Addressing Race Condition Problem in a Graph Based Visual Language

Kamal Z. Zamli and Nor Ashidi Mat Isa

[1]School of Electrical & Electronic Engineering,
Universiti Sains Malaysia, Engineering Campus, 14300 Nibong Tebal, Penang Malaysia

Summary

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

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

1. Introduction

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

While a visual programming language may not be able to provide a silver bullet to solve every problem related to engineering a software system, a carefully chosen level of abstractions (e.g. by working at the same level of abstraction as the problem domain) coupled with easy to understand notations may help alleviate the low-level complexities offered by the textual counterpart. Motivated by the abovementioned arguments, much research has been undertaken on visual languages over the last 20 years. Of interest to us is the graph based visual language [9][10][11] .

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

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

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

The Granite and the Limestone are still remained the main rocks types used for the aggregates production. The aggregates are referred to these crush rocks which normally used in road and other construction purposes [1]. It is a fast growing and emerging industry as one of the most demanding with a lot of expectations to fulfill the needs and the requirement for various industry and domestic purposes.

2. Overview of VRPML

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

Manuscript received July 5, 2007
Manuscript revised July 25, 2007

followed by software engineers to pursue the goals of software engineering.

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

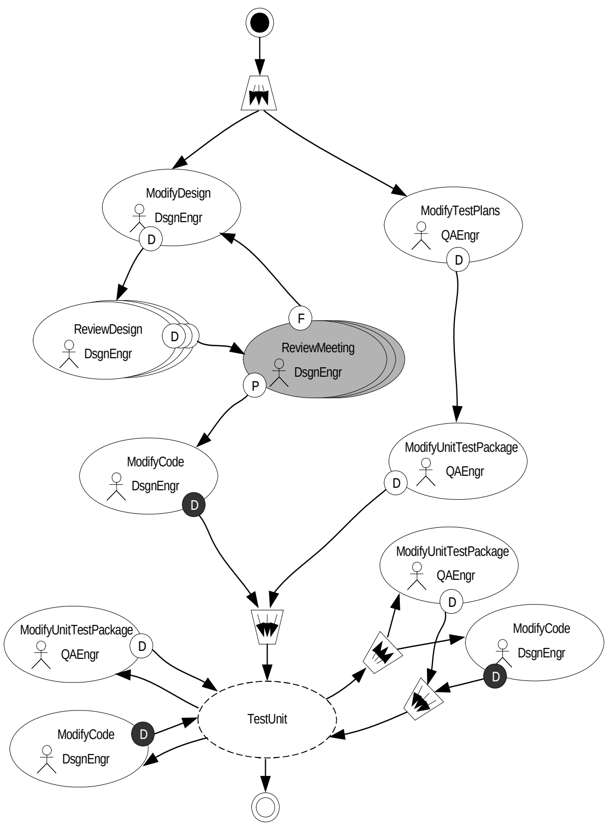

Fig. 1 Main VRPML Graph for the ISPW-6 Problem

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

As depicted in Figure 1, VRPML supports many different kinds of activity nodes. They include: *general-purpose activity nodes* (shown as individual small ovals with stick figures); *multi-instance activity nodes* (shown as overlapping small ovals with stick figures); and *meeting activity node* (shown as small and shaded overlapping ovals with stick figures). Both multi-instance activity nodes and meeting activity nodes have associated depths, indicating the actual number of engineers involved (and also the number of identical activities in the case of multi-instance activity). Also, a set of VRPML nodes can be grouped together using a *macro node* (shown as dotted line ovals) to improve the graph readability.

The firing of activity nodes is controlled by the arrival of a necessary control-flow signal. Control-flow signals may be generated from *transitions* (shown as small white circles with a capital letter attached to an activity node) or *decomposable transitions* (shown as small black circles with a capital letter attached to an activity node). However, the initial control-flow signal must always be generated from a *start node* (shown as a white circle enclosing a black circle). A *stop node* (shown as a white circle enclosing another white circle) does not generate any control-flow signal. In VRPML, activity nodes can also be enacted in parallel using combinations of language elements called *merger* and *replicator* nodes (shown as trapezoidal boxes with arrows inside).

