A Survey of Visual Programming and its Future in Concurrency

by: Rishidhar Reddy Bommu

Mentor: Ron Mak

CS 180H, Spring 2016

San Jose State University

Abstraction

How do we define the evolution of programming languages. We have built programming languages on top of other programming languages. The immensity of languages is overwhelming, but spawns the question on what is the next form of programming. In this paper, we will be overseeing the possible creation of visual programming languages and their potential to depict concepts in Concurrency. I will initially define certain key elements in order to create consistency throughout the paper and help define some of the jargon for non-technical readers. The paper will then discuss the motivation for the paper and the current market in Visual Programming Languages. After understanding the current market, the paper will discuss potential concepts of shared memory and message passing in visual programming languages. With the theoretical concepts in place, the paper will then show a potential blue print design of a Visual Programming Languages that uses a drag and drop system. Lastly, the paper will look at complexities and future thoughts about Visual Programming Languages.

Definitions

A computer is a generic machine that can be given instructions to carry out arithmetic and logical operations. Thus the computer can solve a diverse set of problems through the instructions it is given. A programming language is a set of instructions that a computer can interpret and compute.

However, it is key to note that a programming language can abstract the instructions it gives to a computer. Thus, we have programming languages built on top of others in order to abstract computer instructions. In computing, a visual programming language is any programming language that allows the user to code graphically through the manipulation of program elements. Concurrency on the other hand is defined as the property of a programming language to execute various command sequences simultaneously. An example of this would be a computer playing audio as well as video for the user. However, when there are multiple processors in a computer to execute tasks, we can consider the tasks to be completed in parallel. This concept can be abstracted as Multi-Threading. Multi-Threading is the ability of a program or operating system to run multiple processes concurrently and in parallel. When these processes are running in parallel, there are two ways that they can communicate with each other. They can either send messages to each other in the form of Message Passing or the other option is to have shared memory between the two processes. Both options have their set of advantages and disadvantages.

Introduction

I am attempting to find a solution to two core problems with software and computer science today. One is the issue of multi-threading. As our day and age reaches the physical limits of a single processor, multiple processors are becoming increasingly important. The other issue that is tackled is the issue of textual programming languages. Many people have been swayed away from programming due to the initial learning curve. This may be due to the interface that is constantly in place for people. I am talking about the interface of writing programs textually. As a solution, a graphical programming language might provide great insight into the issue. I wish to give programmers an alternative to a textual based programming language. The goal is to depict that Graphical Programming Languages are able to show concepts in Concurrency. In this way, we can model concepts of programming in a graphical programming language and prove its usability and scalability.

The Visual Programming Market

There are hundreds of Virtual(Graphical) programming languages in the market. Many that have diverse purposes other than compiling into a language. The definition of a virtual programming as mentioned before, doesn't require one to compile into binary software. Thus various graphical programs fall into this category as long as they have the ability to manipulate graphical elements that create a sequence of instructions.

The following are some categories and a popular example from each:

1.Making games- Game Maker is a popular virtual programming language in game making software. This category has several varieties and seems to have an ongoing community. The demand for a non-programming environment to make games seems to be on the rise. One core feature that I noticed in testing this software was that, It had a feature that allowed you to script, in case you wanted a little more complexity. This gives the realization that graphical programming languages can coexist with textual programming languages. Game Maker is a living example of the combined model of textual and graphical programming.

2. Automating tools- Automator- Automator is a program that is shipped with Mac OS X. It can build a script graphically. The goal of this program is to automatize daily work. Having a GUI to do this on the MAC OS X seems quite useful, however the use cases are very limited. The capabilities reach their limits based on the finite set of elements. Each element is a certain abstract instruction. The finite number of instructions seems to be a problem with many visual programming languages. Yet Automator had a decent library to have several useful graphical scripts to be built.

3. Education- Scratch- Scratch is an education tool that teaches children to program. Scratch has a game style GUI that lets users make animations and controls. The use of basic commands in a row causes the display to move, causing a immersive learning environment.

4. Media tools- Photoshop- Photoshop is a graphical image editing tool. Image editing has been around since the early 1990s. Although programming libraries have made great strides in this category such as Open CV, Photoshop takes dominance in photo-editing. Photoshop is a key example of a market where textual programming languages have minimal use compared to their graphical forerunner.

In conclusion, we are in an age where there is a plethora of examples in visual programming languages. However, none of these languages are written in order to create generic software. These visual programming languages have trouble modeling abstract concepts such as Concurrency. However, we are not far from concurrency becoming reality in Visual Programming Languages.

