The Applicability of VRPML for Supporting Distributed Software Engineering Teams

Kamal Zuhairi Zamli, and 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
E-mail: eekamal@eng.usm.my
E-mail: ashidi@eng.usm.my, ashidi75@yahoo.co.uk

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.

Keywords: 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 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 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.
Software processes are specified in VRPML as graphs, by interconnecting nodes from top to bottom using arcs that carry runtime 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 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](image)

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](image)

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](image)
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.

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
The Applicability of VRPML for Supporting Distributed Software Engineering Teams

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.

![Partial VRPML graph for the ISPW-6 problem](image)

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 QAEEngr) 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 7. Jon’s To-Do-List](image)

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 8.

![Figure 8. Workspace for Modify Design](image)

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 9 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 9. Resource Allocations for Modify Test Plans](image)

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 10. 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
The Applicability of VRPML for Supporting Distributed Software Engineering Teams

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 10. 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 11.

Figure 11. 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 8) 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 terms 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.

References


