Malaysian Journal of Computer Science, Vol. 18 No. 1, June 2005, pp. 57-69

THE DESIGN AND IMPLEMENTATION OF THE VRPML SUPPORT ENVIRONMENTS

Kamal Zuhairi Zamli and Nor Ashidi Mat Isa
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@eng.usm.my
ashidi@eng.usm.my

Norazlina Khamis
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 modelled and enacted by a Process Modelling Language (PML) and its process support system (called the Process Centered Environments i.e. PSEE).

This paper discusses the design of the process support environment for a new process modelling language, called the Virtual Reality Process Modelling Language (VRPML). It identifies the main components of the VRPML process support environments as well as summarises the current implementation prototypes. It highlights lessons learned and offers insights into the design of next-generation PMLs and PSEEs.

Keywords: ***Process Modelling Languages (PML), Process Centered Software Engineering Environments (PSEE), Software Engineering, Virtual Reality Process Modelling Language (VRPML)***

1.0 INTRODUCTION

Engineering as a discipline relates to the creative application of mathematical and scientific principles to devise and implement solutions to problems 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. For example, consider an assembly line 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 necessary to emphasize on the processes by which the software is produced. In software engineering, these processes are 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 modelled and enacted by a process modelling language (PML) and its process support system called the Process Centered Environments (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 assume 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 Modelling Language (VRPML) [13-19]. We also discuss how the environment can be used to realise 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 organised 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. The paper ends with a brief discussion on further work to be done.

2.0 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 customised for specific projects from a generic model. To illustrate this, Fig. 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 include: 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, pre-conditions and post-conditions, which must be respected by the software engineers to complete the task.

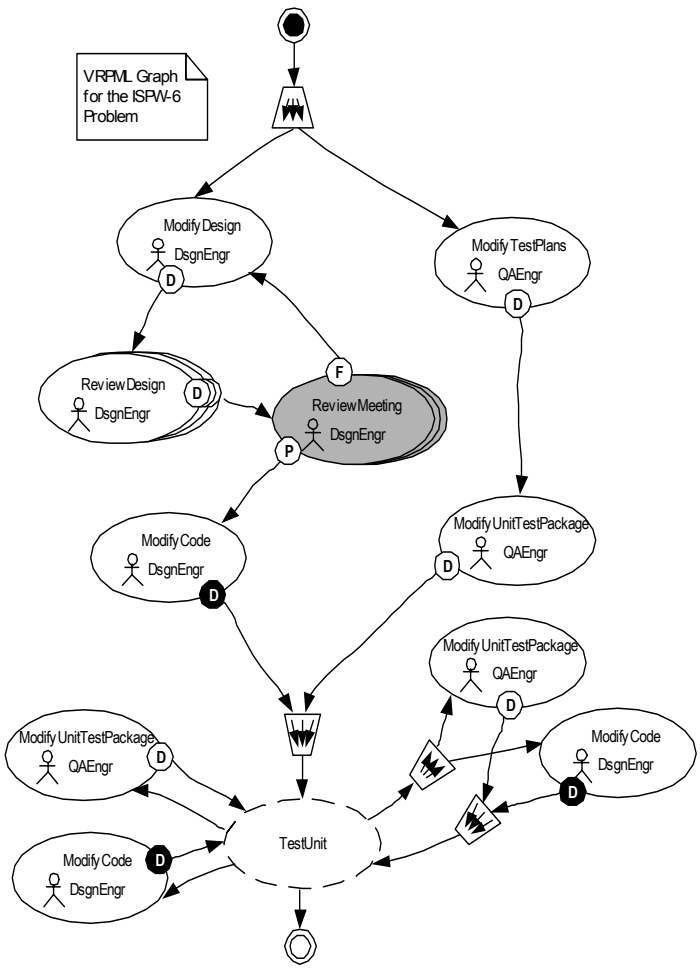

Fig. 1: Excerpt from the VRPML Graph for the ISPW-6 Problem

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 Fig. 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 Fig. 2.

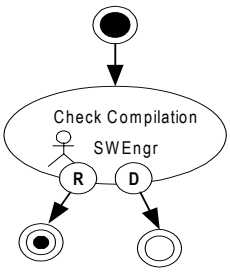

Fig. 2: Sub-graph for Decomposable Transition labeled D in Modify Code

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, with reference to Fig. 1, Test Unit is a macro node. The macro expansion of Test Unit is given in Fig. 3.

For every activity node, VRPML provides a separate *workspace*, the concept borrowed from ADELE-TEMPO [1], APEL [3] and MERLIN [7]. Fig. 4 depicts the sample workspace for the activity node called Review Meeting in Fig. 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].