Message Passing versus Shared Memory Concepts

Many textual programming languages that have concurrency follow one of two patterns. These two patterns are functional and imperative. These two programming patterns usually have separate models of Concurrency. Functional Languages usually follow a protocol of Message Passing, while languages with imperative design follow Shared Memory models. Some languages implement both and some neither. However, this creates a certain division when interpreting the problem visually. How would one represent shared memory and Message Passing in a visual format?

This problem creates us to look at functional and imperative design a little closer in order to understand Shared Memory and Message Passing. Message Passing specifically sends messages between processes. By definition, a purely functional language, always has a function return the same thing given the same input. This isn't true in imperative design as the internal state may change. Thus purely functional programming languages follow message passing to keep functions pure. In terms of visual design, this can be modeled simply. The visual program would consist of placing specific elements that instruct information for other functions to begin, stop, or pass input. For example, we can pass information as the input and instruct two functions to start at once.

On the other hand, we could also visually think about shared memory. Shared memory at first is just an element that two processes share. In this stage, we can abstract mutexes in a visual programming language. Mutexes are locks that assure that certain memory isn't accessed at once by two processes. We can always keep shared memory safe and uncorrupted by abstracting this concept. Thus, we have a foundation of strong shared memory that can be used as an abstract element. From this stage, we can give instructions that route what processes share memory and specifically what type of memory. In conclusion, these are ways we can theoretically imagine shared memory and message passing in a visual programming language.

Potential Design

In terms of building a Visual Programming Language, I have built an application blueprint on JavaFX. The code is only a blueprint of a potential drag and drop system. It is a proof of concept model that depicts the potential for a Visual Programming Language. The design has several goals that define its design decisions.

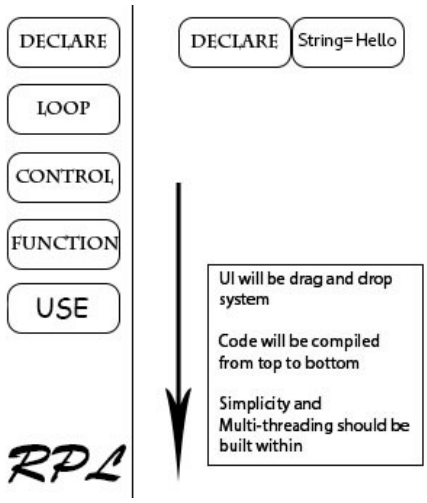


Illustration: Example Visual Progamming Language

The first design decision is simplicity and usability. The left side includes a Drag and drop button layout that can be dragged on to the drawing board on the right. The Majority of the screen is the drawing board. Dragging to the bottom right corner will delete any element in the drawing board. The left side has an unlimited amount of elements that spawn as the user drags and drops. This core foundation is placed in order to assure an elegant design that is easy to use, yet can scale quickly.

The second design decision is what each drag and drop element can do! The first element Declare is used to declare various primitive and non-primitive types. Having a uniform Declare structure makes Visual Programming simple in defining objects and using them. The second element is

Loop which is used to make for and while statements. The third element, Control, is used to have case and if statements. Loop and Control Elements are the basic elements of programming. These can provide the basic structure for the language. Lastly, the Use statement is used as a statement to Print various elements. This is a very basic structure to model the core elements in programming languages in a visual design.

The third design decision is based on multi-threading concepts. Each Programming board potentially represents a set of instructions to be executed. The final problem is to have a programming board that Declares Threads. This allows a very simple design in building multiple programming boards and executing them. Although, this pattern has many places of improvement, the goal is to show that a simple, feasible solution exists in building a visual programming language that caters concurrency.

Future Complexities, Related Work and Solutions

Visual Design has its core problem in abstraction. As we abstract various computer architecture concepts with Visual Programming Languages, we tend to make it harder to understand what is happening behind the covers. Visual Programming Languages have the potential to compile into binary, but may struggle in debugging as concepts are abstracted. We may not know exactly where a certain program made an error. The most specific example could be placed in threaded bugs. When simultaneous threads make a memory corruption error.

This problem has been sought out by Macquerie University, as they have a potential design solution for such a problem. There perspective is to use visual programming in a three-dimensional perspective. This allows for a greater deal of abstraction and utilizing visual design. The paper specifically defines these three dimensions as the processors, data flow, and control flow. However, this approach might lead to more complexity than less. It may be easier to pinpoint errors in this model, however, fixing the problem still comes with the same level of complexity.

