory, the only data available to a node for its operation is that from its inputs. In addition, because of the lack of any shared data amongst nodes, there can be no *side effects* (one node interfering with other node's data, potentially causing unexpected results).

As the data-flow firing rule depends solely on the availability of data, nodes whose data is available can potentially be executed in parallel. The sequencing of the execution of nodes, for example in terms of the assignment of runtime processes to processors, is determined solely at runtime by the runtime system. Thus, a data-flow based model supports parallelism naturally.

Apart from the control-flow or the data-flow based models, one less popular paradigm is the computational model based on both models. Here, there are two kinds of arcs with different semantics: the data-flow and the control-flow arc. The firing rule for this paradigm can be complex because it is based on the combination of both the data-flow and the control-flow signals. Furthermore, while the problem of arcs crossing each other and resulting in a cluttered view is inherent in a flow based visual language based on directed graphs, the fact that two arcs are used here means that the crossover problem can be even greater. Generally, if there are too many arc crossovers, the overall program understanding may be compromised.

Although the data flow and a combined model can be used to build the semantics, the focus of this paper is on the control flow semantics of the graph based visual language. We foresee that the control flow semantics seems to be popular as it fits well into our understanding of a general purpose programming language [12].

### III. INTRODUCING VRPML

VRPML is a domain specific executable graph based visual language for supporting the modeling and enacting of software processes [10-13]. The main novel features of VRPML are:

- It considers virtual environments as a fundamental constituent, manipulatable as part of the construction of the process model (i.e. via features in the language) as well as being part of the runtime environment.

- It supports dynamic allocation of resources through its enactment model.
- It supports dynamic allocation of resources through its enactment model

In VRPML, software processes are generically modeled. Resources (in terms of software engineers, artifacts and tools) can be dynamically assigned and customized for specific projects from a generic model.

Referring to Figure 1, software processes are specified in VRPML as graphs, by interconnecting nodes from top to bottom using arcs that carry runtime control-flow signals. Similar to JIL [8] and Little JIL [9], a software process activity in VRPML are described using process step abstractions, which represent the most atomic representation of a software process (i.e. the actual activity that software engineers are expected to perform). These activities are represented as nodes, called activity nodes (shown as small ovals with stick figures).

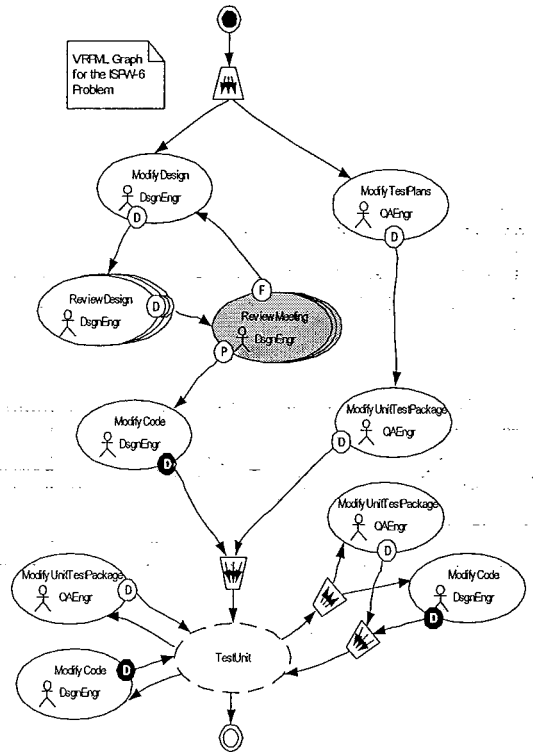


Fig. 1. The VRPML Graph

As depicted in Figure 1, VRPML supports many different kinds of activity nodes. They include: *general-purpose activity nodes* (shown as individual small ovals with stick figures); *multi-instance activity nodes* (shown as overlapping small ovals with stick figures); and *meeting activity node* (shown as small and shaded overlapping ovals with stick figures). Both multi-instance activity nodes and meeting activity nodes have associated depths, indicating the actual number of engineers involved (and also the number of identical activities in the case of multi-instance activity).

The firing of activity nodes is controlled by the arrival of a control flow signal. In VRPML, an initial control flow signal is always generated from a *start node* (a white circle enclosing a small black circle). A *stop node* (a white circle enclosing another white circle) does not generate any control flow signals. Control flow signals may also be generated at the completion of a node, often from special completion events called *transitions* (shown as small white circles with a capital letter, attached to an activity node) or *decomposable transitions* (small black circles with a capital letter). Decomposable transitions enable automation scripts or sub-graphs to be specified (and executed if selected) as post-conditions before allowing transition to generate a control flow signal. The sub-graph associated with the decomposable transition representing Done (labeled D) for the activity node called Modify Code is given in Figure 2.

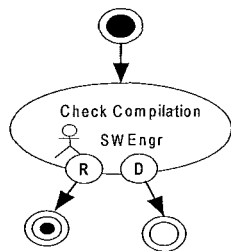


Fig. 2. Sub-graph for Decomposable Transition labeled D in Modify Code

When Check Compilation fails, the assigned software engineer can select the transition R (for redo). As a result, a control-flow signal will be generated to re-enact its parent node (i.e. Modify Code) through a *re-enabled node* (shown as two white circles enclosing black circle). Otherwise, if the compilation is successful, the assigned engineer can select the transition D (for Done). In this case, the control-flow signal will be generated and propagated back to the main graph to enable the subsequent connected node.

In VRPML, activity nodes can also be enacted in parallel using combinations of language elements called *merger* and *replicator* nodes (shown as trapezoidal boxes with arrows inside). To improve readability, a set of VRPML nodes can be grouped together and replaced by a *macro node* (shown as dotted line ovals), with the macro expansion appearing on a separate graph. For example, referring to Figure 1, Test Unit is a macro node. The macro expansion of Test Unit is given in Figure 3.

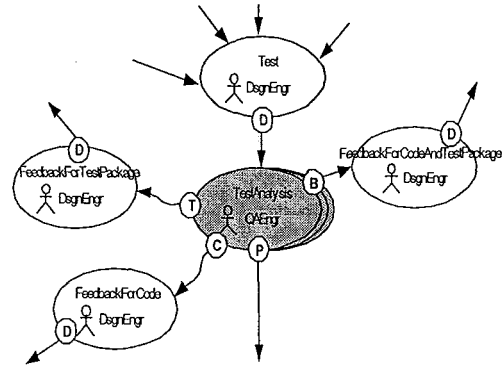


Fig. 3. Macro Expansion for Test Unit in Figure 1

For every activity node, VRPML provides a separate *workspace*, the concept borrowed from ADELE-TEMPO [3], APEL [5] and MERLIN [7]. Figure 4 depicts the sample workspace for the activity node called Review Meeting in Figure 1. A workspace typically gives a *work context* of an activity as it hosts resources needed for enacting the activity: transitions, artifacts (shown as overlapping two overlapping documents with arrows for depicting access rights), communication tools (shown as a microphone, and an envelope), and any task descriptions (shown as a question mark).

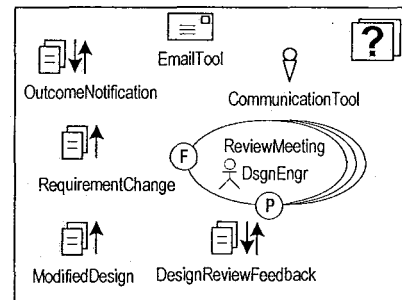


Fig. 4. Sample Workspace for Activity Node Review Meeting from Figure 1

#### IV. VRPML RUNTIME COMPONENTS

Having discussed the control flow semantics of VRPML, this section outlines the possible implementation components in their context. It must be noted that the execution of the VRPML graph occurs in a distributed environment, that is, enabling of an activity means assigning that activity to a particular person playing certain role and may not be co-located with other persons.

Briefly, the main implementation components are as follows (see Figure 5 in the next page):

- Graph Editor – allows the VRPML graphs to be specified.
- Compiler – compiles the VRPML graphs into an immediate format for enactment.
- Server/Client Interpreter daemon – interprets the compiled VRPML graph.

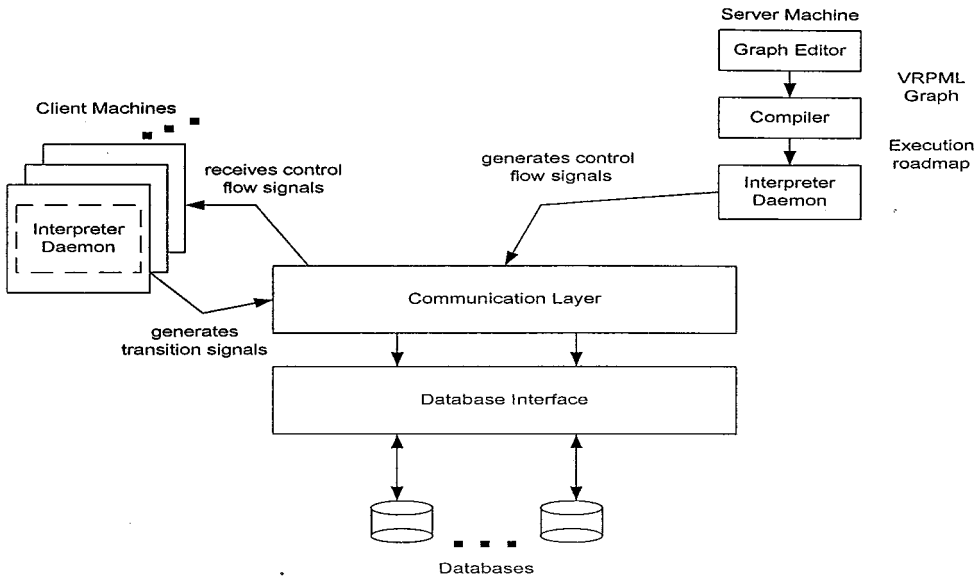


Fig. 5. VRPML Runtime Component

- Communication Layer – act as mailboxes to coordinate the control flow signal for enabling execution in a distributed environment

The most important components which warrant further discussion are: the compiler; the client/server interpreter daemon; and the communication layer. A compiler performs syntax checking and translates a VRPML graph into an intermediate format known to the runtime interpreter. An important consideration for compiling VRPML is the information stored in the intermediate format. Clearly, the topology of the VRPML graph in terms of the ordering and sequencing of activities together with their resource assignments (if any) needs to be preserved.

A format for the roadmap and workspaces has been identified and used to facilitate the compilation of the VRPML graph, and hence permit execution. To illustrate this technique, Figure 6 shows an example graph to be compiled.

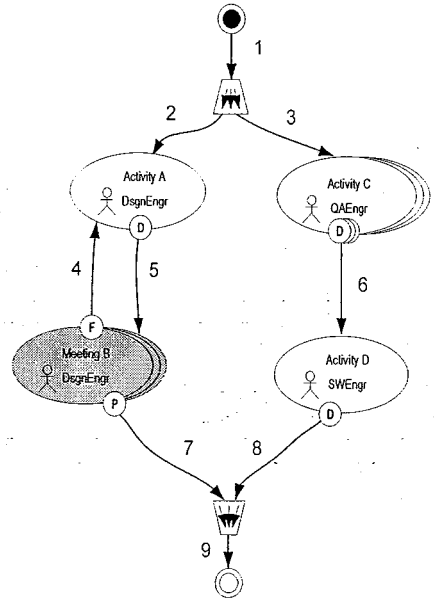


Fig. 6. Simple VRPML Graph

The roadmap that is generated for the above VRPML graph is as follows (where the number shown on each arc is the internally generated control-flow signal id which is allowed to flow through the arc):

- 1, Master 2] Master 3])
- 2, Administrator] Activity A])
- 3, Administrator] Activity C])
- 4, Administrator] Activity A])
- 5, Administrator] Meeting B])
- 6, Administrator] Activity D])
- 7, 8, Master] 9])
- 9, Master] Terminate])

A number of items in the roadmap need clarification and several terms in the roadmap need to be defined. "Master" refers to the server interpreter daemon whilst "Administrator" refers to the role in charge of performing activity assignments. The characters ,, ] and ) merely serve as separators which are used by the runtime interpreter to parse the enabling control-flow signal id (shown as a unique number for clarity), the defined activities and the target activity assignment (e.g. master or administrator).

Workspaces must be generated for each activity defined in the graph. To illustrate the contents of a workspace, assume that the workspace definition for activity A shown in Figure 6 is defined below:

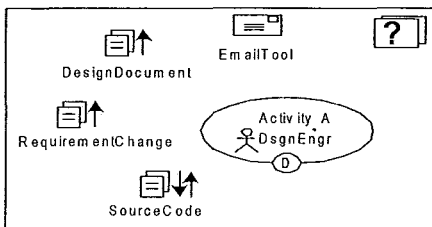


Fig. 7. Example Workspace

The workspace for activity A is generated as follows with keywords (shown in bold) used for clarity and human readability.

**ActivityName** = Activity A, 2,  
**ActivityType** = General Purpose,  
**Role** = DsgnEngr  
**AssignedEngineer** = Unspecified,  
**Artifact** = Design Document, Path/Url for Modified Design, Read, Path/Url for tool,  
**Artifact** = Requirement Change, Path/Url for Req. Change, Read, Path/Url for tool,  
**Artifact** = Source Code, Path/Url for Source Code, Read/Write, Path/Url for tool,  
**Tool** = Email Program, Email, Path/Url for tool,  
**Transition** = D, Transition Done, Non-Decomposable, 5,  
**Descriptions** = Put the description of the activity here.

A workspace carries runtime information about the workspace consisting of: activity name and type; resource assignments including access rights for artifacts; tool assignments; and the defined transitions as well as the id of each control flow that will be generated if a particular transition is selected. One important aspect to observe in order to generate the workspace is that the id of each control-flow signal to be generated must be consistent with that defined in the roadmap. For example, transition Done must generate the control-signal id 5 in order to enable activity B.

Having considered the compiler, the next component is the client/server interpreter daemon. Much of the functionality has already been implied in

the earlier discussion. The full list of the client/server interpreter daemon's functions is as follows:

- parse, maintain, and interpret the runtime information held in the roadmap
- check for the arrival of control-flow signals in the communication layer, and decide when activities are able to fire
- detect the termination of enactment and shut down gracefully

Finally, the communication layer acts as an intermediate mailbox for keeping the assigned activities as well as the control-flow signals. There are three main components which interact with the communication layer: the client interpreter daemon to allow query of activity assignments; the server interpreter daemon to allow assignment of activities and their workspaces to be made as well as to allow control-flow signals generated from transitions (as transition signals) to be sent.

## V. IMPLEMENTATION PROTOTYPE

A proof-of-concept prototype implementation in Java has been built based on the components identified on section 4 (see [13] for details). The server interpreter daemon translates the VRPML roadmap in order to correctly assign tasks to software engineers (i.e. based on a given control flow). Given the task assignment, the client interpreter daemon puts the task in the engineer's to-do-list. When the engineer chooses the task, the client interpreter daemon acquires the necessary resources in order to allow engineer to perform the task.