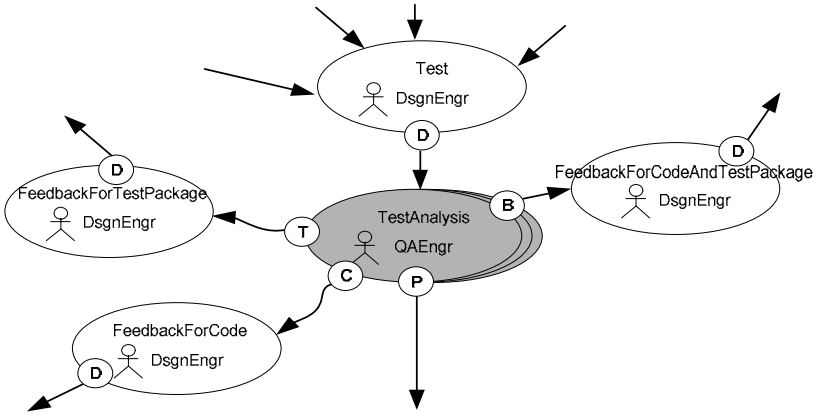

Fig. 3: Macro Expansion for Test Unit in Fig. 1

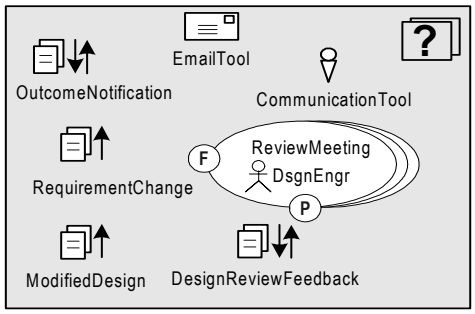

Fig. 4: Sample Workspace for Activity Node Review Meeting from Fig. 1

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 be allocated either 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, the 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 the 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 is complete when the software engineer selects one of the possible transitions (e.g. passed, failed, done, or aborted).

3.0 THE VRPML SUPPORT ENVIRONMENT

In order to implement VRPML, a number of components for the support environment are identified. These components and their interactions are shown in Fig. 5 (from next page).

The main components of the complete VRPML support environment and their functions 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 - retrieves 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

Each component is described accordingly in the next section.

