Jurnal Antarabangsa (Teknologi Maklumat) 3(2002): 53-72

VRPML Solution to the ISPW-6 Problem

KAMAL Z. ZAMLI

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 15 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 our own PML called Virtual Reality Process Modeling Language (VRPML). In doing so, we also discuss some of the novel features of VRPML.

Keywords: Process Modeling Languages, Software Processes, Software Engineering

ABSTRAK

Proses pembangunan applikasi berkait rapat dengan activiti-aktiviti yang mesti diambil oleh jurutera perisian untuk mencapai matlamat utama kejuruteraan perisian. Selalunya, proses pembangunan applikasi ini di terjemahkan sebagai model program melalui bahasa pengaturcaraan proses (PML) yang boleh di lari bertujuan untuk membantu, mengawal, dan mengautomasi aktiviti pembangunan perisian. Di dalam kertas kerja ini, kami menjelaskan secara terperinci penyelesaian kepada masalah ISPW-6 menggunakan bahasa pengaturcaraan process realiti maya (VRPML). Serentak dengan itu, kami juga menerangkan beberapa ciri-ciri novel yang terdapat pada VRPML.

INTRODUCTION

For the last 15 years, modeling of software processes using a process modeling languages (PMLs) and its execution (termed 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, Fuggetta et al. 1994), Funsoft Nets (Emmerich and Como 1991)), database languages (e.g. ADELE-TEMPO (Belkhatir, Estublier et al. 1994)), textual process modeling languages (e.g. JIL (Sutton Jr. and Osterweil 1997), ALF (Canals, Boudjlida et al. 1994)) and visual process modeling languages (e.g. APEL (Dami, Estublier et al. 1998), DYNAMITE (Heiman, Joeris et al. 1996), Little JIL (Wise 1998)). 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 2001).

To assist comparison and contrast among various approaches, the ISPW-6 problem (Kellner, Feiler 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). 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 June 2001). The novel features of VRPML are that it supports integration with a virtual environment and allows dynamic allocation of resources by exploiting the enactment model.

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 not be fully discussed until Section 4.0.

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

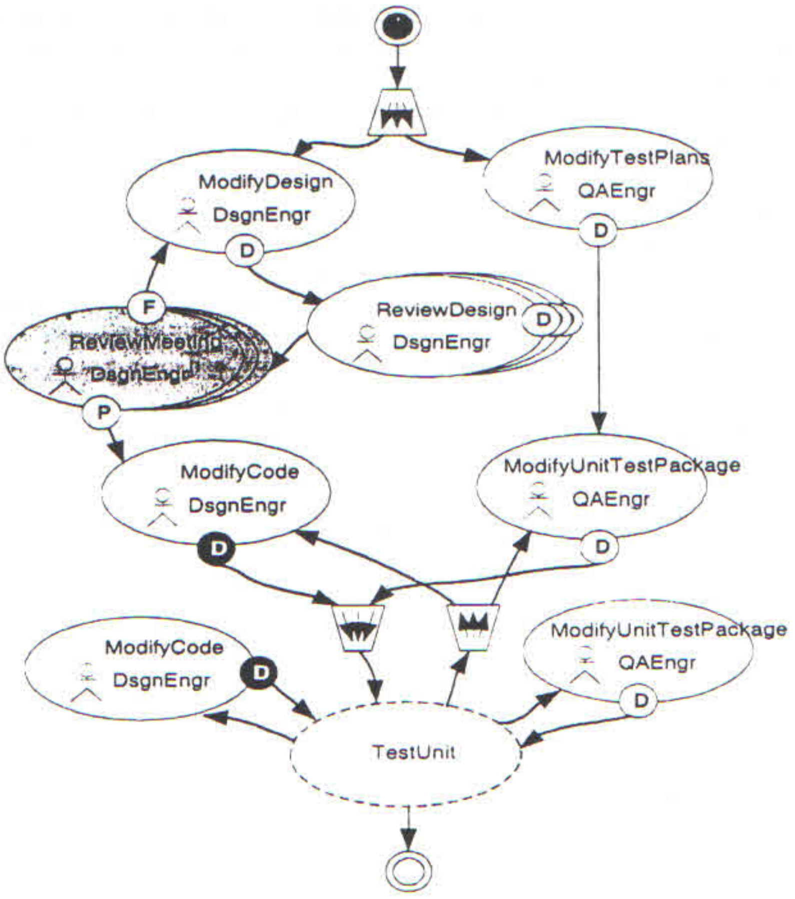

FIGURE 1. VRPML Solution of the ISPW-6 Problem

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 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) 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 representing Done (labeled D) for the activity node called modify code is given in Figure 2.