As far as the communication layer is concerned, the distributed shared memory model based on the Linda tuple space [6] is a suitable choice for the communication repository layer. The main reason for choosing the Linda tuple space stemmed from the fact that Linda provides several pre-defined primitives which facilitate pattern matching of tuples in the tuple space and they can be used to simplify the implementation. While there are many Linda implementations available, Jada [4], the Linda implementation based on Java, has been chosen for this research work. Jada permits the user to setup a client-server based Linda tuple space that uses Java Remote Method Invocation. It is this tuple space that facilitates enactment in a distributed environment.

## VI. CONCLUSION

This paper has discussed how execution in a distributed environment can be achieved for a graph based visual language based on the control flow semantics. It is hopeful that lesson learned from this implementation is beneficial to other domain specific visual language, such as VORLON [14], particularly in achieving its execution within a distributed environment.



## ACKNOWLEDGMENT

The work undertaken in this research is partially funded by the USM Short Term Grants – “The Design and Implementation of the VRPML Runtime Environment”.

## REFERENCES

- [1] W.B. Ackerman, 1982. Data Flow Languages. In IEEE Computer, pp 15-23.
- [2] T. Agerwala and Arvind, 1982. Data Flow Languages. In IEEE Computer, pp 10-13.
- [3] N. Belkhatir, J. Estublier, and W. Melo. ADELE-TEMPO: An Environment to Support Process Modelling and Enaction. in Nuseibeh, B. ed. *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, 187-222.
- [4] P. Ciancarini, and D. Rossi. Jada: A Coordination Toolkit for Java - Technical Report UBLCS-96-15, Department of Computer Science, University of Bologna, Italy, 1997.
- [5] S. Dami, J. Estublier, and M. Amieur. “APEL: A Graphical Yet Executable Formalism for Process Modeling”. *Automated Software Engineering*, 5 (1), 1998, 61-96.
- [6] D. Gelernter. “Generative Communication in Linda”. *ACM Transactions on Programming Languages and Systems*, 7 (1), 1985, pp. 80-112.
- [7] 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, 103-129.
- [8] S. Sutton Jr., and L.J. Osterweil, “The Design of a Next-Generation Process Language”. In Proc. of the Joint 6th European Software Engineering Conference and the 5th ACM SIGSOFT Symposium on the Foundation of Software Engineering, (1997), Lecture Notes in Computer Science Volume 1301, Springer, 142-158.
- [9] A. Wise. “Little JIL 1.0 Language Report - Technical Report 98-24”, Dept. of Computer Science, Univ. of Massachusetts at Amherst, April 1998.
- [10] 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 (PROFES02), Lecture Notes in Computer Science Volume 2559, Rovaniemi, Finland, 2002, Springer, pp. 49-62.
- [11] 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, December 2003, IEEE CS Press, pp. 243-252.
- [12] K.Z. Zamli and N.A. Mat Isa, “The Computational Model for a Flow Based Visual Language”. In Proc. of AIDIS Intl. Conf. on Applied Computing 2005, Algarve, Portugal, pp. 217-224.
- [13] K.Z. Zamli, N.A. Mat Isa, and N. Khamis, “The Design and Implementation of the VRPML Support Environments”. *Malaysia Journal of Computer Science* 18 (1), pp. 57-69.
- [14] J. Webber, and P.A. Lee, “Visual, Object Oriented Development of Parallel Applications”. *Journal of Visual Languages & Computing* 12 (2), pp. 145-161.
- [15] K.N. Whitley, 1997. “Visual Programming Languages and the Empirical Evidence For and Against”. In *Journal of Visual Language and Computing*, Vol. 8, No. 1, pp. 109-142.

# Modeling and Enacting Software Processes: The Why and How Questions

Kamal Zuhairi Zamli, Nor Ashidi Mat Isa

**Abstract** - This paper describes the why, and how questions relating to the need to support the modeling and enacting of software processes for supporting the activities of software engineers. In doing so, a number of related works are presented in order to highlight the current advancement in the area.

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

## I. INTRODUCTION

Engineering as a discipline relates to the creative application of mathematical and scientific principles to devise and implement solutions to problems in our everyday lives in an economic and timely fashion. To provide a quality solution, it is not usually sufficient to focus only on the final product. Often, it is also necessary to consider the *processes* involve in producing that product [15]. For example, consider an assembly of a car. From the customer's perspective, it is the final product that matters (i.e. a quality car). From an engineering perspective, such quality could not be achieved if some of the processes (e.g. assembly lines) are faulty. Although additional rework can fix the problems caused by the faulty assembly lines, this tends to raise the overall costs because it deals only with symptoms of the problem. In contrast, going to the cause of the problem and improving the process (e.g. the faulty assembly lines) avoids the introduction of quality defects in the first place and leads to better results with lower costs. As this example illustrates, it is through the processes that engineers can observe and improve quality, control productions costs and possibly reduce the time to market their products.

Similar analogies can be applied in the case of software engineering. To produce quality software, it is also necessary to place emphasis on the processes by which the software is produced. In software engineering, these processes are usually called software processes.

This paper describes the why, and how questions relating to the need to support the modeling and enacting of software processes for supporting the

activities of software engineers. In doing so, a number of related works are presented in order to highlight the current advancement in the area.

The organization of this paper is as follows. Section 2 gives an overview of software processes as well as provides justification for supporting the software processes. Section 3 presents a survey of the related work. Finally, section 4 presents some conclusion.

## II. THE WHY QUESTIONS

Software processes can be defined as sequences of steps that must be carried out by humans (e.g. software engineers), to pursue the goals of software engineering. There are many ways that can be used to define a particular software process. Perhaps the simplest way to define the software process is to use a natural language such as English. For example, one may describe a software process for the unit testing stage of software development as the following steps:

- Step 1: Check out the affected modules from the configuration management system.
- Step 2: Obtain the test cases from the project manager.
- Step 3: Perform the testing for all the test cases.
- Step 4: Produce the test report for the project manager.

Using a natural language for defining a software process is straightforward but exhibits a number of difficulties. In general, the description of software process using a natural language is often imprecise, ambiguous, inconsistent and open to user interpretation. Typically, such characteristics may lead to discrepancies in the software process undertaken by software engineers – for example, what is performed may not be what is required in the description of the software process. To eliminate such discrepancies, there is a need for a more precise way of specifying a software process.

In software engineering, such a need is translated into the use of modeling languages to specify a software process. In particular, software processes can be specified using a *process modeling language* (PML) and assisted by an environment called *Process*

Centered Software Engineering Environment (PSEE). Through the use of a PML, software processes can be described in a precise way in terms of what a process comprises and how it is structured and organized. This can be instrumental in eliminating inconsistencies in the process specification.

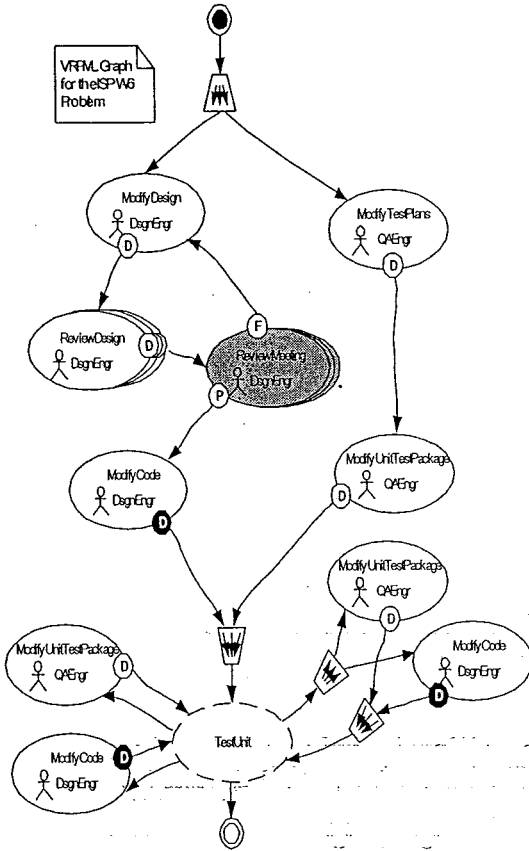


Figure 1. Excerpt from the VRPML Graph for the ISPW-6 Problem

As its name suggests, a PML is used to construct a form of model of the actual software development process. Such a model is often called *process model*, which is a representation of the actual software process excluding details which do not influence its relevant behavior. For example, Figure 1 depicts the process model for the common software process problem called the ISPW-6 problem [12] expressed using the PML developed as part of our research (the Virtual Reality Process Modeling Language – VRPML [19-25]).

Referring to Figure 1, the ISPW-6 problem involves a software requirement change request occurring either towards the end of the development phase or during the maintenance and enhancement phase of the software lifecycle. When a software change request is received,

the project manager assigns and schedules specific tasks to a number of participating software engineers. These tasks includes: Modify Design; Review Design; Modify Code; Modify Test Plans; Modify Unit Test Package; and Test Unit. Some tasks may be executed in parallel, while others have to be executed in a sequential manner. Not all tasks start at the same time (i.e. they require coordination). Although not completely shown in Figure 1, in each task, there are defined roles, tools, source files, task ordering constraints, and pre-conditions and post-conditions which must be respected by the software engineers to complete the task.

While the overall syntax and semantics of VRPML used in Figure 1 has not been fully explained (and is discussed in Section 3 of this paper as part of our survey), the above model of the ISPW-6 problem provides some insights into the modeling of software processes and their degree of complexities.

In addition to being able to support the modeling of software processes, PMLs may also allow execution of the process models to support the activities of software engineers. The execution of such processes is usually termed *process enactment*. Enactment is very useful feature of a PML for the following reasons [7]:

- It provides guidance through the steps to be taken. Such guidance is particularly useful for junior software engineers.
- It can enforce strict procedures and policies. Enforcement of strict procedures is sometimes important in cases such as developing critical systems where human lives depend on a piece of software. An example of such a system would be a car auto-cruise control system. In this case, the software development team in charge of developing such a system may require its defined steps to be followed precisely. For example, evolution of the software in such a system must be strictly controlled. *Ad hoc* changes must not be permitted because such changes may introduce bugs which may not be tested and accounted for. Such bugs could be dangerous especially if they affect the mechanism to control the speed of the car in auto-cruise.
- It permits the automation of tasks. In software engineering, there are many tasks which can benefit from automation. For example, although tasks such as compiling and linking source codes look simple, they can be painstakingly dull especially if the source codes are very large

and involving multiple modules. Such mundane tasks, if automated, can relieve software engineers from tedious routine work (and reduce potential human errors), and consequently, improve software engineer's productivity.

### III. THE HOW QUESTIONS

Based on the discussion given earlier, software process can also be viewed as a partially ordered set of activities that must undertaken by software engineers to manage, develop, maintain and evolve software systems. To allow better control of a particular software process, a model of that process (i.e. a process model) can be created using a PML making the process explicit and open to examination. Also, a process model can show deficiencies and inconsistencies in the current software process that would otherwise be obscured, hence making it easier to analyze the process and suggest improvements.

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 or interpreting system, 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.

In an attempt to find an ideal PML for the modeling and enacting of software processes, there has been much research into different language paradigms. Many of the language paradigms have been adapted following experiences from existing approaches in software engineering applied in the context of a software process.

There have already been a number of attempts to classify these different language paradigms. For example, Liu and Conradi [13] 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 utilizes 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 [14] proposes a similar classification of PMLs, consisting of the following categories:

- Graphical PMLs – PMLs which provide a graphical syntax.
- Net-oriented PMLs – PMLs which utilises nets such as Petri nets.
- Procedural PMLs – PMLs which adopts a procedural programming language.
- Object-oriented PMLs – PMLs which utilises some features from object-oriented languages (e.g. objects and inheritance).
- Rule-based PMLs – PMLs which exploits rules-based techniques mainly based on Prolog.
- Multi-paradigm PMLs – PMLs which utilizes 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 include object-orientation. This object-oriented categorization was appropriate at the time because many authors of PMLs were beginning to incorporate features from object-oriented languages as part of their PMLs.

Based on the classifications from Liu and Conradi and Lonchamp, Huff [9] 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 Huff includes the non-executable category of PMLs. Essentially, the outcome of Huff's non-executable categorization 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 [14]. 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. In fact, as will be seen later, some approaches use more than one PML in order to support the modeling and enacting of software processes.

In order to understand how the modeling and enacting of software processes are supported in the literature, it is necessary to survey existing PMLs. Due

to space constraint, the survey in this paper presents snapshots of the state of the arts on PMLs, that is, by mainly focusing on the more recent PMLs. A more comprehensive survey of PMLs can be found in the author's previous work in [20, 23].

## A. SLANG

SLANG [2], 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 criteria 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 sets of input and output arcs. In fact, an interface serves as a SLANG modularization 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 produces 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 (though 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.

## B. MERLIN

MERLIN [11] is a Prolog-like PML which has been developed by the University of Dortmund and STZ – Gesellschaft für Software-Technologie mbH. 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, a software engineer may interact with this diagram in a hypertext manner to perform their work.

## C. APPL/A

APPL/A [16] is a PML based on the ADA programming language. 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. Transactions-like statements control access to relations and may affect the enforcement of predicates.

## D. Dynamic Task Nets

Dynamic Task Nets [8] is the visual PML for the PSEE called Dynamite. Dynamic Task Nets described 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 updated 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 customized 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 customization 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.

## E. LATIN

Language to tolerate Inconsistencies (LATIN) [5] 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 parameters lists. Instantiation of a task type constitute 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 characterized 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 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. In such a situation, the PSEE runtime system automatically performs *pollution analysis* which gives some analysis of the deviations from the defined process model and the identification of some polluted data (mainly artifacts) caused by such deviations.

## F. JIL

JIL [17] is a PML derived from experiences in developing APPL/A. 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* specify 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 keyword 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. precondition 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.

## G. Little JIL

Little JIL [18] is a PML based on JIL. Little JIL is a visual PML, and 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 unit of work and whose structure represents the way in which this work will be coordinated. As an illustration, Figure 2 displays the Little JIL step notation.

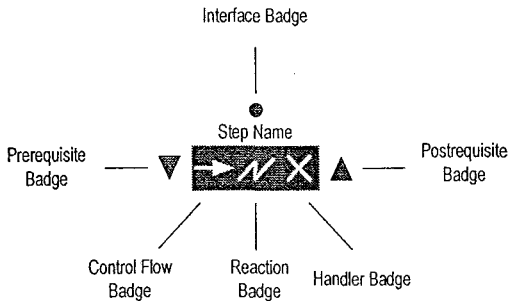


Figure 2. The Little JIL Step Notation

Referring to Figure 2, the Little JIL step notation has a number of components including:

- Step name section*: Every step must be given a name.
- 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.
- Pre-requisite and post-requisite badge section*: Here, pre-conditions and post-conditions for the step are specified.
- 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.

- 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.
- 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.

## H. CSPL

CSPL [3] 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:

- 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.
- 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.
- Work assignment statements are CSPL statements which allow activities, tools and roles to be assigned to one or more software engineers.
- Communication related statements allow synchronization and ordering of tasks with other tasks, similar to the ADA95 rendezvous.
- 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 scripts.

## I. APEL

APEL [6] 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 state Object Management 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 essentially consisting of self-defined goals, questions related to the process models achieving that goals, and metrics to quantify such questions.

## J. VRPML

VRPML is a flow-based visual PML. The main novel features of VRPML are that it considers the virtual environment as a fundamental constituent, manipulatable as part of the construction of the process model (i.e. via features in the language) as well as being part of the runtime environment, and supports dynamic allocation of resources through its enactment model [20-25].