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

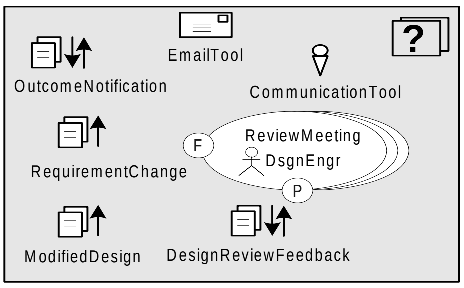

Fig. 2 Sample Workspace for Activity Node Review Meeting from Figure 1

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

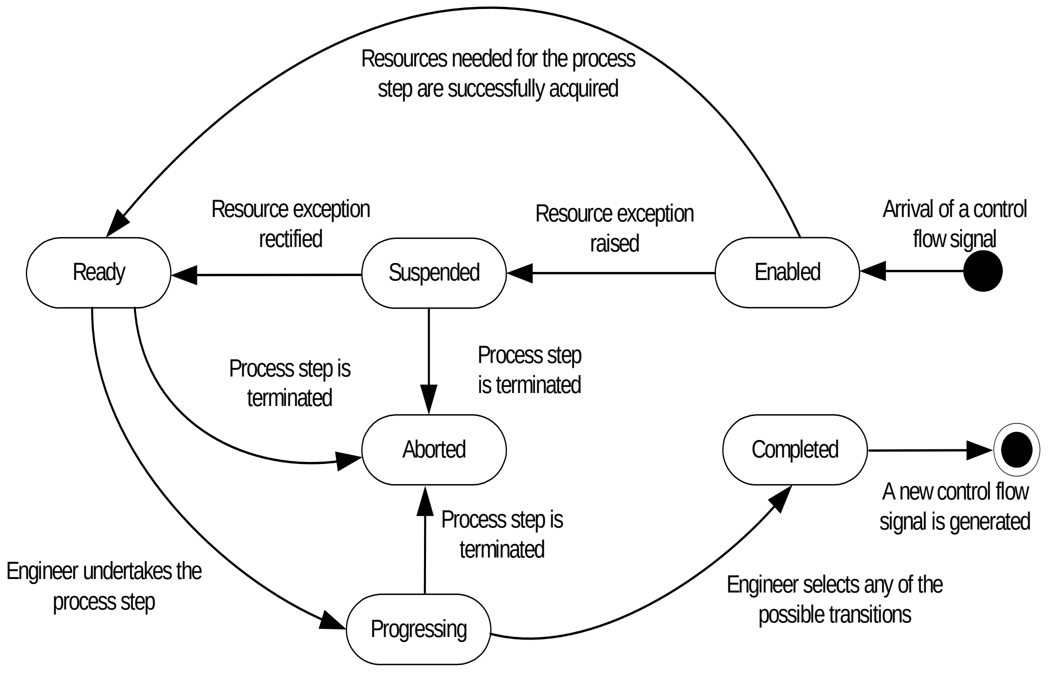
Resources needed for the process step are successfully acquired
Resource exception rectified
Resource exception raised
Arrival of a control flow signal
Ready
Suspended
Enabled
Process step is terminated
Process step is terminated
Aborted
Completed
A new control flow signal is generated
Process step is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

Fig. 3 VRPML Enactment Model

Upon the arrival of the control-flow signal, an activity node will be enabled (see Figure 3). 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. 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. Solution to the ISPW-6 Problem