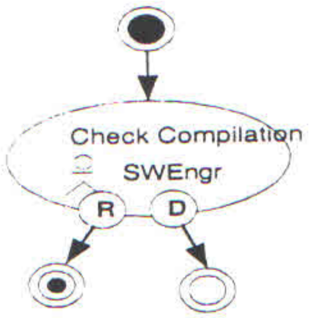

FIGURE 2. Sub-graph for the decomposable transition labeled D in Modify Code

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.

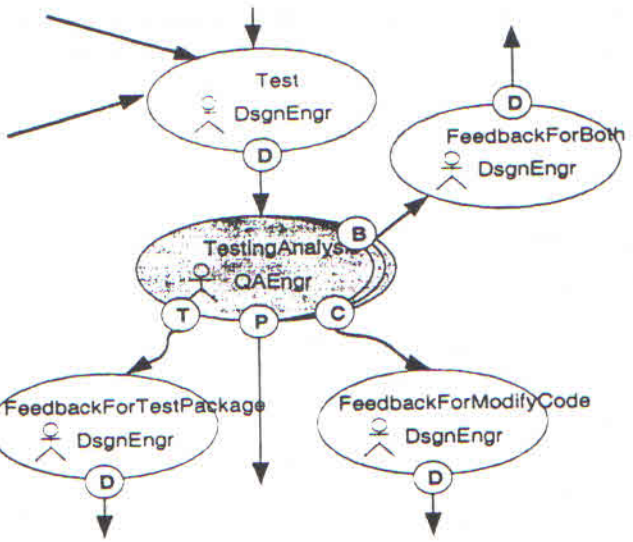

FIGURE 3. Macro Expansion for Test Unit

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, Estublier 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.

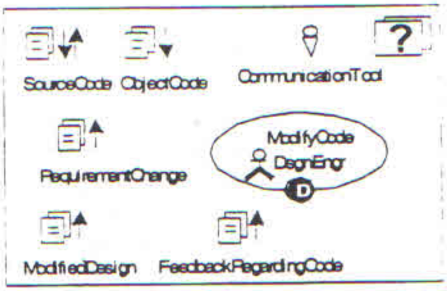

FIGURE 4. Sample workspace for an activity node

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.

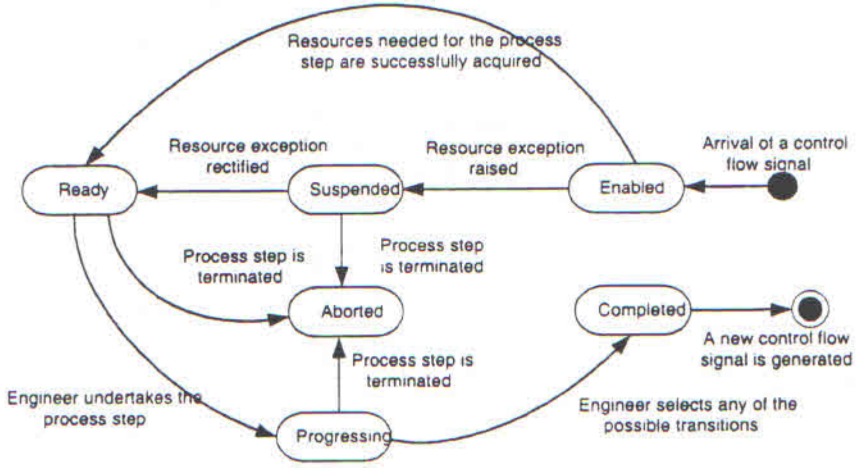
Resources needed for the process step are successfully acquired
Resource exception rectified
Resource exception raised
Arrival of a control flow signal
Ready
Suspended
Enabled
Process step is terminated
Process step is terminated
Aborted
Completed
A new control flow signal is generated
Process step is terminated
Engineer undertakes the process step
Progressing
Engineer selects any of the possible transitions

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).