In VRPML, software processes are generically modeled. Resources (in terms of software engineers, artifacts and tools) can be dynamically assigned and customized for specific projects from a generic model.

Referring to Figure 1 given earlier, software processes in VRPML are described using process step abstractions, which represent the most atomic representation of a software process (i.e. the actual activity that software engineers are expected to perform). These activities are represented as nodes, called activity nodes (shown as small ovals with stick figures).

As depicted in Figure 1, VRPML supports many different kinds of activity nodes. They include: *general-purpose activity nodes* (shown as individual small ovals with stick figures); *multi-instance activity nodes* (shown as overlapping small ovals with stick figures); and *meeting activity node* (shown as small and shaded overlapping ovals with stick figures). Both multi-instance activity nodes and meeting activity nodes have associated depths, indicating the actual number of engineers involved (and also the number of identical activities in the case of multi-instance activity).

The firing of activity nodes is controlled by the arrival of a control flow signal. In VRPML, an initial control flow signal is always generated from a *start node* (a white circle enclosing a small black circle). A *stop node* (a white circle enclosing another white circle) does not generate any control flow signals. Control flow signals may also be generated at the completion of a node, often from special completion events called *transitions* (shown as small white circles with a capital letter, attached to an activity node) or *decomposable transitions* (small black circles with a capital letter). Decomposable transitions enable automation scripts or sub-graphs to be specified (and executed if selected) as post-conditions before allowing transition to generate a control flow signal.

Activity nodes can also be enacted in parallel using combinations of language elements called *merger* and *replicator* nodes (shown as trapezoidal boxes with

arrows inside in Figure 1). To improve readability, a set of VRPML nodes can be grouped together and replaced by a *macro node* (shown as dotted line ovals), with the macro expansion appearing on a separate graph (e.g. Test Unit in Figure 1).

For every activity node, VRPML provides a separate *workspace*. Figure 3 depicts the sample workspace for the activity node called Review Meeting in Figure 1. A workspace typically gives a *work context* of an activity as it hosts resources needed for enacting the activity: transitions, artifacts (shown as overlapping two overlapping documents with arrows for depicting access rights), communication tools (shown as a microphone, and an envelope), and any task descriptions (shown as a question mark). Effectively, when an activity is undertaken, the workspace is mapped into a virtual room, transitions into buttons, and artifacts, communication tools (i.e. for synchronous and asynchronous forms of communications) and task description into objects which can be manipulated by software engineers to complete the particular task at hand.

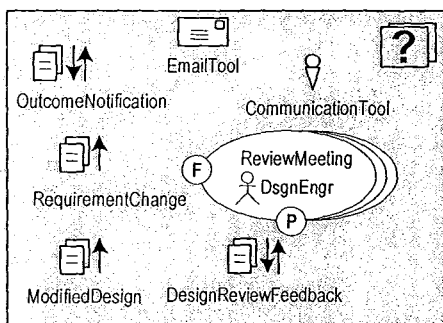


Figure 3. Sample Workspace for Activity Node Review Meeting from Figure 1

As part of its enactment model, VRPML relies on its resource exception handling mechanism. In VRPML, resources include roles assignment, artifacts and tools (including communication tools) in a workspace as well as the depths of multi-instance activity nodes and meeting activity nodes. Depending on the needs of a particular software development project, these resources can either be allocated during graph instantiation or dynamically during graph enactment.

#### IV. CONCLUSION

In conclusion, this paper answers the basic questions relating to the need to support the modeling and enacting of software processes as well as provides

a survey of PMLs which highlights the current advancement in the area.

As has been shown, because of the potential benefits in terms of being able to provide automation, guidance and enforcement of software engineering practices and policies through enactment, a PML could form an important feature of future software engineering environments. Nonetheless, despite the above promise, the adoption of PMLs in industry has still not been widespread [10]. Furthermore, no single existing PML has emerged dominant as the *de facto* standard for modeling and enacting software processes. For these reasons, it follows that research into PMLs is still necessary.

#### Acknowledgement

The work undertaken in this research is partially funded by the USM Short Term Grants – “The Design and Implementation of the VRPML Runtime Environment”.

#### REFERENCES

- [1] V. Ambriola, R. Conradi, and A. Fuggetta. Assessing Process-Centered Software Engineering Environments. *ACM Transactions on Software Engineering and Methodology*, 6 (3). 283-328.
- [2] 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*, Research Studies Press, Taunton, England, 1994, 223-247.
- [3] J.J. Chen. CSPL: An Ada95-Like, Unix-Based Process Environment. *IEEE Transactions on Software Engineering*, 23 (3). 171-184.
- [4] R. Conradi, and M.L. Jaccheri. Process Modelling Languages, in Derniame, J.C., Kaba, B.A. and Wastell, D. eds. *Software Process: Principles, Methodology and Technology*, Lecture Notes in Computer Science Volume 1500, Springer, Berlin-Heidelberg, 1999, 27-52.
- [5] G. Cugola, E.D. Nitto, C. Ghezzi, and M. Mantione. How to Deal with Deviations during Process Model Enactment. in *Proc. of the 17th Intl. Conf on Software Engineering*, Seattle, Washington, April 1995, IEEE Computer Society Press, 265-273.
- [6] S. Dami, J. Estublier, and M. Amieur. “APEL: A Graphical Yet Executable Formalism for Process

- Modeling". *Automated Software Engineering*, 5 (1), 1998, 61-96.
- [7] 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*, Lecture Notes in Computer Science Volume 1500, Springer, Berlin-Heidelberg, 1999, 1-13.
- [8] 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, 331-341.
- [9] K.E. Huff. Software Process Modeling. in A. Fuggetta, and A. Wolf, eds. *Trends in Software Process*, John Wiley & Sons, 1996, 1-24.
- [10] M.L. Jaccheri, R. Conradi, and B.H Drynes. Software Process Technology and Software Organisations. in *Proc.s of the 7th European Workshop on Software Process (EWSPT 2000)*, Kaprun, Austria, February 2000, Lecture Notes in Computer Science Volume 1780, Springer, 96-108.
- [11] 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, 103-129.
- [12] M.I. Kellner, P.H. Feiler, A. Finkelstein, T. Katayama, L.J. Osterweil, M.H. Penedo, and H.D. Rombach. "Software Process Modeling Example Problem". In *Proc. of the 6th Intl. Software Process Workshop*, Hakodate, Hokkaido, Japan, October 1990. IEEE CS Press.
- [13] C. Liu, and R. Conradi. Process Modeling Paradigms: An Evaluation. in *Proc. of the 1st European Workshop on Software Process Modeling*, Milano, Italy, May 1991, Italian National Association for Computer Science, 39-52.
- [14] J. Lonchamp. An Assessment Exercise. in A. Finkelstein, J. Kramer, and B. Nuseibeh, eds.. *Software Process Modelling and Technology*, Research Studies Press Ltd., Taunton, Somerset, U.K., 1994, 335-356.
- [15] L.J. Osterweil. Software Processes are software too, revisited. In *Proc. of the 19th IEEE Intl. Conf. on Software Engineering*, Boston, USA, 1997, IEEE CS Press, 540-548.
- [16] S. Sutton Jr., D. Heimbigner, and L.J. Osterweil. APPL/A: A Language for Software Process Programming. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4 (3), 221-286.
- [17] S. Sutton Jr., and L.J. Osterweil. The Design of a Next-Generation Process Language. in *Proc. of the Joint 6th European Software Engineering Conference and the 5th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, (1997), Lecture Notes in Computer Science Volume 1301, Springer, 142-158.
- [18] A. Wise. "Little JIL 1.0 Language Report - Technical Report 98-24", Dept. of Computer Science, Univ. of Massachusetts at Amherst, April 1998.
- [19] 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*, 2001, IEEE CS Press, 435-437.
- [20] K.Z. Zamli. "Process Modeling Languages: A Literature Review". *Malaysia Journal of Computer Science* 14, 2 (December 2001)
- [21] 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 (PROFES02)*, Lecture Notes in Computer Science Volume 2559, 2002, Springer, 49-62.
- [22] 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*, December 2003, IEEE CS Press, 243-252
- [23] K.Z. Zamli. "Supporting Software Processes for Distributed Software Engineering Teams", School of Computing Science, Univ. of Newcastle upon Tyne, PhD Thesis, 2003.
- [24] K.Z. Zamli and N.M. Mat Isa, "A Computational Model for a flow-based visual PML", submitted for publication.
- [25] K.Z. Zamli and N.M. Mat Isa, "The Applicability of VRPML for Supporting Distributed Software Engineering Teams", submitted for publication.

## 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,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, analyze, 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 utilizes 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 categorization 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 executable. 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 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 appeared. 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 characterize the requirements of PMLs. However, one aspect perceived to be lacking in the previous classifications of PMLs is 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 organizational 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 utilizing visual syntax; enactment within a virtual environment; supporting user and process awareness; process visualization; virtual meetings; as well as reflecting that support in the features provided in a PML [32].

Capitalizing on these issues relating to the human dimension and building from the earlier characterizations and requirements of PMLs, Table 1.0 presents an alternative characterization 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 1.0: Taxonomy for PML

	<b>PML Characteristics</b>
<b>Modeling Support</b>	Sequential and parallel activities as well as their constraints
	Input and output artifacts
	Role representations
	External tools
	Abstraction and modularization
<b>Enactment Support</b>	Enactment in a distributed environment
	Dynamic allocation of resources
<b>Evolution Support</b>	Reflection
<b>Evaluation Support</b>	Collection of enactment data
<b>Human Dimension Support</b>	Visual notations
	User awareness
	Process awareness
	Process visualization
	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 certain set 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 modularization. Through modularization 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 modularization 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 allowed 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 are 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 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 organizational 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 visualization, 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 visualization.
- 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 criteria 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 sets of input and output arcs. In fact, an interface serves as a SLANG modularization 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 produces 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 (though 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, modularization 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 resource 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 customizable 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, modularization 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* and *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. *Assoc* 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, modularization 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, a software 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 specialized PROLOG facts. Parallel activities are supported by the PSEE runtime engine. However, it is not clear how the modularization 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 (utilizing 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 visualization.

### 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 customization 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 specialized 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 utilizing 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. Modularization 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 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 defines 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, modularization 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 centralized 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 subtle meaning than the one defined in 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 synchronization 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 rule 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, modularization 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 visualization, 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. Transactions-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 modularization 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 described 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 updated 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 customized 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 customization 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 customized 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, modularization 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 parameters lists. Instantiation of a task type constitute 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 characterized 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 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, modularization 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* specify 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 keyword 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 specialized 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 a pre- or post-conditions of a step. Tools can be invoked directly through the reactive control specification. Finally, modularization 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.

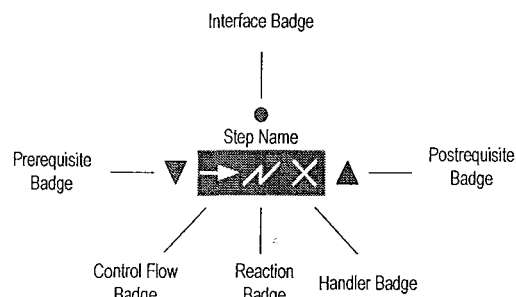


Figure 1. Little JIL Step Notation

A process model in Little JIL can be viewed as a tree of steps whose leaves represent the smallest specified unit of work and whose structure represents the way in which this work will be coordinated. As shown in Figure 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.

### 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 modularization 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 visualization.

### 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 synchronization 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 scripts.

### 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, modularization and abstraction are supported by utilizing 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 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 that 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 event 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, modularization and abstraction are supported as activities 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 visualization 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 is 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 customized 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 utilizing 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 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, modularization 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 summarizes the description of existing PMLs. Although only enactable PMLs are considered, some of the categorizations 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 visualization, 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 term 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		S	L	B	M	S	M	A	D	A	D	L	C	A	P
		L	I	M	&	E	R	A	E	P	P	L	J	L	E
		O	M	P	P	S	L	S	L	E	L	J	C	A	P
		P	I	A	P	P	L	P	E	L	L	J	S	P	E
		A	M	M	P	L	L	L	E	L	L	J	P	E	L
		T	E	P	M	L	L	L	E	L	L	J	P	E	L
		E	P	M	P	L	L	L	E	L	L	J	P	E	L
		L	I	M	&	E	R	A	D	A	D	L	C	A	P
		O	M	P	P	S	L	P	E	L	L	J	C	A	P
		P	I	A	P	L	L	P	E	L	L	J	S	P	E
		A	M	M	P	L	L	L	E	L	L	J	P	E	L
		T	E	P	M	L	L	L	E	L	L	J	P	E	L
		E	P	M	P	L	L	L	E	L	L	J	P	E	L
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 modularization	√	X	√		√	√	√	√	√	√	√	X	√	√
Enactment Support	Enactment in a distributed environment	√	√	√	√	√	√	√		√	√		√	√	√
	Dynamic allocation of resources		X		X		X	X	X	√	X	X	X	X	X
Evolution Support	Reflection	√	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
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
	Process awareness	X	X	X	√	X	X	√	X	X	X	X	X	√	X
	Process visualization	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

The third perceived deficiency in the surveyed PMLs is in term 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:

- V isual notations
- U ser awareness
- P rocess awareness

- P rocess visualization
- V irtual meetings
- D ynam ic allocation of resources
- C ollection of enactment data
- R eflection

To address the support the features above, 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 visualization 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.

## Acknowledgement

The work undertaken in this research is partially funded by the USM Short Term Grants – “The Design and Implementation of the VRPML Runtime Environment”.

## 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 Modelling 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 Intl. 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 Catalonia, 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 specializes in the area of image processing, neural networks for medical applications, and software engineering.

# Addressing Race Condition Problem in a Graph Based Visual Language

Kamal Z. Zamli and Nor Ashidi Mat Isa  
Pusat Pengajian Kejuruteraan Elektrik dan Elektronik  
Universiti Sains Malaysia Engineering Campus  
14300 Nibong Tebal, Penang  
Email: {eekamal, ashidi}@eng.usm.my

## Abstract

*Graph based visual languages are typically based on nodes, directed arcs and sub-graphs. Nodes represent function or actions, arcs carry data or control-flow signals, and sub-graphs provide abstraction and modularization. Operations in graphs follow a firing rule which defines the conditions under which execution of node occurs. In the control flow based model, the firing rule is based solely on the availability of the control-flow signals on the node's input arcs. In the data flow based model, the firing rule is based on the availability of the required data on the node's input arcs.*

*This paper discusses our partial evaluation of a domain specific graph based visual language, called VRPML utilizing the well-known ISPW-6 benchmark problem as a case study. The focus of the evaluation is on the firing rule. Due to its control flow based firing rule, we observe that VRPML suffers from race condition problem, that is, two or more control-flow signals can inadvertently and erroneously compete to enable a particular activity node. This paper outlines our proposal to address the problem and enhance the language syntax and semantics.*

## 1. Introduction

Visual programming languages have been around for quite some time now. The basic idea behind a visual programming language is that computer graphics (e.g. graphs consisting of icons, nodes, and arcs) are used instead of a textual representation. In fact, the central argument for a visual programming language is based on an observation that picture is better than text (i.e. a picture is worth a thousand words [5]).