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

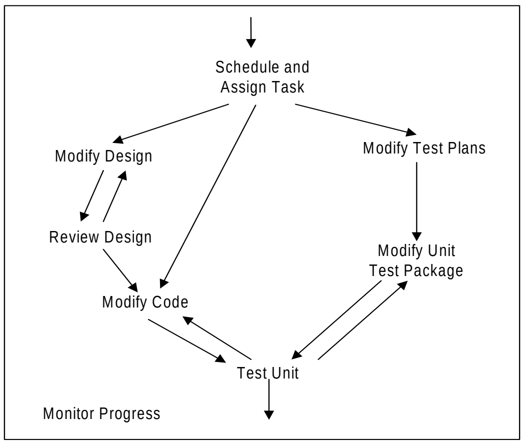

Fig. 4 Flow of Tasks in the ISPW-6 Problem

Table 1: ISPW-6 Activities, Inputs and Outputs

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

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

Two aspects of the VRPML solution shown in Figure 1 are worth highlighting. The first aspect relates to macro nodes. Figure 5 below illustrates the macro expansion for Test Unit. As seen in the figure, macro nodes serve as the modularization and abstraction facility for VRPML, apart from improving the graph's readability.

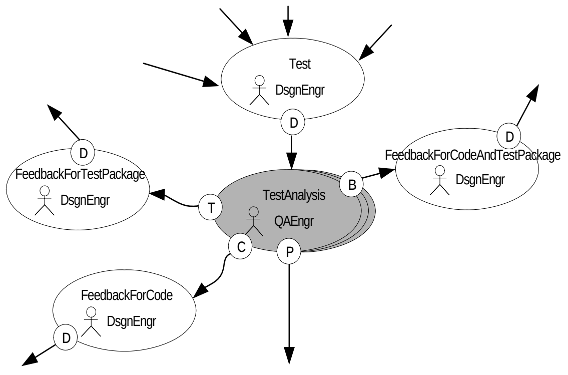

Fig. 5 Macro Expansion for Test Unit in Figure

The second aspect also relates to modularization and abstraction, that is, through the decomposable transitions. In VRPML, decomposable transitions permit the specification of the activity to check whether or not the pre-conditions of the parent activity are satisfied before allowing the control-flow signal to be generated. Referring to Figure 1, an example of the decomposable transition (labeled D) can be seen attached to Modify Code. The sub-graph representing that transition is shown in Figure 6 below.

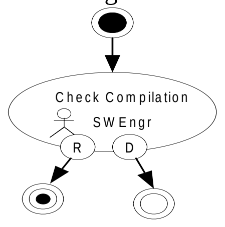

Fig. 6 Sub-graph for Decomposable Transition labeled D in Modify Code

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.

4. Lessons Learned About VRPML

Upon generating the ISPW-6 solution, we observe some limitation in the VRPML's firing rule particularly relating to race condition, whereby two or more control-flow signals compete to enable a particular activity node. Figure 7 below illustrates the situation

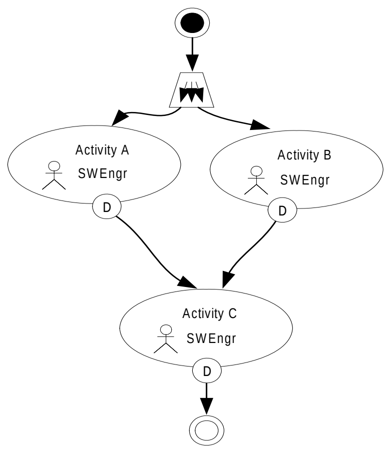

Fig. 7 The Problem of Race Conditions

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

To address this issue, we propose to amend the VRPML syntax and semantics. Instead of permitting multiple incoming arcs, the syntax of an activity node could be changed to allow only a single incoming arc to connect to it. Therefore, referring to Figure 7, the two arcs connections from the done transitions of activity A and B to activity C would be syntactically incorrect. In this way, the possibilities for race conditions would be eliminated.

This proposed change of syntax to activity nodes raises an issue relating to iteration. Disabling multiple incoming arcs would not permit iterations, hence, limiting the expressiveness of the VRPML notation. For example, the VRPML graph in Figure 8 below would give rise to a syntax error during compilation due to more that one arc connection to activity P (i.e. from the start node and the transition R of activity Q).

