Visual Programming Language Design
Using software engineering and visual representation approaches

Eric Su

Supervisor: Gary Marsden

Department of Computer Science
University of Cape Town
Visual Programming Languages (VPLs) are programming languages that allow users to create programs by manipulating visual elements on a GUI rather than coding. In comparison to textual programming languages such as C or Java, VPLs are on an even higher-level as it must allow users with no prior knowledge of standard programming techniques to be able to create a functional software program. These types of users will be referred to in this paper as end-user programmers. In a study of Software Engineering for VPLs, Margaret Burnett (2001) suggested that Visual programming can be described as computer programming which uses more than one dimension to convey the semantics of a program [1]. Furthermore, she claims that multi-dimensionality is the essential difference between VPLs and strictly textual languages. However, prior approaches to VPL implementations have shown that this is not the case. There many are other factors that can show how VPLs and conventional programming languages differ. Most of which will be highlighted in this review.

A good motivation to create a VPL is to provide end-user programmers with a good visualization paradigm to map hardware components when using a graphical interface, which is why many VPLs have been applied in the field of electrical engineering, specifically embedded systems. However, the problem with VPLs is that they are difficult to design and they lack a definitive approach to developmental procedures such as data abstraction. Therefore, the high level programming capabilities required for VPLs to achieve seems more farfetched. This literature review attempts to demonstrate the methodologies and rationales used for visual representations as well as topics concerning current VPL design and software engineering approaches.

In a study done on interactive visual data abstraction, Burnett and Ambler (1994) suggested that visual programming approaches can be classified into three categories. The first approach is when a VPL is used for part of a programming task while the remaining is done by coding. The second approach is when a VPL is used for special-purpose problem domains. The third and final classification is the approach of using design techniques that are not fully declarative [2]. When designing a VPL, it is often useful to know which of these three categories the intended system will fit in. This is important because, in order to follow a software engineering design approach, the designer will need to set user requirements by first gathering all relevant information about the user environment, as well as the available software engineering tools which can be useful during the development process.
Much research has been done on how end-user programmers interact with programming languages. It has been discovered that although it is advantageous to empower end-user programmers with the tools necessary to create working programs, the quality of these programs are often of low standard. “End users’ programs are all too often turning out to be of too low quality for the purposes for which they were created.”[3] For this reason, much work has been done in trying to design VPLs that will accommodate software quality issues such as usability, reliability and efficiency for end-user programmers. This area of research is known as end-user software engineering. Since VPLs deal with graphical representations of programs rather than textual representations, we would need to make use of several visualization techniques during the design process. These visualization techniques include several different approaches to diagrammatic reasoning and meta-modelling.

In a research paper done by Howse, Molina, and Taylor, (2000) based on diagrammatic reasoning systems, they looked for ways to prove completeness by giving formal syntax and semantic inference rules to spider diagram systems. A similar research conducted on meta-modelling by Bottoni and Grau (2004) presents an approach in providing a formal basis for visual language classification. The use of meta-modelling is particularly useful when classifying VPLs in order to determine how the different languages express domain elements and relations graphically. Classification of a basic meta-model depends on the characteristics of significant special relations among identifiable elements. [5] Other meta-model classifications include connection based meta-models, multipartite graphs, and containment based meta-models. Connection based meta-models are usually represented by endpoints that are linked by polylines, and are among the most used meta-models, whereas multipartite and containment based meta-models are less common.

Sometimes, a visual language will have identifiable elements that act as containers and adjacent elements. This is known as a hybrid language. Hybrid languages are defined by the existence of several spatial relations simultaneously. In fact, most existing VPLs today are hybrid. For example, the use of labels attached to or contained inside other visual elements provides the user with additional information about the element. A good example of a hybrid visual language would be something similar to a chess board application where each square of the board is a container for the pieces as well as an adjacent element to other pieces.

Diagrammatic visual languages are languages which makes use of graphics to represent a program. A diagrammatic language such as UML requires form
methods of defining syntax and semantics in order to handle diagram notations properly. In a study done on Diagrammatic Reasoning, Bottoni, Meyer, and Parisi-Presicce (2001) suggested that diagrams can be represented as Multi-sets, also arguing that a purely grammatical approach is not suitable for more general interpretation tasks. [6] I agree with this argument as it would be easier for visual language compilers to parse diagrams using linear deduction rather than grammatically. In a similar study, Fish, Flower, Howse (2003) demonstrated how recursion can be used in an effective algorithm to read constraint diagrams. They also provided methodologies which were highly effective on how a reading tree can be constructed from a dependence graph generated by a constraint diagram.