While a visual programming language may not be able to provide a silver bullet to solve every problem

related to engineering a software system, a carefully chosen level of abstractions (e.g. by working at the same level of abstraction as the problem domain) coupled with easy to understand notations may help alleviate the low-level complexities offered by the textual counterpart. Motivated by the abovementioned arguments, much research has been undertaken on visual languages over the last 20 years. Of interest to us is the graph based visual language [9][10][11].

Graph based visual languages are typically based on nodes, directed arcs and sub-graphs [1][2]. Nodes represent function or actions, arcs carry data or control-flow signals, and sub-graphs provide abstraction and modularization. Operations in graphs follow a firing rule which defines the conditions under which execution of node occurs. In the control flow based model, the firing rule is based solely on the availability of the control-flow signals on the node's input arcs. In the data flow based model, the firing rule is based on the availability of the required data on the node's input arcs.

This paper discusses our partial evaluation of a domain specific graph based visual language, called VRPML [7][10], utilizing the well-known ISPW-6 benchmark problem [4] as a case study. The focus of the evaluation is on the VRPML's firing rule. Due to its control flow based firing rule, we observe that VRPML suffers from the race condition problem, that is, two or more control-flow signals can inadvertently and erroneously compete to enable a particular activity node. This paper outlines our proposal to address the problem and enhance the language syntax and semantics.

This paper is organized as follows. Section 2 gives an overview on the VRPML syntax and semantics. Section 3 discusses the ISPW 6 problem and provides brief descriptions of the corresponding VRPML solution. Section 4 discusses the lessons learned. Finally, section 5 presents the conclusions of the paper.

## 2. Overview of VRPML

VRPML [7][10] is a domain specific graph based visual language adopting the control-flow model firing rule. VRPML is used to model and execute a software process, that is, a sequence of steps that must be followed by software engineers to pursue the goals of software engineering.

Software processes are specified in VRPML as graphs, by interconnecting nodes from top to bottom using arcs that carry run-time control-flow signals. As an illustration, Figure 1 presents the main VRPML solution to a benchmark process, i.e. the ISPW-6 problem [4]. This solution will be discussed further in the next section.

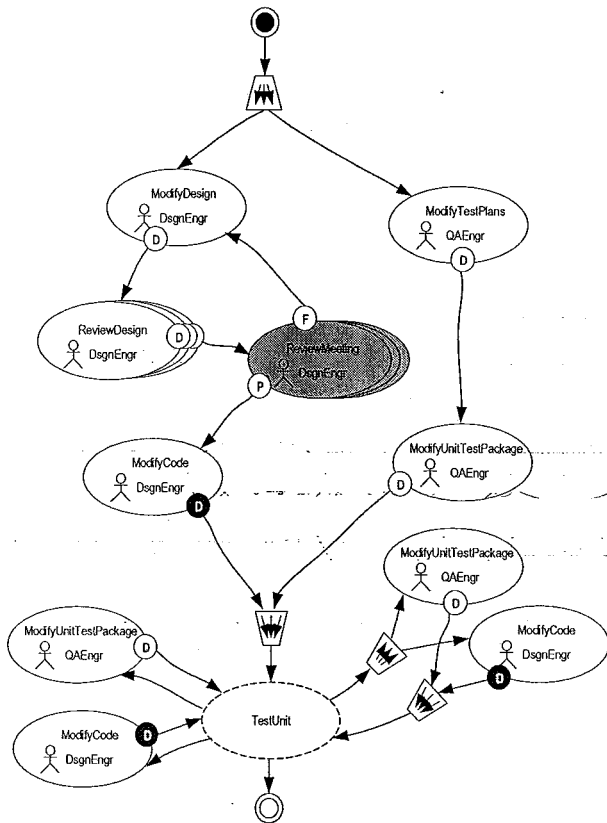


Figure 1. Main VRPML Graph for the ISPW-6 Problem

Similar to Little JIL [6], software processes in VRPML are described using process step abstractions, which represent the most atomic representation of a software process (i.e. the actual activity that software engineers are expected to perform). These activities are represented as nodes, called activity nodes (shown as small ovals with stick figures).

As depicted in Figure 1, VRPML supports many different kinds of activity nodes. They include:

*general-purpose activity nodes* (shown as individual small ovals with stick figures); *multi-instance activity nodes* (shown as overlapping small ovals with stick figures); and *meeting activity node* (shown as small and shaded overlapping ovals with stick figures). Both multi-instance activity nodes and meeting activity nodes have associated depths, indicating the actual number of engineers involved (and also the number of identical activities in the case of multi-instance activity). Also, a set of VRPML nodes can be grouped together using a *macro node* (shown as dotted line ovals) to improve the graph readability.

The firing of activity nodes is controlled by the arrival of a necessary control-flow signal. Control-flow signals may be generated from *transitions* (shown as small white circles with a capital letter attached to an activity node) or *decomposable transitions* (shown as small black circles with a capital letter attached to an activity node). However, the initial control-flow signal must always be generated from a *start node* (shown as a white circle enclosing a black circle). A *stop node* (shown as a white circle enclosing another white circle) does not generate any control-flow signal.

In VRPML, activity nodes can also be enacted in parallel using combinations of language elements called *merger* and *replicator* nodes (shown as trapezoidal boxes with arrows inside).

For every activity node, VRPML provides a separate *workspace*. Figure 2 depicts the sample workspace for the activity node called Review Meeting in Figure 1. A workspace typically gives a *work context* of an activity as it hosts resources needed for enacting the activity: transitions, artifacts (shown as overlapping two overlapping documents with arrows for depicting access rights), communication tools (shown as a microphone, and an envelope), and any task descriptions (shown as a question mark). Effectively, when an activity is undertaken, the workspace is mapped into a virtual room, transitions into buttons, and artifacts, communication tools and task description into objects which can be manipulated by software engineers to complete the particular task at hand. This mapping is based on Doppke's task-centered mapping described in [3].

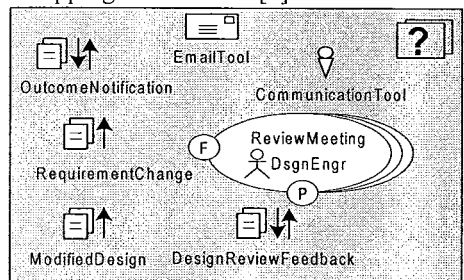


Figure 2. Sample Workspace for Activity Node Review Meeting from Figure 1

### 3. Solution to the ISPW-6 Problem

The ISPW-6 problem [4] concerns with a software change request occurring at the end of the development project. A number of activities are defined including: Modify Design; Review Design; Modify Code; Modify Test Plans; Modify Unit Test Package; and Test Unit. Some activities may be executed in parallel, while others have to be executed in a sequential manner. In each activity, there are also defined roles, tools, source files, and pre-conditions and post-conditions which must be respected by the software engineers to complete the task. Figure 3 and Table 1 summarizes the flow of activities, responsibility assignments, inputs and outputs involved in the ISPW-6 problem.

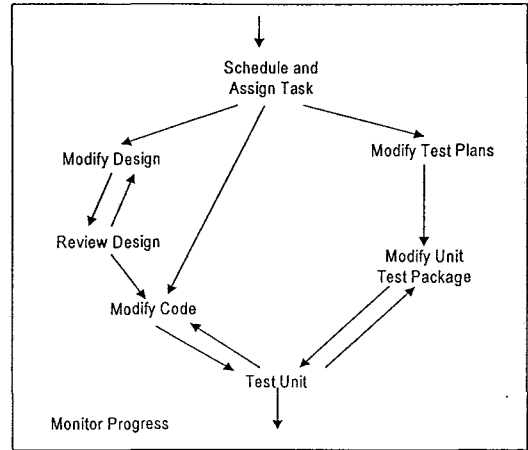


Figure 3. Flow of Tasks in the ISPW-6 Problem

Table 1: ISPW-6 Activities, Inputs and Outputs

Schedule And Assign Tasks	Responsibility: Project Manager
	Inputs: Requirement Change, Authorisation, Project Plans
	Outputs: Updated Project Plans, Notification of Task Assignments and Schedule Dates, Requirement Change
	Constraints: - Begins as soon as authorisation is given - Ends when outputs have been provided
Modify Design	Responsibility: Design Engineer
	Inputs: Requirement Change, Current Design, Design Review Feedback
	Outputs: Modified Design
	Constraints: - Can begin as soon as the task been assigned - Subsequent iteration can begin if design is not approved by the Review Design - Ends when outputs have been provided
Review Design	Responsibility: Design Review Team
	Inputs: Requirement Change, Modified Design
	Outputs: Design Review Feedback, Approved Modified Design, Outcome Notification
	Constraints: - Begins on schedule provided the modified design is available at the time - Ends when outputs have been provided
Modify Code	Responsibility: Design Engineer
	Inputs: Requirement Change, Modified Design, Current Source Code, Feedback Regarding Code
	Outputs: Modified Source Code, Object Code
	Constraints: - Can begin as soon as the task has been assigned even if Modify Design has not begun (discretion) - Ends when clean compilations are achieved, outputs have been provided and design is approved - Subsequent iteration can begin if required when test unit has completed
Modify Test Plans	Responsibility: QA Engineer
	Inputs: Requirement Change, Current Test Plans
	Outputs: Modified Test Plans
	Constraints: - Can begin as soon as the task has been assigned - Ends when outputs have been provided
Modify Unit Test Package	Responsibility: QA Engineer
	Inputs: Requirement Change, Modified Test Plans, Current Unit Test Package, Modified Design, Source Code, Feedback Regarding Test Package
	Outputs: Modified Unit Test Package
	Constraints: - Can begin as soon as Modify Test Plans has completed - Subsequent iteration can begin if required as Test Unit has completed - Ends when outputs have been provided
Test Unit	Responsibility: Design Engineer, QA Engineer
	Inputs: Requirement Change, Object Code, Unit Test Package
	Outputs: Test Results, Feedback Regarding Code, Feedback Regarding Test Package, Notification of Successful Testing
	Constraints: - Can begin as soon as both Object Code and Unit Test Package are available - Ends when outputs have been provided
Monitor Progress	Responsibility: Project Manager
	Inputs: Requirement Change, Notification of Completion (from all tasks), Current Project Plans, Outcome Notification, Notification of Successful Testing, Decision Regarding Cancellation
	Outputs: Updated Project Plans, Notification of Revised Task, Cancel Recommendation
	Constraints: - Persists throughout the duration of the process - Ends when Test Unit has been successfully completed or cancellation of the whole ISPW process

The main VRPML graph for the ISPW-6 problem has already been given earlier in Figure 1. There are many aspects of the solution that can be elaborated further to capture the details (e.g. workspace descriptions), but space does not permit this. The complete solution is described in [8].

Two aspects of the VRPML solution shown in Figure 1 are worth highlighting. The first aspect relates to macro nodes. Figure 4 below illustrates the macro expansion for Test Unit. As seen in the figure, macro nodes serve as the modularization and abstraction facility for VRPML, apart from improving the graph's readability.

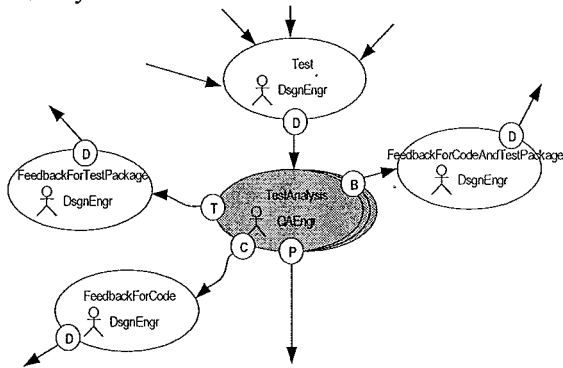


Figure 4. Macro Expansion for Test Unit in Figure 1

The second aspect also relates to modularization and abstraction, that is, through the decomposable transitions. In VRPML, decomposable transitions permit the specification of the activity to check whether or not the pre-conditions of the parent activity are satisfied before allowing the control-flow signal to be generated. Referring to Figure 1, an example of the decomposable transition (labeled D) can be seen attached to Modify Code. The sub-graph representing that transition is shown in Figure 5 below.

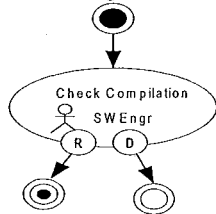


Figure 5. Sub-graph for Decomposable Transition labeled D in Modify Code

When Check Compilation fails, the assigned software engineer can select the transition R (for redo). As a result, a control-flow signal will be generated to re-enact its parent node (i.e. Modify Code) through a *re-enabled node* (shown as two white circles enclosing black circle). Otherwise, if the compilation is successful, the assigned engineer can select the transition D (for Done). In this case, the control-flow

signal will be generated and propagated back to the main graph to enable the subsequent connected node.

#### 4. Lessons Learned About VRPML

Upon generating the ISPW-6 solution, we observe some limitation in the VRPML's firing rule particularly relating to race condition, whereby two or more control-flow signals compete to enable a particular activity node. Figure 6 below illustrates the situation.

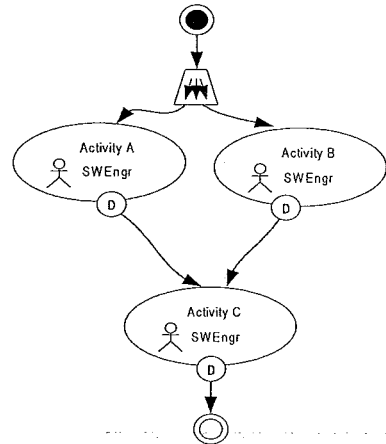


Figure 6. The Problem of Race Conditions

Although syntactically correct, the VRPML graph above represents an erroneous situation where activity C can be enabled when either the done transition of activity A or the done transition of activity B is selected. As a result, there is a possibility for activity C to be enabled twice.

To address this issue, we propose to amend the VRPML syntax and semantics. Instead of permitting multiple incoming arcs, the syntax of an activity node could be changed to allow only a single incoming arc to connect to it. Therefore, referring to Figure 6, the two arcs connections from the done transitions of activity A and B to activity C would be syntactically incorrect. In this way, the possibilities for race conditions would be eliminated.

This proposed change of syntax to activity nodes raises an issue relating to iteration. Disabling multiple incoming arcs would not permit iterations, hence, limiting the expressiveness of the VRPML notation. For example, the VRPML graph in Figure 7 below would give rise to a syntax error during compilation due to more than one arc connection to activity P (i.e. from the start node and the transition R of activity Q).

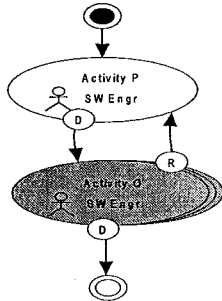


Figure 7. VRPML Feedback Example

As a solution to support iteration in a VRPML graph, a new decomposable cyclic node could be introduced. Figure 8 depicts the proposed notation for the decomposable cyclic node.



Figure 8. The Proposed Decomposable Cyclic Node Notation

Semantically, a decomposable cyclic node permits the specification of sub-graphs and allows the use of re-enabled nodes to re-enable the parent of those sub-graphs (i.e. the decomposable cyclic node itself). In this way, general feedback loops can be specified.

An example usage of the decomposable cyclic node called "Activity P and Q" alongside its decomposition is demonstrated below to express the feedback loop for the VRPML graph given earlier (see Figure 7).