It must be stressed that it is impossible to describe all the syntax and semantic of VRPML notation here due to space limitation, however, interested readers are referred to our earlier work in (Zamli and Lee 2002). In the next section, the ISPW-6 problem will be revisited. A detailed discussion of the ISPW-6 problem will be offered along with their solution expressed in VRPML.

THE ISPW 6.0 PROBLEM

The ISPW-6 problem (Kellner, Feiler 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 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. Table 1 summarizes the role responsibility, inputs and outputs as well as constraints defined for each of the activity.

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.

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

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 been assigned - Ends when outputs have been provided

cont. TABLE 1

Modify Unit Test Package	Responsibility: QA Manager 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

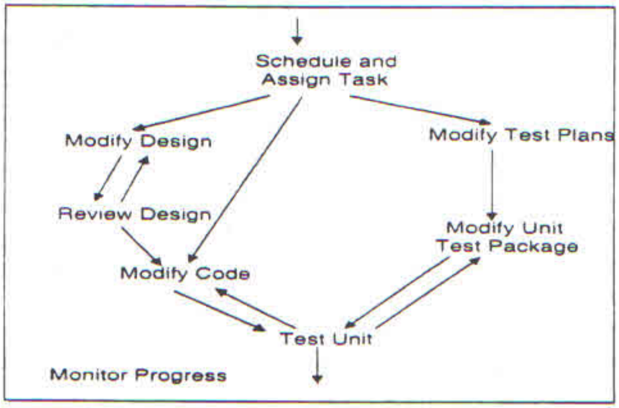

FIGURE 6. Flow of activities in the ISPW-6 Problem

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 straight forward, 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.

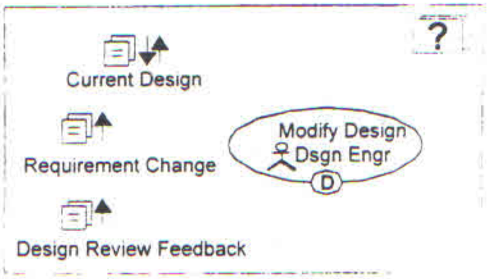

FIGURE 7. Workspace for an activity node called Modify Design

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.

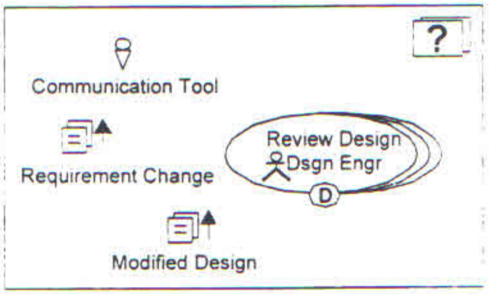

FIGURE 8. Workspace for a multi-instance activity node called Review Design

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).

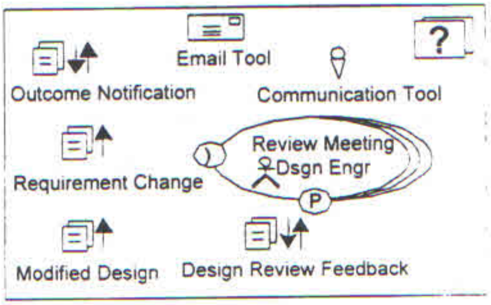

FIGURE 9. Workspace for a meeting activity node called Review Meeting

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.

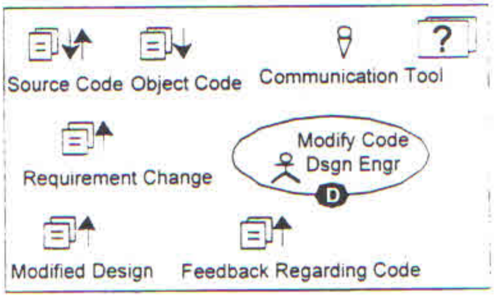

FIGURE 10. Workspace for an activity node called Modify Code

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

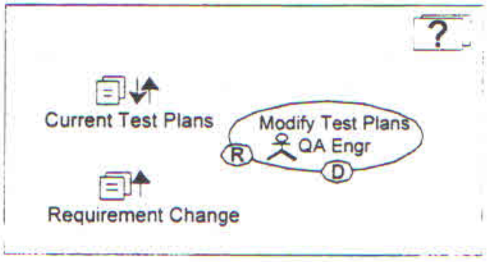

FIGURE 11. Workspace for an activity node called Modify Test Plans

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.