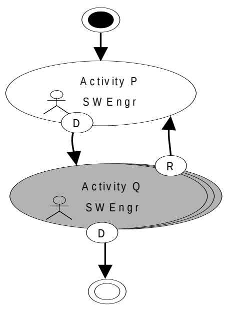

Fig. 8 VRPML Feedback Example

As a solution to support iteration in a VRPML graph, a new decomposable cyclic node could be introduced. Figure 9 depicts the proposed notation for the decomposable cyclic node.

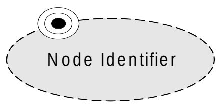

Fig. 9 The Proposed Decomposable Cyclic Node Notation

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

An example usage of the decomposable cyclic node called "Activity P and Q" alongside its decomposition is demonstrated below to express the feedback loop for the VRPML graph given earlier (see Figure 8).

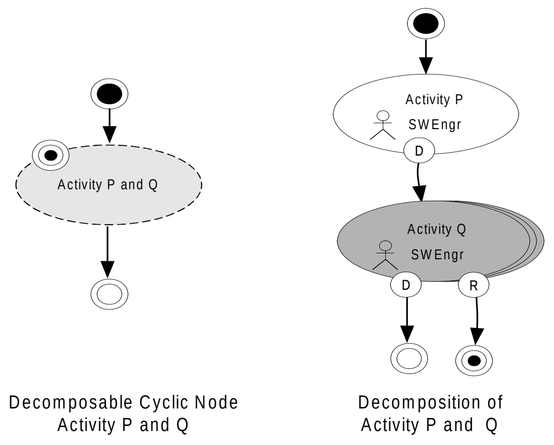
Activity P and Q
Activity P
SWEngr
D
Activity Q
SWEngr
D
R
Decomposable Cyclic Node Activity P and Q
Decomposition of Activity P and Q

Fig. 10 Example Usage of the Proposed Decomposable Cyclic Node

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

Having considered the proposed changes to the VRPML syntax, Figure 11 below highlights the new partial VRPML solution to the ISPW-6 problem.

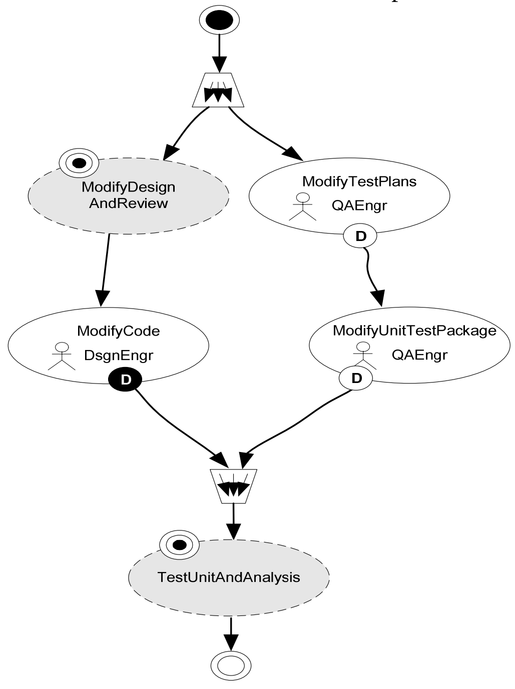

Fig. 11 Using Decomposable Cyclic Node as part of the VRPML Solution to the ISPW-6 Problem

Two decomposable cyclic nodes can be used:

(i) Modify Design and Review

(ii) Test Unit and Analysis

Because the decomposition for Modify Design and Review is relatively straightforward and similar to the case discussed earlier in Figure 8, it will not be developed further. Instead, only the decomposition for Test Unit and Analysis will be shown in Figure 10. Here, the decomposition of Test Unit and Analysis has captured the feedback requirement as stipulated by the ISPW-6 problem. Apart from addressing the possible race condition problems, it can be observed that the VRPML solution in Figure 11 and 12 can be more compact as compared to the ones given earlier. One reason is that the VRPML notation now becomes acyclic.