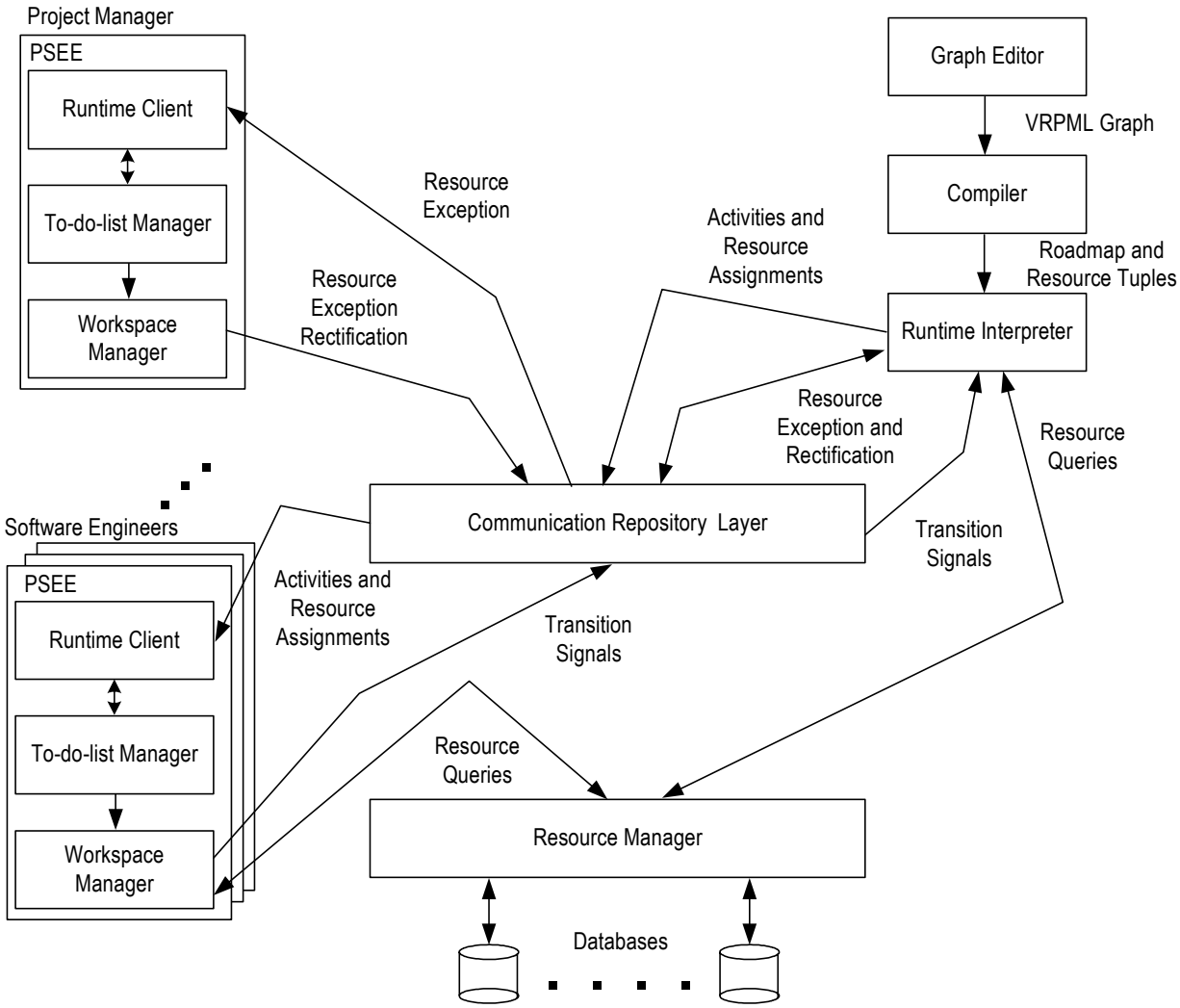

Fig. 5: The VRPML Support Environment

Graph Editor

Much like a programmer's editor in a textual programming language, the graph editor allows VRPML graphs to be drawn and changed. It also allows 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 are drawn manually.

Compiler

A compiler performs syntax checking and translates 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 is performed manually. However, in order to ensure that a compiler could be written, research into the information needed for enactment is 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 needs to be preserved. One solution shown in Fig. 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*.

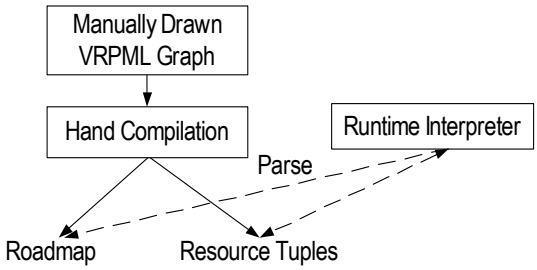

Fig. 6: Compilation of the VRPML Graph

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

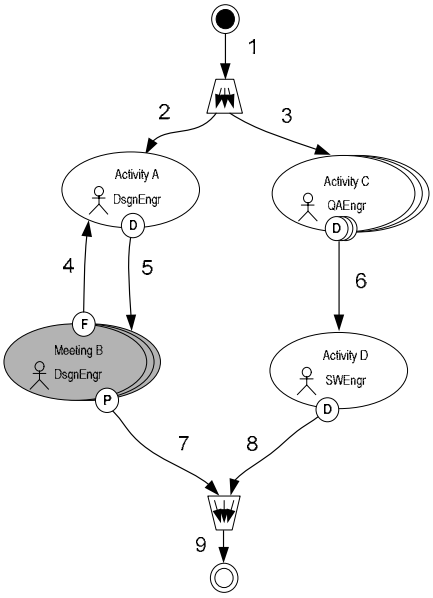

Fig. 7: Simple VRPML Graph

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])

"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 Fig. 8 is defined below:

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.*

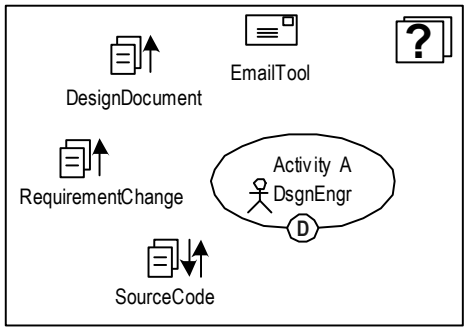

Fig. 8: Example Workspace

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, the enactment can easily be achieved. In fact, by adopting the Linda tuple space as the communication repository layer, the 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 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 set up 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:

- to create a to-do-list for a particular software engineer
- to interact with the runtime client upon request to retrieve the assigned activities from the communication repository layer
- to provide an internal to-do-list queue to store the assigned activities received from the runtime client
- to 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
- to forward an activity and its resource allocations to the workspace manager (described below) when the activity is selected to be undertaken
- to provide an interface to quit the to-do-list.