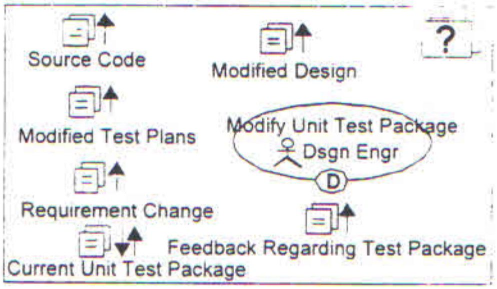

FIGURE 12. Workspace for an activity node called Modify Unit Test Package

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.

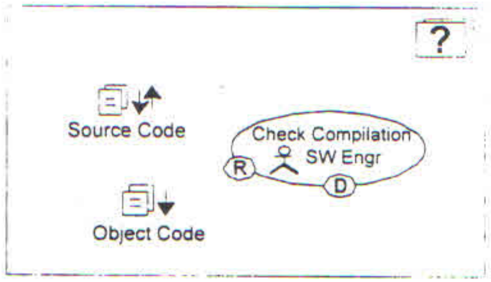

FIGURE 13. Workspace for an activity node called Check Compilation

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.

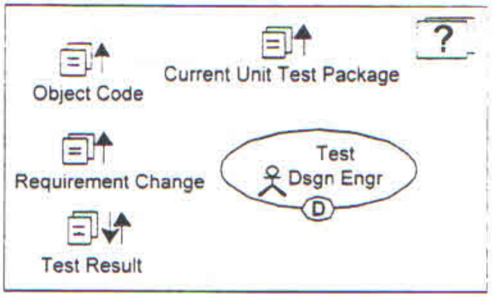

FIGURE 14. Workspace for an activity node called Test

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 synchronous tools as well as three artifacts consisting of Requirement Change, Test Result, and Notification of Successful Testing along with their appropriate access rights.

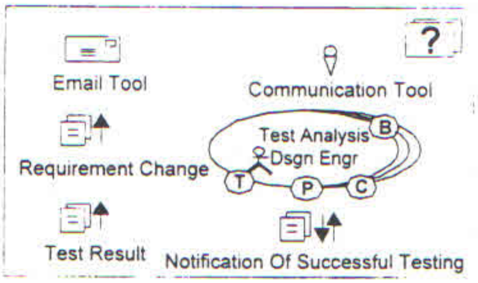

FIGURE 15. Workspace for a meeting activity node called Test Analysis

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.

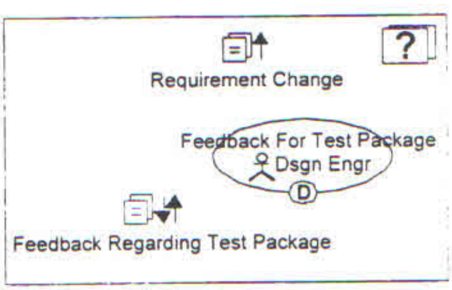

FIGURE 16. Workspace for an activity node called Feedback for Test Package

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.

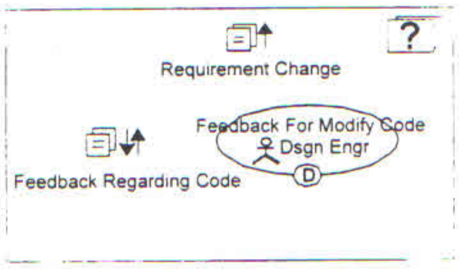

FIGURE 17. Workspace for an activity node called Feedback for Modify Code

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.

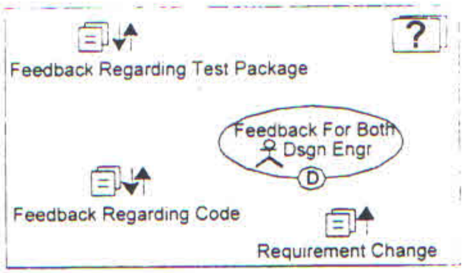

FIGURE 18. Workspace for an activity node called Feedback for Both

LESSON LEARNED

In providing the solution to the ISPW-6 problem, a number of lessons can be learned in terms of the effectiveness of representations, language expressiveness, and modularity as well as scalability of VRPML. These lessons are summarised below:

Effectiveness of representations: Because VRPML notations are purely graphical, it seems straightforward to make sense out the VRPML representations. In fact, to some extent, the general structure of VRPML 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 problem. 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 flow signals can compete to enable a particular node. This problem is currently being addressed in the notation by exploiting the resource exception handling mechanism.