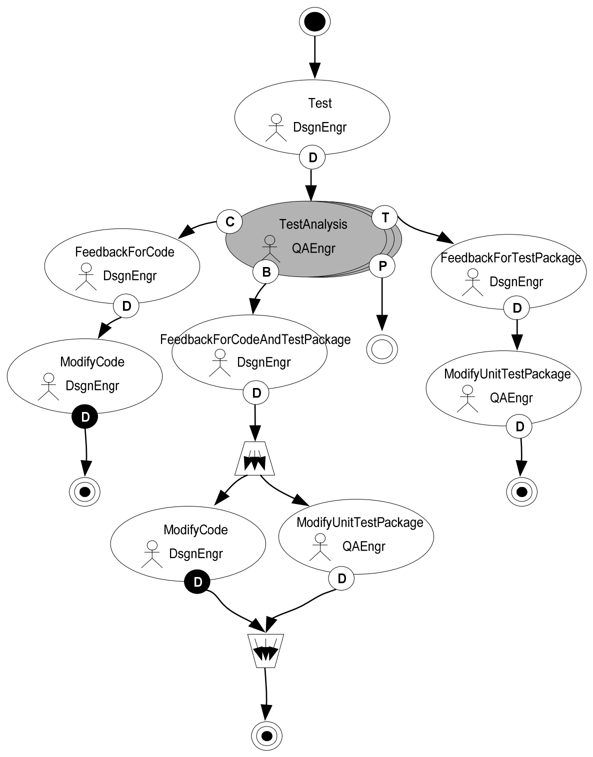

Fig. 12 Decomposition of Test Unit and Analysis

5. Conclusion

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

Acknowledgement

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

References

1. W.B. Ackerman, Data Flow Languages. In *IEEE Computer*, pp 15-23, February 1982
2. T. Agerwala, and Arvind, 1982. Data Flow Languages. *In IEEE Computer*, pp. 10-13, February 1982.
3. J.C. Doppke, D. Heimbigner, and A.L. Wolf. “Software Process Modeling and Execution within Virtual Environments”. *ACM Transactions on Software Engineering and Methodology, 7* (1), January 1998. pp. 1-40.
4 M.I. Kellner, P.H. Feiler, A. Finkelstein, T. Katayama, L.J. Osterweil, M.H. Penedo, and H.D. Rombach. “Software Process Modeling Example Problem”. In *Proc. of the 6th Intl. Software Process Workshop*, Hakodate, Hokkaido, Japan, October 1990. IEEE CS Press.
5. K.N. Whitley, “Visual Programming Languages and the Empirical Evidence For and Against”. *Journal of Visual Language and Computing* 8 (1), January 1997, pp. 109-142.
6. A. Wise. “Little JIL 1.0 Language Report - Technical Report 98-24”, Dept. of Computer Science, Univ. of Massachusetts at Amherst, April 1998.
7. K.Z. Zamli and P.A. Lee. “Exploiting a Virtual Environment in a Visual PML”. In *Proc. of the 4th Intl. Conf. on Product Focused Software Process Improvements*, LNCS 2559, pp. 49-62, Rovaniemi, Finland, 2002, Springer.
8. K.Z. Zamli. “Supporting Software Processes for Distributed Software Engineering Teams”, School of Computing Science, Univ. of Newcastle upon Tyne, PhD Thesis (Oct 2003).
9. K.Z. Zamli, and N.A. Mat Isa, “The Computational Model for a Flow-based Visual Language”, in *Proc. of the AIDIS Intl. Conf. in Applied Computing 2005*, Algarve, Portugal, pp.217-224 , Feb 22-25, 2005.
10. K.Z. Zamli, N.A. Mat Isa, and N. Khamis, "The Design and Implementation of the VRPML Support Environment", *Malaysian Journal of Computer Science* 18 (1), June 2005, pp. 57-69.

[11] K.Z. Zamli, N.A. Mat Isa, N. Khamis, “Implementing Executable Graph Based Visual Language in a Distributed Environment”, in *Proc. of the IEEE Intl. Conf. on Computing and Informatics*, Kuala Lumpur, June 2006.

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

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