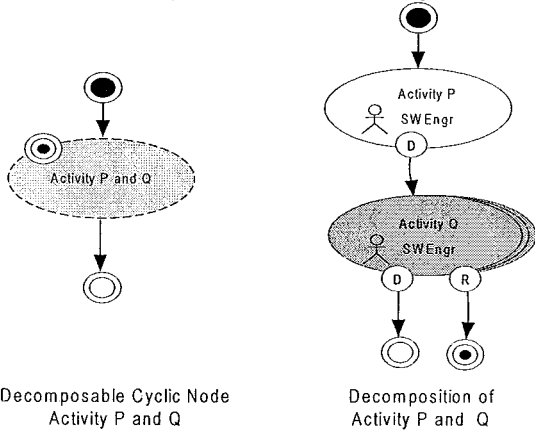


Figure 9. Example Usage of the Proposed Decomposable Cyclic Node

If the activity Q fails, the assigned software engineer can select the transition R (for re-do). As a result, a control-flow signal will be generated to enable its parent node (i.e. Activity P and Q) through a re-enabled node (shown as two white circles enclosing black circle). Otherwise, if the activity Q is successful,

the assigned engineer can select the transition D (for Done). In this case, the control-flow signal will be generated and propagated back to the main graph to enable the connected node.

Having considered the proposed changes to the VRPML syntax, Figure 10 below highlights the new partial VRPML solution to the ISPW-6 problem.

Two decomposable cyclic nodes can be used:

- (i) Modify Design and Review
- (ii) Test Unit and Analysis

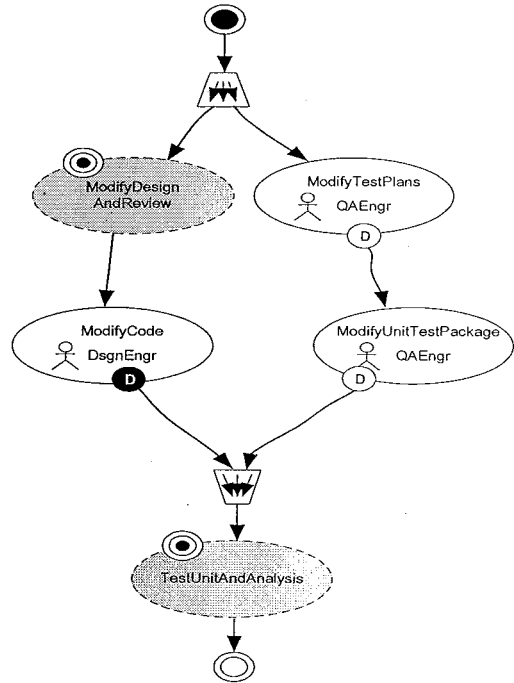


Figure 10. Using Decomposable Cyclic Node as part of the VRPML Solution to the ISPW-6 Problem

Because the decomposition for Modify Design and Review is relatively straightforward and similar to the case discussed earlier in Figure 8, it will not be developed further. Instead, only the decomposition for Test Unit and Analysis will be shown in Figure 10. Here, the decomposition of Test Unit and Analysis has captured the feedback requirement as stipulated by the ISPW-6 problem. Apart from addressing the possible race condition problems, it can be observed that the VRPML solution in Figure 10 and 11 can be more compact as compared to the ones given earlier. One reason is that the VRPML notation now becomes acyclic.

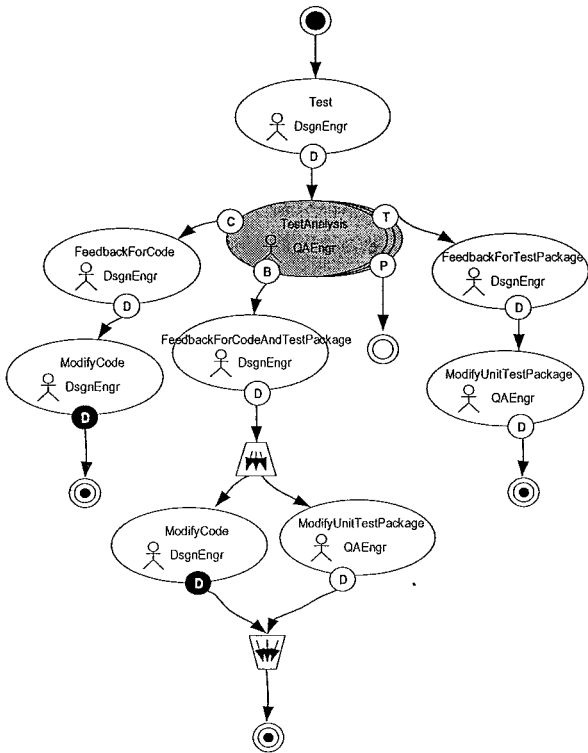


Figure 11. Decomposition of Test Unit and Analysis

## 5. Conclusion

In conclusion, this paper has presented a partial evaluation of a domain specific graph based visual language, called VRPML, whose firing rule are based on control flow model. This evaluation uncovers some of the limitations of associated with the control flow based firing rule and suggests some improvement in the language syntax and semantics to address these limitations. Such evaluation and its improvement suggestion can hopefully provide valuable guidance for the design of other graph based visual languages in the near future.

## Acknowledgement

The work undertaken in this research is partially funded by the USM Short Term Grants – “The Design and Implementation of the VRPML Runtime Environment”.

## References

[1] W.B. Ackerman, Data Flow Languages. In *IEEE Computer*, pp 15-23, February 1982

[2] T. Agerwala, and Arvind, 1982. Data Flow Languages. In *IEEE Computer*, pp. 10-13, February 1982.

[3] J.C. Doppke, D. Heimbigner, and A.L. Wolf. “Software Process Modeling and Execution within Virtual Environments”. *ACM Transactions on Software Engineering and Methodology*, 7 (1), January 1998. pp. 1-40.

[4] M.I. Kellner, P.H. Feiler, A. Finkelstein, T. Katayama, L.J. Osterweil, M.H. Penedo, and H.D. Rombach. “Software Process Modeling Example Problem”. In *Proc. of the 6th Intl. Software Process Workshop*, Hakodate, Hokkaido, Japan, October 1990. IEEE CS Press.

[5] K.N. Whitley, “Visual Programming Languages and the Empirical Evidence For and Against”. *Journal of Visual Language and Computing* 8 (1), January 1997, pp. 109-142.

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

[7] 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 2559, pp. 49-62, Rovaniemi, Finland, 2002, Springer.

[8] K.Z. Zamli. “Supporting Software Processes for Distributed Software Engineering Teams”, School of Computing Science, Univ. of Newcastle upon Tyne, PhD Thesis (Oct 2003).

[9] K.Z. Zamli, and Nor Ashidi Mat Isa, “The Computational Model for a Flow-based Visual Language”, in *Proc. of the AIDIS Intl. Conf. in Applied Computing 2005*, Algarve, Portugal, pp.217-224 , Feb 22-25, 2005.

[10] K.Z. Zamli, Nor Ashidi Mat Isa, and Norazlina Khamis, “The Design and Implementation of the VRPML Support Environment”, *Malaysian Journal of Computer Science* 18 (1), June 2005, pp. 57-69.

[11] K.Z. Zamli, Nor Ashidi Mat Isa, Nor Azlina Khamis, “Implementing Executable Graph Based Visual Language in a Distributed Environment”, in *Proc. of the IEEE Intl. Conf. on Computing and Informatics*, Kuala Lumpur, June 2006.



# Modeling and Enacting the ISPW-6 Problem

KAMAL ZUHAIRI ZAMLI AND NOR ASHIDI MAT ISA  
SCHOOL OF ELECTRICAL AND ELECTRONICS,  
USM ENGINEERING CAMPUS, 14300 NIBONG TEBAL  
PENANG, MALAYSIA

## ABSTRACT

*Software processes relate to sequences of steps that must be carried out by software engineers to pursue the goals of software engineering. For the last 20 years, modeling of software processes using a process modeling languages (PMLs) and its enactment has gained much interest, notably to provide guidance, automation and enforcement of procedures and policies in software engineering. To assist comparison and contrast among various approaches, the ISPW-6 software process was defined by the software engineering community. In this paper, we present a step by step solution to the ISPW-6 problem using a PML called Virtual Reality Process Modeling Language (VRPML). In doing so, we also discuss some of the novel features of VRPML.*

## ABSTRAK

*Proses perisian merangkumi turutan aktiviti yang perlu dilakukan oleh jurutera perisian untuk mencapai matlamat kejuruteraan perisian. Sepanjang 20 tahun kebelakangan ini, pemodelan proses perisian menggunakan bahasa permodelan proses dan lariannya menarik minat ramai sebagai panduan, automasi, dan pengekangan kepada prosedur dan polisi kejuruteraan perisian. Untuk membantu dalam perbandingan antara bahasa permodelan lain, masalah proses perisian ISPW-6 telah dikenalpasti oleh komuniti kejuruteraan perisian. Manuskrip ini membincangkan penyelesaian langkah demi langkah kepada masalah ISPW-6 menggunakan Bahasa Permodelan Realiti Maya (VRPML). Manuskrip ini juga membincangkan ciri-ciri novel VRPML.*

## INTRODUCTION

Engineering as a discipline relates to the creative application of mathematical and scientific principle to devise and implement solutions to problems in our everyday lives in an economic and timely fashion. Essentially, engineering aims to provide an engineering solution for a given problem. To provide a quality solution, it is not usually sufficient to focus only on the final product. Often, it is also necessary to consider the *processes* involve in producing that product. For example, consider an assembly of a car. From the customer's perspective, it is the final product that matters (i.e. a quality car).

From an engineering perspective, such quality could not be achieved if some of the processes (e.g. assembly lines) are faulty. Although additional rework can fix the problems caused by the faulty assembly lines, this tends to raise the overall costs because it deals only with symptoms of the problem. In contrast, going to the cause of the problem and improving the process (e.g. the faulty assembly lines) avoids the introduction of quality defects in the first place and leads to better results with lower costs. As this example illustrates, it is through the processes that engineers can observe and improve quality, control productions costs and possibly reduce the time to market their products.

Similar analogies can be applied in the case of software engineering. To produce quality software, it is also necessary to place emphasis on the processes by which the software is produced. In software engineering, these processes are usually called *software processes*. Software processes are often specified using a *process modeling language* (PML) and assisted by an environment called *Process Centered Software Engineering Environment* (PSEE). As its name suggests, a PML is used to construct a form of model of the actual software development process. Such a model is often called process model. In addition to being able to support the modeling of software processes, PMLs may also allow execution of the process models to support the activities of software engineers. The execution of such processes is usually termed *process enactment*.

For the last 20 years, modeling of software processes using a process modeling languages (PMLs) and its enactment has gained much interest, notably to provide guidance, automation and enforcement of procedures and practices in software engineering. Various approaches have been proposed ranging from the use of Petri Nets (e.g. SLANG (Bandinelli et al. 1994), Funsoft Nets (Emmerich and Como 1991)), database languages (e.g. ADELE-TEMPO (Belkhatir et al. 1994)), textual process modeling languages (e.g. JIL (Sutton Jr. and Osterweil 1997), ALF (Canals et al. 1994)) and visual process modeling languages (e.g. APEL (Dami et al. 1998), DYNAMITE (Heiman et al. 1996), Little JIL (Wise 1998), PROMENADE (Franch and Ribo 2003)). A survey of PMLs is actually beyond the scope of this paper but can be found elsewhere in (Huff 1996; Conradi and Jaccheri 1999; Zamli and Mat Isa 2004).

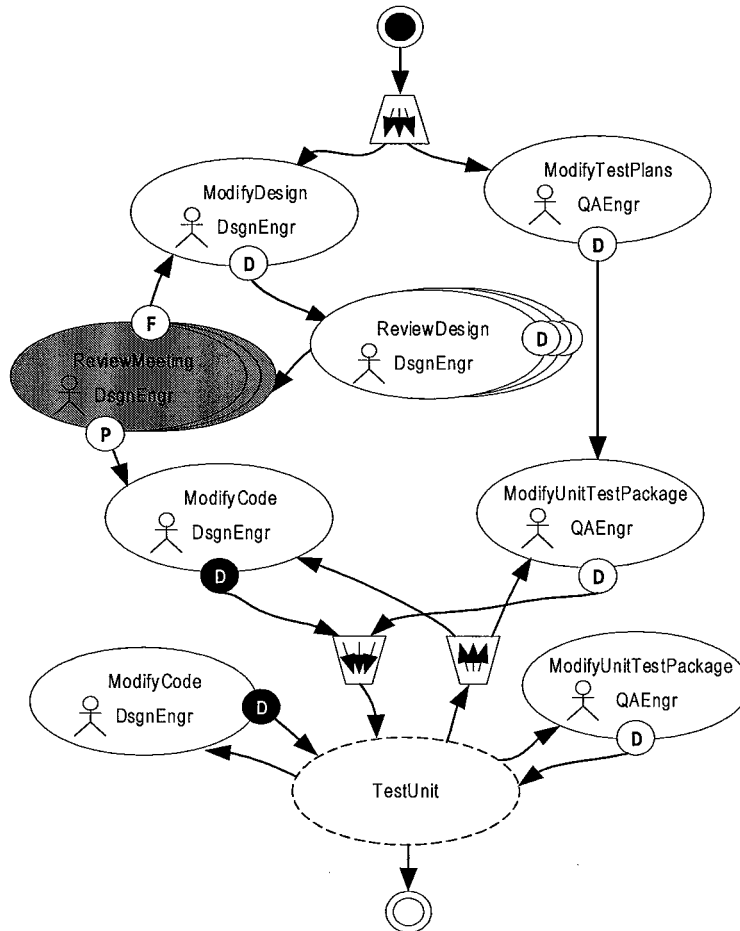
To assist comparison and contrast among various approaches, the ISPW-6 problem (Kellner et al. 1990) has been defined by the software engineering community. In this paper, we present a step by step solution to the ISPW-6 problem using our own PML called Virtual Reality Process Modeling Language (VRPML) (Zamli and Lee 2002; Zamli and Lee 2003; Zamli et al. 2005). In doing so, we also discuss some of the novel features of VRPML.

## OVERVIEW OF VRPML

VRPML is our research vehicle of investigating the issues relating to need the support for dynamic creation tasks and allocation of resources (in terms of software engineers, artifacts and tools) (Zamli and Lee 2002), as well as the human dimensions in terms of the support for process awareness, user awareness and process visualization (Zamli and Lee 2001).

Software processes are written in VRPML as graphs, by interconnecting nodes from top to bottom using arcs carrying control flow signals. In terms of its structure, VRPML graphs are cyclic. In terms of the language computational model, VRPML is a control-flow based visual language which supports modeling and enacting of software processes in a virtual environment. Software processes are generically modeled and resources can be dynamically assigned and customized for specific projects.

As an illustration for VRPML syntax, Figure 1 depicts the main VRPML graph of the ISPW-6 problem. The overall explanation of the ISPW-6 problem and the VRPML solution of the problem will be fully discussed in the later part of this manuscript.