Modularity and Scalability: VRPML is highly modular in the sense that activities are model 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 twelve different types of activities nodes (including macro expansion) as well as twelve corresponding workspaces. One point to note is that this limitation is inherent in any graph based visual language.

DISCUSSION

Throughout section 2, 3, 4 and 5, we have presented a step by step solution to the ISPW-6 problem expressed in VRPML together some of the preliminary lesson learned. It can be seen that our 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, Estublier 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

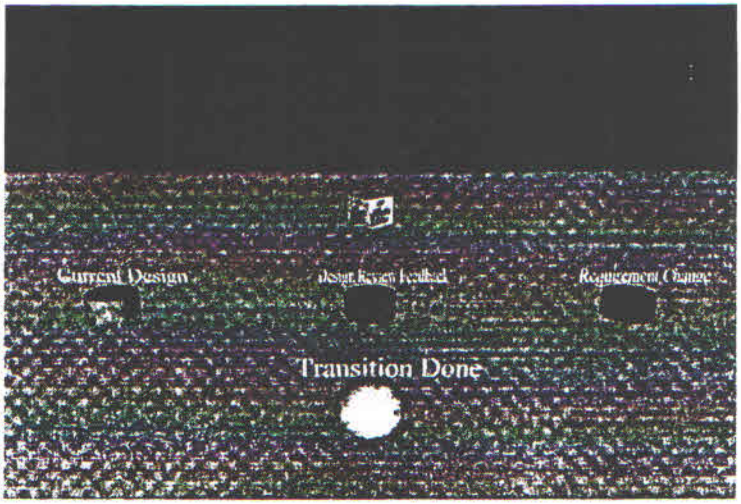

FIGURE 19. Workspace snapshot for Modify Design seen during enactment

an illustration, Figure 19 depicts a snapshot from the enactment of the ISPW-6 for an activity node called Modify Design.

- 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 other PMLs such as Dynamite (Heiman, Joeris et al. 1996) via on the fly process evolution based on graph rewriting and SLANG (Bandinelli, Fuggetta et al. 1994) via process evolution based on reflection. While both approaches have successfully addressed the support for dynamic allocation of resources, they suffer from one major limitation. Because both approaches relies on process evolution, 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.
- 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 reams) 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 introduce 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).

REFERENCES

Bandinelli, S., A. Fuggetta, et al. (1994). SPADE: An Environment for Software Process Analysis, Design and Enactment. *Software Process Modelling and Technology*. B. Nuseibeh. Taunton, England, Research Studies Press: 223-247.

Belkhatir, N., J. Estublier, et al. (1994). ADELE-TEMPO: An Environment to Support Process Modelling and Enaction. *Software Process Modelling and Technology*. B. Nuseibeh. Taunton, England, Research Studies Press: 187-222.

Canals, G., N. Boudjlida, et al. (1994). ALF: A Framework for Building Process-Centred Software Engineering Environments. *Software Process Modelling and Technology*. 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*. D. Wastell. Berlin-Heidelberg, Lecture Notes in Computer Science Volume 1500, Springer: 27-52.

Dami, S., J. Estublier, et al. (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., G. Joeris, et al. (1996). *DYNAMITE: Dynamic Task Nets for Software Process Management*. Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, IEEE Computer Prress.

Huff, K. E. (1996). Software Process Modeling. *Trends in Software Process*. A. Wolf, John Wiley & Sons: 1-24.

Kellner, M. I., P. H. Feiler, et al. (1990). *Software Process Modeling Example Problem*. Proceedings of the 6th International Software Process Workshop, Hakodate, Hokkaido, Japan, IEEE Computer Society Press.

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. (2001). "Process Modeling Languages: A Literature Review." *Malaysia Journal of Computer Science* **14**(2): 26-37.

Zamli, K. Z. and P. A. Lee (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 P. A. Lee (June 2001). *Taxonomy of Process Modeling Languages*. Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications, IEEE Computer Society Press.

MAKLUMAT PENGARANG
Kamal Z. Zamli
Pusat Pengajian Elektrik Elektronik,
Universiti Sains Malaysia,
14300 Nibong Tebal
Pulau Pinang
kamal_zamli@yahoo.com