As an illustration, Fig. 9 shows a sample snapshot of the to-do-list GUI for a software engineer named 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. It must be stressed that implementation efforts concentrates on the functionality of the rather than the aesthetics of the 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 undertake.

At a first glance, generating and maintaining each activity's workspace according to its work context seem 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.

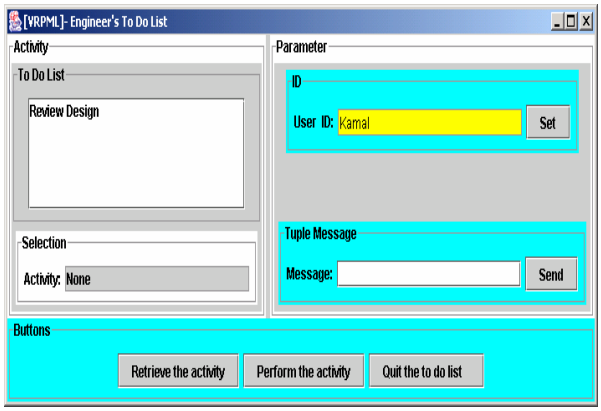

Fig. 9: The To-Do-List Graphical User Interface

As far as the implementation is concerned, the chosen approach to generate the workspace for each activity in a virtual environment is 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.

Fig. 10 depicts the possible translation of the workspace and representation in a virtual environment for an activity called Modify Design. Utilising 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.

Fig. 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 (Fig. 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 discussion.

4.0 DISCUSSION

The main components of the VRPML support environment have been identified in this paper. As far as implementation is concerned, a working prototype has been developed [17}. Fig. 11 summarised the overall class diagrams for the VRPML support environments based on the Unified Modelling Language (UML) notation.

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 provide 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 the 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 earlier (i.e. in terms of whether or not the environment is suitable for implementing the novel features of VRPML), an experiment which demonstrated enactment in a distributed environment by utilising the ISPW-6 problem has been successfully conducted. The complete discussion on the experiment is, however, beyond the scope of this paper and has been presented elsewhere [16, 17].

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

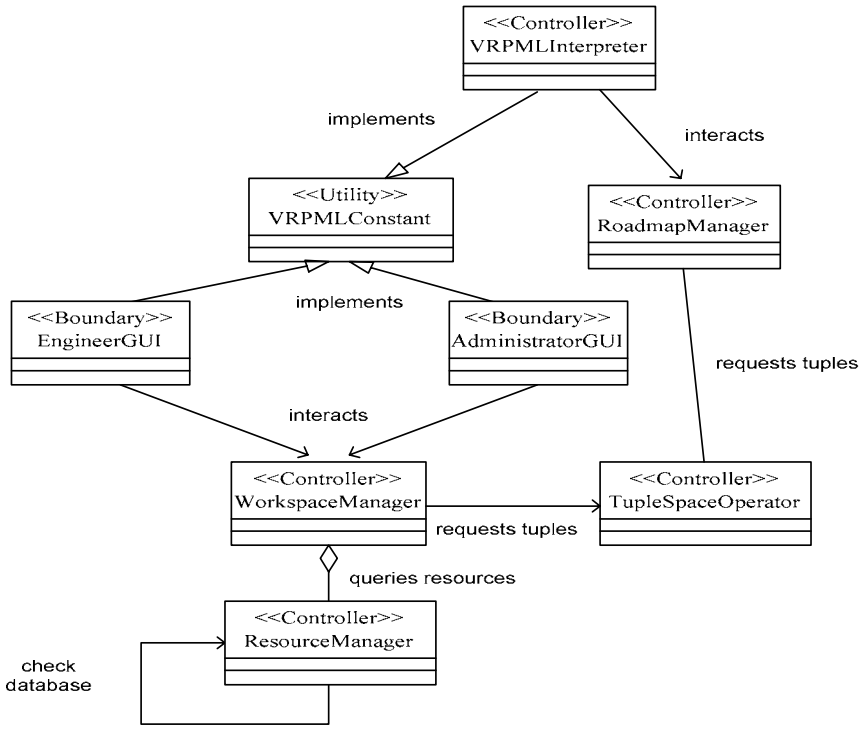

Fig. 11: The UML Class Diagram for the VRPML Support Environment

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.

5.0 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 an 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 visualisation 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 [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 (Fig. 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.

6.0 CONCLUSION

Because of the potential benefits in terms of being able to provide automation, guidance and enforcement of software engineering practices and policies, through modelling 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". 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 and 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". 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, pp. 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, pp. 217-224.

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, University of 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 of 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 specialises 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. She teaches undergraduate computer science courses such as software engineering, database, operating system, software quality and software requirements engineering.