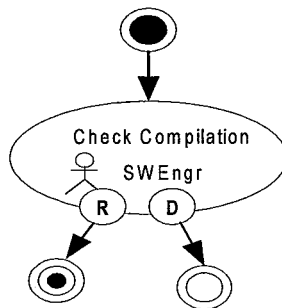
**FIGURE 1. VRPML Solution of the ISPW-6 Problem**

Similar to JIL (Sutton Jr. and Osterweil 1997) and Little JIL (Wise 1998) software processes in VRPML are described using process steps, which represent the most atomic representation of a software process (i.e. the actual task that software engineers are expected to perform). These process steps are represented as nodes, called *activity nodes* (shown as small ovals with stick figures). VRPML supports a number of different kinds of activity nodes as shown in Figure 1.

The activity nodes are: general purpose activity nodes (e.g. modify design, modify test plan, modify code, modify unit test package); multi-instance activity nodes (e.g. review design); and meeting nodes (e.g. review meeting). Activity nodes are

parameterized node accepting a role assignment as a parameter which may be used to allocate a specific software engineer to the task. In a meeting activity node and a multi-instance activity node, the depth of activity can also be specified (describe below) as a resource in terms of how many software engineers will be involved.

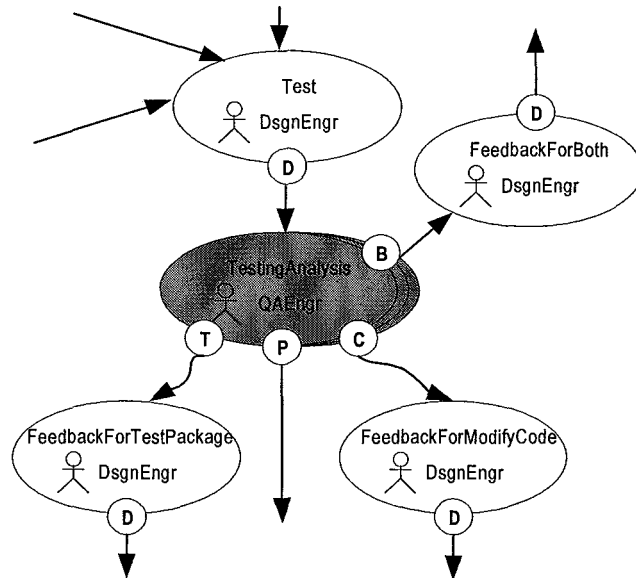
The firing of activity nodes is controlled by the arrival of a control flow signal. In VRPML, an initial control flow signal is always generated from a *start node* (a white circle enclosing a small black circle). A *stop node* (a white circle enclosing another white circle) does not generate any control flow signals. Control flow signals may also be generated at the completion of a node, often from special completion events called *transitions* (shown as small white circles with a capital letter, attached to an activity node) or *decomposable transitions* (small black circles with a capital letter). Decomposable transitions enable automation scripts or sub-graphs to be specified (and executed if selected) before allowing transition to generate a control flow signal. This is to ensure that certain required post-conditions have been satisfied before allowing the completion or cancellation of an activity. For example, the sub-graph associated with the decomposable transition Done (labeled D) for the activity node called modify code is given in Figure 2.



**FIGURE 2. Sub-graph for the decomposable transition labeled D in Modify Code**

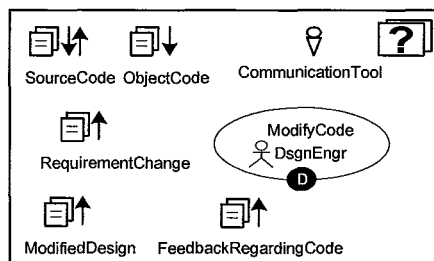
If the transition representing Redo (labeled R) in the activity node check compilation in Figure 2 is selected by the assigned software engineer (i.e. code do not compile), a control flow signal will be generated and will automatically re-enable its parent activity node the modify code through a *re-enabled node* (shown as two overlapping circles enclosing a black circle) .

VRPML allows activity nodes to be enacted in parallel using multi-instance nodes (overlapping ovals) or combinations of language elements called *merger* and *replicator nodes* (trapezoidal boxes with arrows inside). To improve readability, a set of VRPML nodes can be grouped together and replaced by a *macro node* (shown as dotted line ovals), with the macro expansion appearing on a separate graph. For example, referring to Figure 1, Test Unit is a macro node. The macro expansion of Test Unit is given in Figure 3.



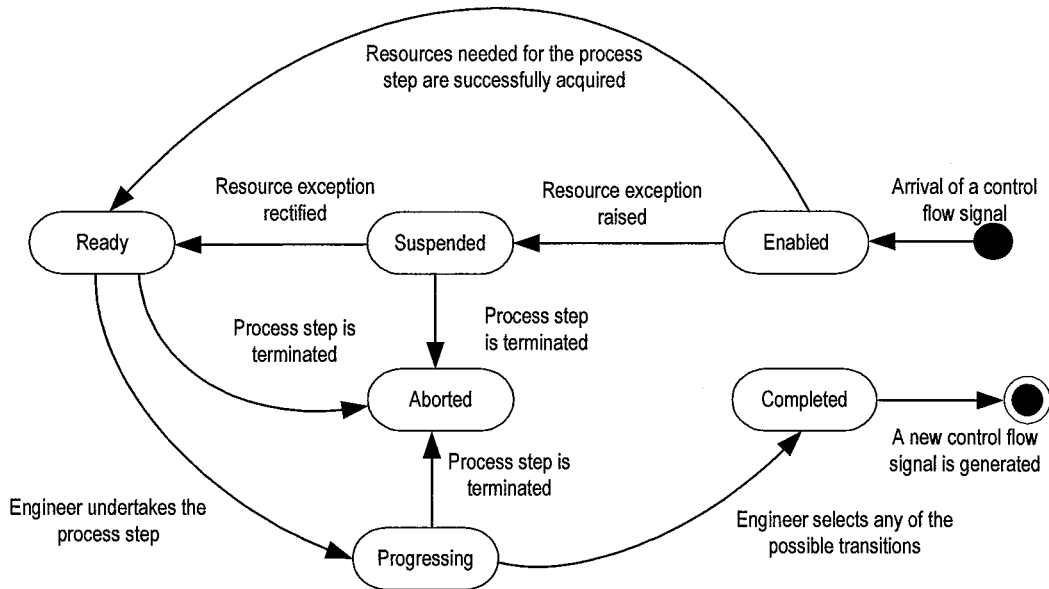
**FIGURE 3. Macro Expansion for Test Unit**

For every activity node, VRPML provides a separate *workspace*. Figure 4 depicts a sample workspace for the activity node called Modify Code in Figure 1. A workspace, the concepts borrowed from ADELE-TEMPO (Belkhatir et al. 1994), typically gives a *work context* of an activity as it hosts resources needed for enacting the activity: transitions, artifacts (shown as overlapping two overlapping documents with arrows for depicting access rights), communication tools (shown as a microphone) and any task descriptions (shown as a question mark). Effectively, when an activity node is enacted, the workspace is mapped into a virtual room, transitions into buttons, and artifacts, communication tools and task description into objects which may be manipulated by the assigned software engineer to complete the particular task at hand.



**FIGURE 4. Sample workspace for an activity node**

As far as enactment is concerned, the enactment model for VRPML can be seen in Figure 5 expressed in terms of a state transition diagram.



**FIGURE 5. VRPML Enactment Model**

The behavior of the runtime systems supporting such an enactment model can be thought of as consisting of a single producer (VRPML interpreter) and multiple consumers (engineer's runtime support system) communicating using a shared tuple space as in Linda (Gelernter 1985). Upon the arrival of a control flow signal, an activity node will be in an **enabled** state. This is the case where the VRPML interpreter attempts to acquire resources (in terms of role assignments, artifacts, tools as well as depths of activity nodes) that the activity node needs. If resources are successfully acquired, the VRPML interpreter then "produces" the process step corresponding to that activity node in the tuple space. The engineer's runtime support system then "consumes" the process step putting it into a **ready** state. Ideally, in this state, the process step is made available in the to-do-list of the assigned software engineer. If for any reason, VRPML interpreter fails to acquire resources it needs, a resource exception (i.e. resource unknown or resource unavailable) will be thrown putting the enactment of that particular process step in the VRPML graph into a **suspended** state. In this state, VRPML interpreter automatically produces a process step in the tuple space for the administrator (in this case, it may be the project manager) to rectify the resource exception or completely terminate the process step (putting it in an **aborted** state). If a process step is terminated, the administrator may optionally terminate the overall enactment of the particular VRPML graph in question or manually re-enact connecting and enclosing nodes (e.g. in a decomposable transition) by providing the necessary control flow signals that they need to fire. If the resource exception is rectified, enactment of the particular VRPML graph can continue allowing VRPML interpreter again to "produce" the process step in the tuple space. This process step can then be put into a **ready** state once the engineer's runtime support system has consumed it. If an engineer selects that particular process step (in the **progressing** state), a workspace for that process step will appear as a virtual room with artifacts, transitions and communication tools as objects which software engineer can manipulate to complete the task. The process step is in the **completed** state when the

software engineer selects any one of the possible transitions regardless of its outcome (e.g. passed, failed, done, or aborted).

Having discussed the syntax and semantics of VRPML, the ISPW-6 problem will be revisited. A detailed discussion of the ISPW-6 problem will be offered along with the step by step solution expressed in VRPML.

### THE ISPW-6 PROBLEM

The ISPW-6 problem (Kellner et al. 1990) is a benchmark problem in research into PMLs. It concerns with the software requirement change request for an existing software component occurring either towards the end of the development phase or during maintenance phase of the software lifecycle.

The ISPW-6 problem starts in response to the software requirement change request with the project manager scheduling and assigning various engineering activity in the process. These activities include: Schedule and Assign Task; Modify Design; Review Design; Modify Code; Modify Test Plans; Modify Unit Test Package; and Test Unit. During the enactment of these activities, the project manager is also required to monitor their progress. In each of these activities, the ISPW-6 problem also defines restrictions in terms of the input and output artifacts, the role responsible for each activity as well as conditions for each activity initiation and termination.

Apart from defining restrictions in terms of the input and output artifacts, the role responsible for each activity as well as conditions for each activity initiation and termination, the ISPW-6 problem also defines specific ordering of the related activities which is shown in Figure 6.

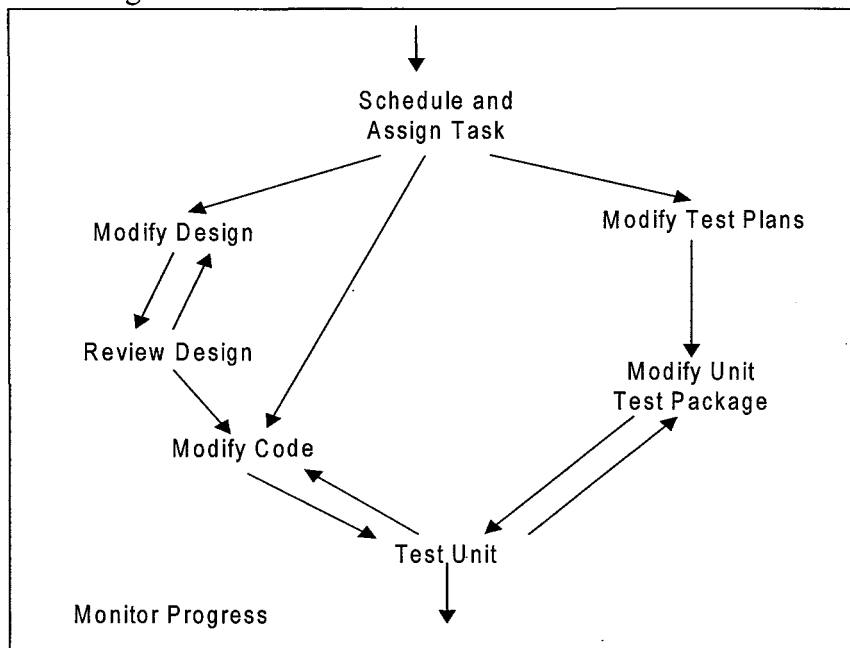


FIGURE 6. Flow of activities in the ISPW-6 Problem

Summing up, Table 1 summarizes the role responsibility, inputs and outputs as well as constraints defined for each of the activity defined in the ISPW-6 problem.

**TABLE 1. Summary of the ISPW-6 Problem**

Schedule And Assign Tasks	Responsibility: Project Manager
	Inputs: Requirement Change, Authorisation, Project Plans
	Outputs: Updated Project Plans, Notification of Task Assignments and Schedule Dates, Requirement Change
	Constraints: - Begins as soon as authorisation is given - Ends when outputs have been provided
Modify Design	Responsibility: Design Engineer
	Inputs: Requirement Change, Current Design, Design Review Feedback
	Outputs: Modified Design
	Constraints: - Can begin as soon as the task been assigned - Subsequent iteration can begin if design is not approved by the Review Design - Ends when outputs have been provided
Review Design	Responsibility: Design Review Team
	Inputs: Requirement Change, Modified Design
	Outputs: Design Review Feedback, Approved Modified Design, Outcome Notification
	Constraints: - Begins on schedule provided the modified design is available at the time - Ends when outputs have been provided
Modify Code	Responsibility: Design Engineer
	Inputs: Requirement Change, Modified Design, Current Source Code, Feedback Regarding Code
	Outputs: Modified Source Code, Object Code
	Constraints: - Can begin as soon as the task has been assigned even if Modify Design has not begun (discretion) - Ends when clean compilations are achieved, outputs have been provided and design is approved - Subsequent iteration can begin if required when test unit has completed
Modify Test Plans	Responsibility: QA Engineer
	Inputs: Requirement Change, Current Test Plans
	Outputs: Modified Test Plans
	Constraints: - Can begin as soon as the task has been assigned - Ends when outputs have been provided
Modify Unit Test Package	Responsibility: QA Engineer
	Inputs: Requirement Change, Modified Test Plans, Current Unit Test Package, Modified Design, Source Code, Feedback Regarding Test Package
	Outputs: Modified Unit Test Package
	Constraints: - Can begin as soon as Modify Test Plans has completed - Subsequent iteration can begin if required as Test Unit has completed - Ends when outputs have been provided
Test Unit	Responsibility: Design Engineer, QA Engineer
	Inputs: Requirement Change, Object Code, Unit Test Package
	Outputs: Test Results, Feedback Regarding Code, Feedback Regarding Test Package, Notification of Successful Testing
	Constraints: - Can begin as soon as both Object Code and Unit Test Package are available - Ends when outputs have been provided
Monitor Progress	Responsibility: Project Manager
	Inputs: Requirement Change, Notification of Completion (from all tasks), Current Project Plans, Outcome Notification, Notification of Successful Testing, Decision Regarding Cancellation
	Outputs: Updated Project Plans, Notification of Revised Task, Cancel Recommendation
	Constraints: - Persists throughout the duration of the process - Ends when Test Unit has been successfully completed or cancellation of the whole ISPW process



Having described and summarized the ISPW-6 problem in details, it is now appropriate that the solution expressed in VRPML is presented.

## **VRPML SOLUTION TO THE ISPW-6 PROBLEM**

As depicted in Figure 1, the ISPW-6 solution expressed in VRPML consists of one start node, one stop node, two replicator nodes, one merger node, six general purpose activity nodes, one meeting activity node, one multi-instance activity node as well as one macro node.

Two activities described in the ISPW-6 problem do not form as parts of the VRPML solution. The two activities are: Schedule and Assigned Task; and Monitor Progress. In VRPML solution of the ISPW-6 problem, these two activities are intentionally left out to demonstrate that VRPML solution is generic, and the enactment of the ISPW-6 problem (and VRPML graph in general) is dynamic in that scheduling of tasks and their assignment of resources can be incrementally achieved according to the dynamic need of a particular project.