Iconic visual languages are similar to diagrammatic languages as they also make use of graphics to represent programs. However, iconic representation forms another field of research as they differ fundamentally from diagrammatic visual languages. The approach to use when designing VPLs while using an iconic visual representation will involve displaying graphical elements as objects that interact with each other, and the lines which connect the objects will show how the objects interact. On the other hand, a diagrammatic visual language will have visual elements which represent the states with connected lines showing the flow of the program. An iconic visual language is also easier to develop than a diagrammatic one, as it generally includes defining icons and rules of the application environment. An older iconic visualizing framework, developed by Hirakawa, Tanaka, and Ichikawa (1990), shows how visual interaction can be achieved using representations of actual objects to establish the concept. What was interesting about this design in particular was that they did not represent functions as icons as it made the design of the icon images simpler and easier to comprehend. I believe this to be a highly successful feature as the simplicity of their system provided a good foundation for future developers who are interested in designing similar systems.

Another approach to designing VPL’s would involve visual representations of static pictorial sequences. A system which used a visual representation based on this approach was done by Howland, Robertson and Good (2006), where a VPL was designed to allow amateur users, particularly young people, to develop their own games. The visualization was represented through the use of script cards which contained static pictorial sequences. I believe this visualization approach was effective in this case because the simplicity of user interface allowed the end-users to express their own ideas in their own preferred order. Furthermore, an evaluation was done to test the effectiveness of the script card system based on this visualization
approach and it has provided very positive results.

Now that these visualization approaches have been considered, one would need to follow a development model to ensure the quality of the software developed. Some software engineering issues such as VPL usability testing and end-user program debugging are important to consider when designing a VPL. How would the VPL developer know if their system is successful in terms of usability? For user testing, a study which focused on user testing for development tools was conducted by Lawrance, Clarke, and Rothermel (2005). They addressed the problem of inadequate software testing methodologies and conducted an empirical study on software development testing. Unfortunately, I believe this testing model will not be effective for the testing of Visual Programming Languages. The reason is because the study was generally conducted based on code coverage instead of usability. Another interesting approach to VPL testing is through a simulator provided by the DEViL toolset. DEViL is a generator framework that supports the development of VPLs. In a study done on DEViL by Cramer and Kastens (2009), it was briefly mentioned that VPL testing is possible by comparing simulation results of the VPLs. However, the disadvantage of this approach is that the VPL which is to be tested will need to have been generated by the DEViL toolset in the first place. So it does not support the testing of existing VPLs which were developed using other design tools. Thus, this solution does not apply to my initial problem either. I believe that a better way to conduct usability testing for VPLs is through means of ethnology studies, which is not in my area of focus for this paper.

The next issue related to software engineering is the question of how the end-user programmer would debug their program in a visual programming environment. Two different methodologies were reviewed in this paper concerning this topic: The first method involves a collaborative pair of users and the other method focuses on preventing the error from occurring by limiting the interface area. The first method, suggested by Chintakovid, Wiedenbeck, and Grigoreanu (2006), proved to be effective when debugging code. However, it is not effective at all to use this method when only one programmer is available. Another disadvantage with using this method is that productivity is reduced when the other programmers could be working on something else. Thus this method would not be recommended for companies who employ software developers based on the hour. The second method seems more promising and consists of designing highly integrated tools to prevent programming errors (Ko, Andrew J. 2003). This method uses contextual inquiry to gather information on end-user programmers and takes advantage of opportunities
to prevent errors from occurring. The disadvantage of this approach is that an error prevention system would be difficult to integrate, and it takes a lot of time to gather enough user data before it can be implemented. I believe the second method is more effective because it will improve user experience of the programmer who does not know how to code.

The results of the readings have identified the fields of research which proved to be relevant in VPL design. To summarize, issues that motivate the development of a VPL can be classified into three categories, namely: a partial implementation approach, a special purpose approach and a non-declarative approach. Additionally, there are three ways of data abstraction which can be used for visualization representation. A diagrammatic visual approach requires formal methods of defining syntax and semantics, an iconic visual approach involves displaying graphical elements as objects that interact with each other, and a static pictorial approach which makes use of static images. The topics highlighted in software engineering concern issues of user interface testing and end-user program debugging. With regards to testing, it is recommended to perform ethnology studies rather than using conventional code based analysis. Finally, a solution was proposed for visual language debugging which involved preventing program errors from occurring in the first place. Thus, a promising research area for my field of study would be the identification of places in a VPL where error prevention would be most effective.
References


10. Lawrance, Joseph, Steven Clarke, Margaret Burnett, and Gregg Rothermel, How Well Do Professional Developers Test with Code Coverage Visualizations? An

