Implementing Executable Graph Based Visual Language in a Distributed Environment

1Kamal Z. Zamli, 2Nor Ashidi Mat Isa, and 3Norazlina Khamis
1,2School of Electrical and Electronic
University Science Malaysia Engineering Campus
14300 Nibong Tebal, Penang, Malaysia
3Faculty of Computer Science and Information Technology
University Malaya
50603 Lembah Pantai, Kuala Lumpur, Malaysia

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

I. INTRODUCTION

Visual programming languages have been around for quite some time now. The basic idea behind a visual programming language is that computer graphics (e.g. graphs consisting of icons, nodes, and arcs) are used instead of a textual representation. In fact, the central argument for a visual programming language is based on an observation that picture is better than text (i.e. a picture is worth a thousand words [14]).

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

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

This paper is organized as follows. Section 2 gives an overview of graph based visual languages. Section 3 introduces the syntax and semantics of VRPML. Section 4 identifies the possible runtime components supporting execution (or enactment) of the VRPML graph. Section 5 outlines our prototype implementation. Finally, section 6 presents the conclusions of the paper.

II. GRAPH BASED VISUAL LANGUAGE

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

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

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

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

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

With a data-driven firing rule, an arc is used as a supply route to transmit data from the source node to the destination node. A destination node is executed as soon as data is available on all input arcs. With a demand-driven firing rule, an arc is used as a demand route to request data from the source node. A source node is executed only if there is a demand for its result. For either firing rule, arcs are conduits for data. In turn, data on an arc is consumed by the executing node to perform its computation (although some
According to Agerwala and Arvind [2], the data-flow based model can be distinguished from the control-flow based model in that it has neither a globally addressable memory nor a single instruction counter. As the data-flow based model possesses no global memory, the only data available to a node for its operation is that from its inputs. In addition, because of the lack of any shared data amongst nodes, there can be no side effects (one node interfering with other node’s data, potentially causing unexpected results).

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

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

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

III. INTRODUCING VRPML

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

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

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

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

Fig. 1. The VRPML Graph

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

The firing of activity nodes is controlled by the arrival of a control flow signal. In VRPML, an initial control flow signal is always 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.

![Fig. 2. Sub-graph for Decomposable Transition labeled D in Modify Code](image1)

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

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

![Fig. 3. Macro Expansion for Test Unit in Figure 1](image2)

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

![Fig. 4. Sample Workspace for Activity Node Review Meeting from Figure 1](image3)

**IV. VRPML RUNTIME COMPONENTS**

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

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

- **Graph Editor** – allows the VRPML graphs to be specified.
- **Compiler** – compiles the VRPML graphs into an immediate format for enactment.
- **Server/Client Interpreter daemon** – interprets the compiled VRPML graph.
- **Communication Layer** – act as mailboxes to coordinate the control flow signal for enabling execution in a distributed environment

The most important components which warrant further discussion are: the compiler; the client/server interpreter daemon; and the communication layer. A compiler performs syntax checking and translates a VRPML graph into an intermediate format known to the runtime interpreter. An important consideration for compiling VRPML is the information stored in the intermediate format. Clearly, the topology of the VRPML graph in terms of the ordering and sequencing of activities together with their resource assignments (if any) needs to be preserved.
A format for the roadmap and workspaces has been identified and used to facilitate the compilation of the VRPML graph, and hence permit execution. To illustrate this technique, Figure 6 shows an example graph to be compiled.

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

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

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

Workspaces must be generated for each activity defined in the graph. To illustrate the contents of a workspace, assume that the workspace definition for activity A shown in Figure 6 is defined below:
The workspace for activity A is generated as follows with keywords (shown in bold) used for clarity and human readability.

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

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

Having considered the compiler, the next component is the client/server interpreter daemon. Much of the functionality has already been implied in the earlier discussion. The full list of the client/server interpreter daemon’s functions is as follows:

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

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

V. IMPLEMENTATION PROTOTYPE

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

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

VI. CONCLUSION

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

ACKNOWLEDGMENT

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

REFERENCES