For the same reason, VRPML solution also intentionally ignores the fact that Modify Code can be started upon discretion of a project manager even when Modify Design and Review Design have not even begun (see flow of activities in Figure 6). This also relates to the dynamic nature of software processes that is, how can the project manager know in advance which source code to modify when the modify design and the review design have not even started. Even if the source code to be modified may be known in advance, allowing the coding activity to precede the design and review activities can often lead to poor and badly structured design. On the other hand, perhaps as a way to do prototyping, the coding activity may precede the design and the review activities but it should be modeled and enacted separately.

Apart from dynamic issues raised by Schedule and Assigned Tasks, Monitor Progress and Modify Code activities, most tasks describe in the ISPW-6 problem appear directly as activity nodes in the VRPML graph in Figure 1 with the exception of Review Design, and Test Unit. Review Design is actually broken down into two activities in the VRPML graph: Review Design; and Review Meeting. This is because Review Design actually involves two activities relating to the review of the design and the collective decision making process.

Test Unit, on the other hand, is broken down into five activities grouped into a macro node called Test Unit (describe below) involving: Test, Test Analysis; Feedback for Modify Code; Feedback for Test Package; and Feedback for Both. Similar to Review Design, the main rationale for breaking down Test Unit into five activities is that apart from testing, Test Unit also involves collective decision making process on the test outcome as well as giving the appropriate feedback on the test results.

In terms of its enactment, when the VRPML graph in Figure 1 is first enacted, a control flow signal is initially generated by the start node. This control flow signal is then replicated by the replicator node to enable the Modify Design and Modify Test Plans. Upon the arrival of the control flow signals for both activity nodes, VRPML

interpreter attempts to acquire the resources (in terms of the artifacts, roles and engineer's assignment) for both activities. If not successful, either resource has not yet been assigned or resource has been assigned but it is not available, resource exception will be thrown. In this case, the enactment of the activity whose resource exception is thrown will be suspended. In turn, VRPML interpreter automatically creates a task for the project manager to fix the resource exception or terminate the overall enactment. In this way, resource allocation can be done dynamically in VRPML. This is an important feature for a PML as resources in software processes are dynamic and rarely can they be completely specified ahead of time.

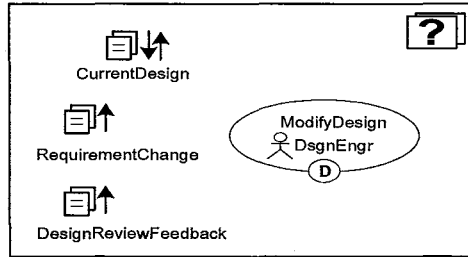
Assuming that no resource exceptions are thrown for both activity nodes, Modify Design and Modify Test Plans can now appear in the to-do-list of the assigned software engineer. Once the assigned software engineer chooses to undertake the activity, the workspace of the activity will be opened in a virtual environment. To allow completion or cancellation of an activity, the software engineer may select from any one of the given transitions which appear as objects in a virtual environment. In turn, the selected transition will automatically generate the appropriate control flow signal to support further enactment. As the enactment of the VRPML graph in Figure 1 is relatively straightforward, it will not be traced further.

Nevertheless, there are a number of issues worth mentioning regarding to the enactment of VRPML graph in Figure 1. The first issue relate to the enactment of the multi-instance activity node called Review Design and the meeting node called Review Meeting. These two tasks are actually representing Review Design activity in Table 1. As discussed earlier, the depths the multi-instance activity node and the meeting activity node, which correspond to how many software engineers involved, can also be dynamically specified as a resource that is, they are also subjected to resource exception. Thus, apart from the resource relating to the assignment of software engineers, tools, and artifacts, how many software engineers that will be assigned for the review design and the review meeting can also optionally be specified either before or dynamically during enactment. This feature enables VRPML to support dynamic creation of tasks according to the need of a particular project.

The second issue relates to the enactment of a macro node called Test Unit. This macro node actually represents Test Unit activity in Table 1 which consists of a number of tasks. These tasks are: Test; Feedback for both; Feedback for Test Package; Feedback for Modify Code; and Testing Analysis. During enactment, when a control flow signal encounters a Test Unit macro, the enacted graph is rewritten to include the tasks defined in that macro.

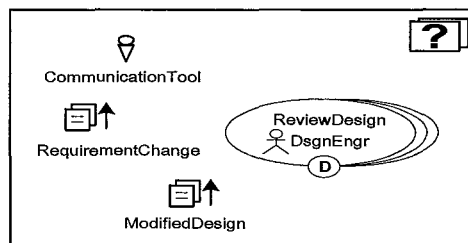
As far as work context is concerned, each activity node (shown in Figure 1, Figure 2, and Figure 3 earlier) must always be accompanied by its respective workspace along with the appropriate definition of resource and transitions. The definition of workspaces, resource, and transitions will be discussed next.

The workspace for an activity node called Modify Design in Figure 1 is defined in Figure 7. It consists of a transition called done and three artifacts consisting of Current Design, Requirement Change, and Design Review Feedback along with their appropriate access rights.



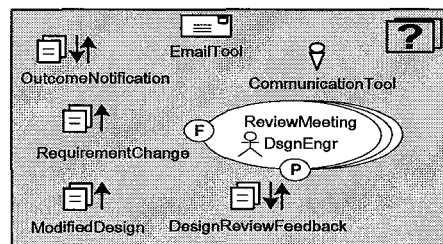
**FIGURE 7. Workspace for an activity node called Modify Design**

The workspace for a multi-instance activity node called Review Design in Figure 1 is defined in Figure 8. It consists of a transition called (D)one, a synchronous communication tool and two artifacts consisting of Requirement Change, and Modified Design along with their appropriate access rights.



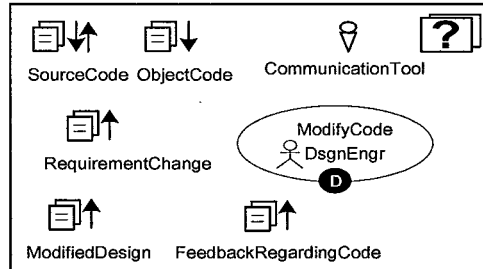
**FIGURE 8. Workspace for a multi-instance activity node called Review Design**

The workspace for a meeting activity node called Review Meeting in Figure 1 is defined in Figure 9. It consists of two transitions (called (P)assed and (F)ailed), both asynchronous and synchronous communication tool and four artifacts consisting of Requirement Change, Design Review Feedback, Outcome Notification and Modified Design along with their appropriate access rights. It must be stressed that the workspace for different types of activity nodes are unique. The reason for having a unique workspace is to support a sense of process awareness during process enactment. For instance, software engineers are able to distinguish whether the process steps that they are undertaking also concurrently involve other software engineers – the case for multi-instance and meeting activities. Such awareness should encourage inter-person communications, which is seen as one of the important aspect of supporting collaborative work (Yang 1995).



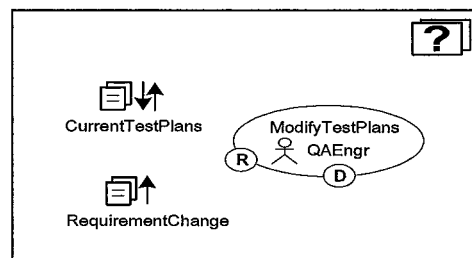
**FIGURE 9. Workspace for a meeting activity node called Review Meeting**

The workspace for an activity node called Modify Code in Figure 1 is defined in Figure 10. It consists of a decomposable transition called (D)one, a synchronous communication tool and five artifacts consisting of Requirement Change, Source Code, Object Code, Feedback Regarding Code and Modified Design along with their appropriate access rights.



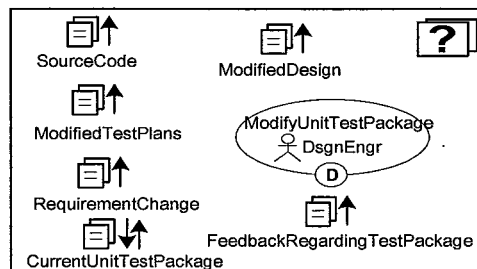
**FIGURE 10. Workspace for an activity node called Modify Code**

The workspace for an activity node called Modify Test Plan in Figure 1 is defined in Figure 11. It consists of two transitions called (D)one and (R)edo and two artifacts consisting of Requirement Change, and Current Test Plans along with their appropriate access rights.



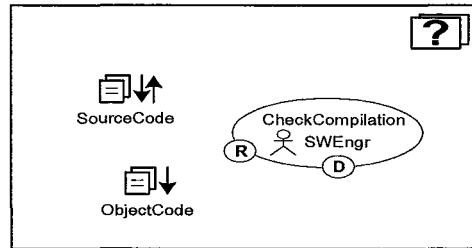
**FIGURE 11. Workspace for an activity node called Modify Test Plans**

The workspace for an activity node called Modify Unit Test Package in Figure 1 is defined in Figure 12. It consists of a transition called (D)one, and six artifacts consisting of Requirement Change, Source Code, Modified Test Plans, Current Unit Test Package, Feedback Regarding Test Package and Modified Design along with their appropriate access rights.



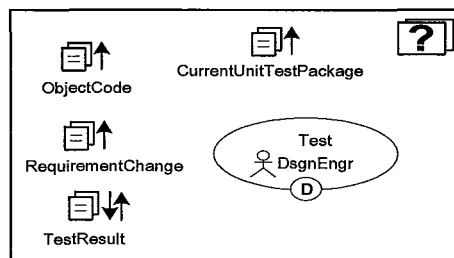
**FIGURE 12. Workspace for an activity node called Modify Unit Test Package**

The workspace for an activity node called Check Compilation (a sub-graph of the decomposable transition D for activity node called Modify Code) in Figure 2 is defined in Figure 13. It consists of two transitions called (D)one and (R)edo, and two artifacts consisting of Source Code, and Object Code along with their appropriate access rights.



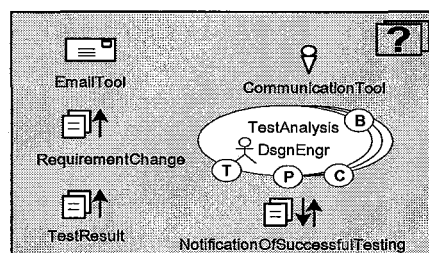
**FIGURE 13. Workspace for an activity node called Check Compilation**

The workspace for an activity node called Test in Figure 3 is defined in Figure 14. It consists of a transition called (D)one, and four artifacts consisting of Current Unit Test Package, Requirement Change, Test Result, and Object Code along with their appropriate access rights.



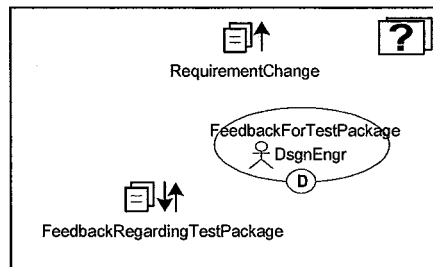
**FIGURE 14. Workspace for an activity node called Test**

The workspace for a meeting activity node called Test Analysis (from Test Unit macro) in Figure 3 is defined in Figure 15. It consists of four transitions (called (P)ass, (T)est, (B)oth, and (C)ode), both synchronous and asynchronous tools as well as three artifacts consisting of Requirement Change, Test Result, and Notification of Successful Testing along with their appropriate access rights.



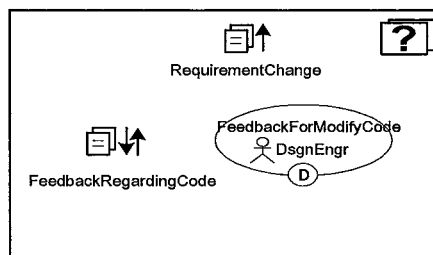
**FIGURE 15. Workspace for a meeting activity node called Test Analysis**

The workspace for an activity node called Feedback for Test Package (from Test Unit macro) in Figure 3 is defined in Figure 16. It consists of a transition called (D)one, and two artifacts consisting of Requirement Change, and Feedback Regarding Test Package along with their appropriate access rights.



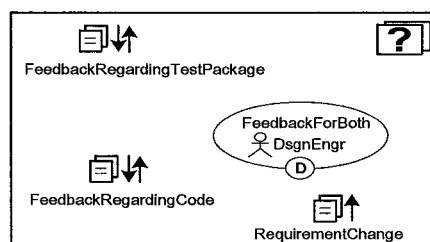
**FIGURE 16. Workspace for an activity node called Feedback for Test Package**

The workspace for an activity node called Feedback for Modify Code (from Test Unit macro) in Figure 3 is defined in Figure 17. It consists of a transition called (D)one, and two artifacts consisting of Requirement Change, and Feedback Regarding Code along with their appropriate access rights.



**FIGURE 17. Workspace for an activity node called Feedback for Modify Code**

The workspace for an activity node called Feedback for Both (from Test Unit macro) in Figure 3 is defined in Figure 18. It consists of a transition called (D)one, and three artifacts consisting of Requirement Change, Feedback Regarding Test Package and Feedback Regarding Code along with their appropriate access rights.



**FIGURE 18. Workspace for an activity node called Feedback for Both**

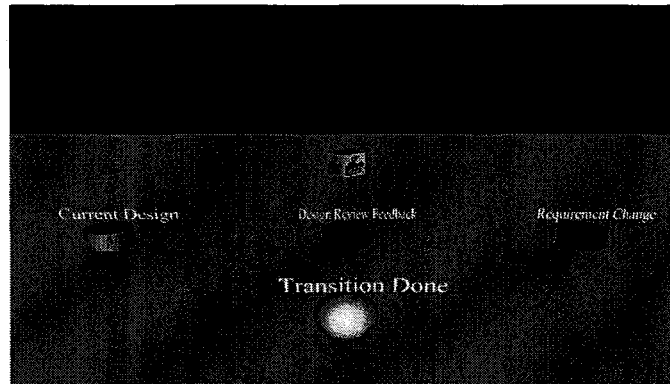
## DISCUSSION AND LESSONS LEARNED

Throughout the previous section, we have presented a step by step solution to the ISPW-6 problem expressed in VRPML. A number of lessons learned in terms of effectiveness of representation, language expressiveness, and modularity as well as scalability of VRPML. The lessons are summarised below:

- **Effectiveness of representations:** Because VRPML notations are purely graphical, it seems straightforward to make sense out of the VRPML representations. In fact, to some extent, the general structure of VRPML graph resembles that of a flowchart.
- **Language expressiveness:** It is difficult to measure expressiveness of VRPML simply by modelling and enacting one specific problem as the ISPW-6. However, because the ISPW-6 problem is designed by experts in the field, and the fact that VRPML can straightforwardly model the solution is a positive indication about its expressiveness. Nonetheless, because VRPML is a control-flow based language, it suffers from the problem of *race condition*, that is, two or more control flows can compete to enable a particular node. The problem is currently being address in the next version of the language (Zamli and Mat Isa 2005).
- **Modularity and Scalability:** VRPML graph is highly modular in the sense that activities are modelled as a step in a software process. VRPML also provides a macro node which can group one or more nodes together. One obvious limitation is that VRPML suffers from the problem of scale, that is, it takes much space. Even a small problem like the ISPW-6, VRPML solution consists of fourteen different types of activity nodes (including macro expansion) as well as fourteen corresponding workspaces. One point to note is that this limitation is inherent in any graph based visual language.

Summing up, VRPML solution is characterized by a number of novel features:

- Software processes, expressed as process steps by activity nodes, are also described in terms of workspaces which host artifacts and tools, and can be represented in a virtual environment. The concept of workspaces in which software engineers can perform their tasks is not new as it can also be seen in ADELE-TEMPO (Belkhatir et al. 1994), but the way that workspace is integrated with a virtual environment in VRPML is. This opens up a possibility for supporting process awareness, user awareness and process visualization utilising a virtual environment. As an illustration, Figure 19 depicts a snapshot from the enactment of the ISPW-6 for an activity node called Modify Design.



**FIGURE 18. Workspace snapshot for Modify Design seen during enactment**

- Resources in terms of software engineers, artifacts, and tools needed for the software processes can be dynamically assigned. This is especially important in the case of activity nodes involving multiple software engineers (e.g. multi-instance activity nodes and meeting activity nodes). How many engineers are assigned for such activity nodes (i.e. their depths) depends on the dynamic needs of a particular project. The support for dynamic creation of tasks and allocation of resources is also provided in Dynamite (Heiman et al. 1996) via on the fly process evolution based on graph rewriting, and in Alf (Canals et al. 1994) by treating the process models into small and independent process models which are enacted separately. While both approaches have successfully addressed the support for generic construction of process model as well as dynamic allocation of resources, they suffer from a number of limitations. Firstly, if on-the-fly process evolution is exploited, extra overhead may be introduced involving steps in to ensure that no *ad hoc* changes and no side effects are done to the process models which can be difficult and expensive to achieve. Secondly, if the process models are broken into a collection of smaller and independent process models, process models for large scale software project can be difficult to manage as there may be literally consists of hundreds of smaller and independent process models, each of which needs to be enacted separately.
- In line with the current trends of software engineers working across geographically and temporally distributed software engineering teams, VRPML also provide support for specifying virtual meetings. Supporting virtual meetings seem advantageous since meetings are an important characteristic of software engineering. Furthermore, virtual meetings could help reduce costs if a meeting would otherwise be held face to face. However, supporting a virtual meeting (for geographically and temporally distributed software engineering teams) raises an issue of time differences. While this issue is beyond the scope of VRPML, one solution might be that software engineers are given access to communication tools in a meeting activity node workspace to allow communication and scheduling of the virtual meeting at a time convenient to all parties.



## CONCLUSION

In conclusion, this paper describes a step by step solution of the ISPW-6 problem expressed in VRPML, a visual PML for modelling and enacting of software processes. Novel features introduced in VRPML include the support for dynamic creation of tasks and allocation of resources, and integration with a virtual environment at the PML enactment level. Currently, additional experimentations are planned to provide further evaluations of VRPML especially in the field of Workflow Management System (WFMS) (Zamli et al. 2005; Zamli and Mat Isa 2005b).

## ACKNOWLEDGEMENT

This project has been undertaken under the partial funding from the USM Short Term Grants – “The Design and Implementation of the VRPML Support Environment”.

## REFERENCES

- Bandinelli, S., Fuggetta, A., Ghezzi, C. and Lavazza, L. (1994). SPADE: An Environment for Software Process Analysis, Design and Enactment. *Software Process Modelling and Technology*. A. Finkelstein, J. Kramer and B. Nuseibeh. Taunton, England, Research Studies Press: 223-247.
- Belkhatir, N., Estublier, J. and Melo, W. (1994). ADELE-TEMPO: An Environment to Support Process Modelling and Enaction. *Software Process Modelling and Technology*. A. Finkelstein, J. Kramer and B. Nuseibeh. Taunton, England, Research Studies Press: 187-222.
- Canals, G., Boudjlida, N., Derniame, J.C., Godart, C. and Lonchamp, J. (1994). ALF: A Framework for Building Process-Centred Software Engineering Environments. *Software Process Modelling and Technology*. A. Finkelstein, J. Kramer and B. Nuseibeh. Taunton, England, Research Studies Press: 153-185.
- Conradi, R. and M. L. Jaccheri (1999). Process Modelling Languages. *Software Process: Principles, Methodology and Technology*. J. C. Derniame, B. A. Kaba and D. Wastell. Berlin-Heidelberg, Lecture Notes in Computer Science Volume 1500, Springer: 27-52.
- Dami, S., Estublier, J. and Amiour, M. (1998). APEL: A Graphical Yet Executable Formalism for Process Modelling. *Automated Software Engineering* 5(1): 61-96.
- Emmerich, W. and V. G. Como (1991). FUNSOFT Nets: A Petri-Net based Software Process Modeling Language. *Proceedings of the 6th International Workshop on Software Specification and Design*, Italy, IEEE Computer Society Press.

- Gelernter, D. (1985). Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems* 7(1): 80-112.
- Heiman, P., Joeris, G. and Krapp, C.A. (1996). DYNAMITE: Dynamic Task Nets for Software Process Management. *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, IEEE Computer Press: 331-341.
- Huff, K. E. (1996). Software Process Modeling. *Trends in Software Process*. A. Fuggetta and A. Wolf, John Wiley & Sons: 1-24.
- Kellner, M.I., Feiler, P.H., Finkelstein, A., Katayama, T., Osterweil, L.J., Penedo, M.H. and Rombach, H.D. (1990). Software Process Modeling Example Problem. *Proceedings of the 6th International Software Process Workshop*, Hakodate, Hokkaido, Japan, IEEE Computer Society Press.
- Franch, X. and Ribo, J.M. (2003). A UML-Based Approach to Enhance Reuse within Process Technology. *Proceedings of the 9th European Workshop on Software Process Technology*, LNCS Vol. 2786, Helsinki, Finland, 2003, Springer :74-93.
- Sutton Jr., S. and L. J. Osterweil (1997). The Design of a Next-Generation Process Language. *Proceedings of the Joint 6th European Software Engineering Conference and the 5th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, Lecture Notes in Computer Science Volume 1301, Springer.
- Wise, A. (1998). Little JIL 1.0 Language Report - Technical Report 98-24, Department of Computer Science, University of Massachusetts at Amherst.
- Yang, Y. (1995). Coordination for Process Support is Not Enough. *Proceedings of the 4th European Workshop on Software Process Technology*, Lecture Notes in Computer Science Volume 913, Springer.
- Zamli, K. Z. and Lee, P.A. (2001). Taxonomy of Process Modeling Languages. *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, IEEE Computer Society Press.
- Zamli, K. Z. and Lee, P.A. (2002). Exploiting a Virtual Environment in a Visual PML. *Proceedings of the 4th International Conference on Product Focused Software Process Improvements*, Rovaniemi, Finland, Lecture Notes in Computer Science Volume 2559, Springer.
- Zamli, K.Z. and Lee.P.A. (2003). Modeling and Enacting Software Processes Using VRPML. *Proceedings of the 10th IEEE Asia-Pacific Conference on Software Engineering*, IEEE CS Press: 243-252.

- Zamli, K.Z. and Mat Isa, N.A. (2004). A Survey and Analysis of Process Modeling Languages. *Malaysian Journal of Computing Science* Vol. 17, No 2: 68-89.
- Zamli, K.Z., Mat Isa, N.A., and Khamis, N. (2005). The Design and Implementation of the VRPML Support Environment. *Malaysian Journal of Computer Science* Vol. 18, No 1: 57-69.
- Zamli, K.Z., Mat Isa, N.A. and Ali, A.N. (2005). Coordinating Business Processes Using a PML. *Proceedings of the International Conference on Information Integration and Web-based Applications and Services (IIWAS 2005)*, Kuala Lumpur :445-455.
- Zamli, K.Z., and Mat Isa, N.A. (2005). The Computational Model for a Flow-based Visual Language. *Proceedings of the AIDIS International Conference in Applied Computing 2005*, Algarve, Portugal: 271-224.
- Zamli, K.Z. and Mat Isa, N.A (2005b). Enacting the waterfall software development model. *Jurnal Teknologi UTM (Siri D)*, in print for December 2005 issue.

# SOFTWARE FAULT INJECTION TOOL

MUHAMMAD IMRAN BIN AHMAD

UNIVERSITI SAINS MALAYSIA  
2005

ENHANCING AND EVALUATING A SOFTWARE  
FAULT INJECTION TOOL (SEIT)

MOHD DAUD B ALANG MASSAN

UNIVERSITI SAINS MALAYSIA  
2005

UNIT KUMPULAN WANG AMANAH  
 UNIVERSITI SAINS MALAYSIA  
 KAMPUS KEJURUTERAAN  
 SERI AMPANGAN  
 PENYATA KUMPULAN WANG  
 TEMPOH BERAKHIR 31/10/ 2006

DR KAMAL ZUHAIRI B ZAMLI

304.PELECT.6035127

THE DESIGN & IMPLEMENTATION OF THE VRPML SUPPORT ENVIROMENT

Tempoh Projek:01/11/2004 - 31/10/2006

JUMLAH GERAN :-

NO PROJEK :-

PANEL :- J/PENDEK

PENAJA :-

Vol	Peruntukan (a)	Perbelanjaan sehingga 31/12/2005 (b)	Tanggung semasa 2006 (c)	Belanja Semasa 2006 (d)	Jum. Belanja 2006 (c + d)	Jumlah Belanja Terkumpul (b+c+d)	Baki Peruntukan Semasa 2006 (a-(b+c+d))
11000 GAJI KAKITANGAN AWAM	8,321.00	3,754.78	0.00	0.00	0.00	3,754.78	4,566.22
21000 PERBELANJAAN PERJALANAN DAN SARA	2,791.00	0.00	0.00	212.75	212.75	212.75	2,578.25
23000 PERHUBUNGAN DAN UTILITI	650.00	70.00	0.00	0.00	0.00	70.00	580.00
27000 BEKALAN DAN ALAT PAKAI HABIS	5,700.00	5,684.00	1,970.00	2,448.00	4,418.00	10,102.00	(4,402.00)
28000 PERKHIDMATAN IKTISAS & HOSPITALITI	2,500.00	3,036.85	0.00	1,300.00	1,300.00	4,336.85	(1,836.85)
35000 HARTA-HARTA MODAL LAIN	0.00	1,400.00	0.00	0.00	0.00	1,400.00	(1,400.00)
	<u>19,962.00</u>	<u>13,945.63</u>	<u>1,970.00</u>	<u>3,960.75</u>	<u>5,930.75</u>	<u>19,876.38</u>	<u>85.62</u>
Jumlah Besar	19,962.00	13,945.63	1,970.00	3,960.75	5,930.75	19,876.38	85.62

**BAHAGIAN PENYELIDIKAN & PEMBANGUNAN  
CANSELORI  
UNIVERSITI SAINS MALAYSIA**

**Laporan Akhir Projek Penyelidikan Jangka Pendek**

1) Nama Penyelidik: DR KAMAL ZUHAIRI BIN ZAMLI  
.....  
.....

Nama Penyelidik-Penyelidik  
Lain (Jika berkaitan) : DR NOR ASHIDI MAT ISA  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....

2) Pusat Pengajian/Pusat/Unit : KEJURUTERAAN ELEKTRIK DAN ELEKTRONIK  
.....  
.....

3) Tajuk Projek: THE DESIGN AND IMPLEMENTATION OF THE  
VRPML SUPPORT ENVIRONMENT  
.....  
.....  
.....  
.....

4) (a) **Penemuan Projek/Abstrak**

*(Perlu disediakan maklumat di antara 100 - 200 perkataan di dalam Bahasa Malaysia dan Bahasa Inggeris Ini kemudiannya akan dimuatkan ke dalam Laporan Tahunan Bahagian Penyelidikan & Pembangunan sebagai satu cara untuk menyampaikan dapatan projek tuan/puan kepada pihak Universiti).*

.....

LIHAT LAMPIRAN A

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....



(b) Senaraikan Kata Kunci yang digunakan di dalam abstrak:

Bahasa Malaysia

Bahasa Inggeris

- |                              |                                |
|------------------------------|--------------------------------|
| .....                        | .....                          |
| 1 - BAHASA PERMODELAN PROSES | 1 - PROCESS MODELLING LANGUAGE |
| 2 - KEJURUTERAAN PERISIAN    | 2 - SOFTWARE ENGINEERING       |
| 3 - VRPML                    | 3 - VRPML                      |
| .....                        | .....                          |
| .....                        | .....                          |
| .....                        | .....                          |
| .....                        | .....                          |
| .....                        | .....                          |
| .....                        | .....                          |
| .....                        | .....                          |

5) Output Dan Faedah Projek

(a) Penerbitan (termasuk laporan/kertas seminar)  
(Sila nyatakan jenis, tajuk, pengarang, tahun terbitan dan di mana telah diterbitkan/dibentangkan).

- .....
- LIHAT LAMPIRAN B UNTUK SENARAI PENERBITAN
  - LIHAT LAMPIRAN C UNTUK ARTIKEL YANG SEDANG DINILAI
- .....
- .....
- .....
- .....
- .....
- .....
- .....

- (b) Faedah-Faedah Lain Seperti Perkembangan Produk,  
Prospek Komersialisasi Dan Pendaftaran Paten.  
(Jika ada dan jika perlu, sila gunakan kertas berasingan)

.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....

(c) Latihan Gunatenaga Manusia

- i) *Pelajar Siswazah* .....  
    — LIHAT LAMPIRAN D

.....  
.....

- ii) *Pelajar Prasiswazah:* .....

.....  
.....  
.....

- iii) *Lain-Lain :* .....

.....  
.....  
.....

6. Peralatan Yang Telah Dibeli:

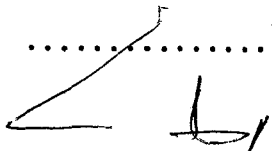
1 - PENSAMPIL JBUILDER } RM 1400  
2 - UML TOOL }

- LIHAT LAMPIRAN E

---

UNTUK KEGUNAAN JAWATANKUASA PENYELIDIKAN UNIVERSITI

Laporan telah diterima



T/TANGAN PENERUSI  
J/K PENYELIDIKAN  
PUSAT PENGAJIAN

PROF. MADYA DR. SOH SITI  
Pemangku Timbalan Dekan  
(Pengajian Stewardship Dan Penyelidikan)  
Pusat Pengajian Kejuruteraan Elektronik dan Sistem  
Universiti Sains Malaysia  
Kampus Kejuruteraan



**USM**

UNIVERSITI SAINS MALAYSIA

**PUSAT PENGAJIAN KEJURUTERAAN ELEKTRIK DAN ELEKTRONIK**  
**SCHOOL OF ELECTRICAL AND ENGINEERING**

16 November 2006

Pn Latifah Abdul Latif  
Assistant Registrar,  
RCMO Office,  
Bangunan Canselori  
USM,  
11800 Pulau Pinang

**Short Term Grants Final Report**

Attached is the final report and latest account statement for the short term grants entitled: "The Design and Implementation of the VRPML Support Environment" for your evaluation.

Thanks in advance.

Yours truly,

Dr. Kamal Zuhairi Zamli  
Ext: 6079  
Email: eekamal@eng.usm.my