This results in my initial statement that when designing visual programming languages, basic concepts need to be abstracted. For example, mutexes and semaphores should be abstracted and taken out of the control of the user. A user should be allowed to use shared memory, but the compiler

automatically assures that memory isn't corrupted. This isn't an impossible goal as java already offers several command to do this. As we abstract the concept in Visual Programming Languages, it is key that the language remain robust.

Conclusion and Personal Thoughts

Visual Programming has a bright future. With the current infrastructure in place, it is imminent that several visual languages might pop up to replace scripting or compiled languages. However, from my research, I can see these languages serve a specific niche more than become general purpose languages. Even with concurrent memory modeled in a visual design, it seems more persistent that these designs will be made for specific goals than for a generic one. However, this is an advantage than a problem in our current world. There is no universal programming language, and I do not expect one to be created. In terms of limitations, Visual Programming has its roots in being operating system specific. Optimization of a graphical language requires full utilization of the Operating System. This may create problems in cross platform designs for Visual Programming Languages. However, with the standardization of several graphical libraries, this may be a small splinter in the road. Another major road block that I see is usability. Although, visual design might be easier to program, it may provide a harder experience in understanding. Software in general is defined by its complexity management in the industry. If this language is to gain widespread use, than there needs to be a standard way to view the visual design of the entire project built upon a visual programming language. In Conclusion, Visual Programming has a prominent future as new technologies are introduced that make the barrier of visual design smaller.

References

Boshernitsan, Marat, and Michael Downes. "Visual Programming Languages: A Survey."*Computer Science Division, University of Berkeley* (2004). Web. 19 Apr. 2016.v

Browne, J.c., S.i. Hyder, J. Dongarra, K. Moore, and P. Newton. "Visual Programming and Debugging for Parallel Computing." *IEEE Parallel Distrib. Technol., Syst. Appl. IEEE Parallel & Distributed Technology: Systems & Applications* 3.1 (1995): 75-83. Web.

Burnett, I., M.j. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee. "Scaling up Visual Programming Languages." *Computer* 28.3 (1995): 45-54. Web.

Haeberli, Paul E. "ConMan." *ConMan: A Visual Programming Language for Interactive Graphics* (1988). Web.

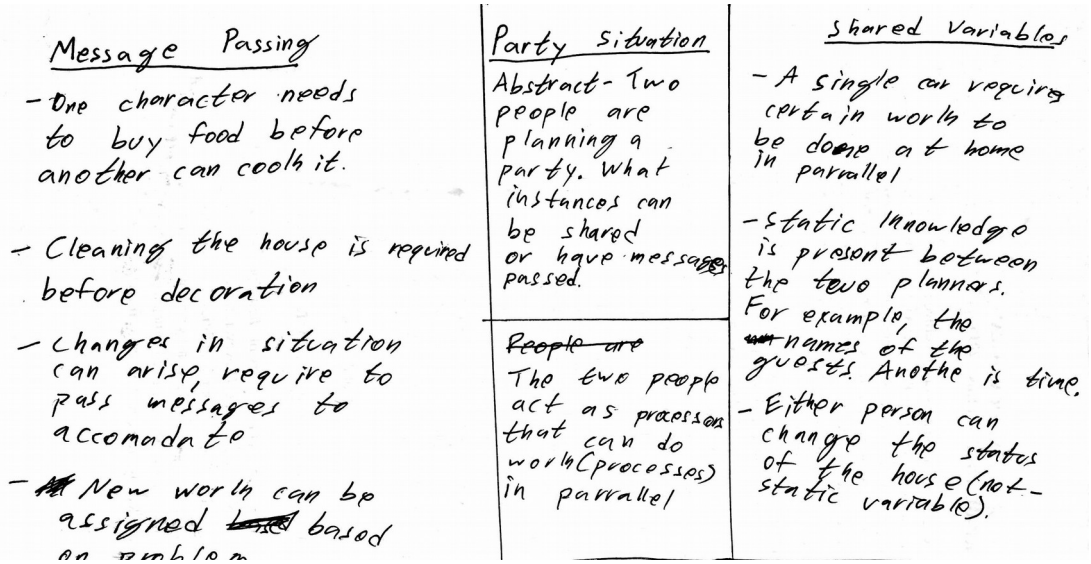
Newton, Peter, and Jack Dongarra. "Overview of VPE: A Visual Environment for Message-Passing." *Computer Science Department, University of Tennessee*. Web.

Shneiderman, Ben. "Direct Manipulation: A Step Beyond Programming Languages."*Computer* 16.8 (1983): 57-69. Web.

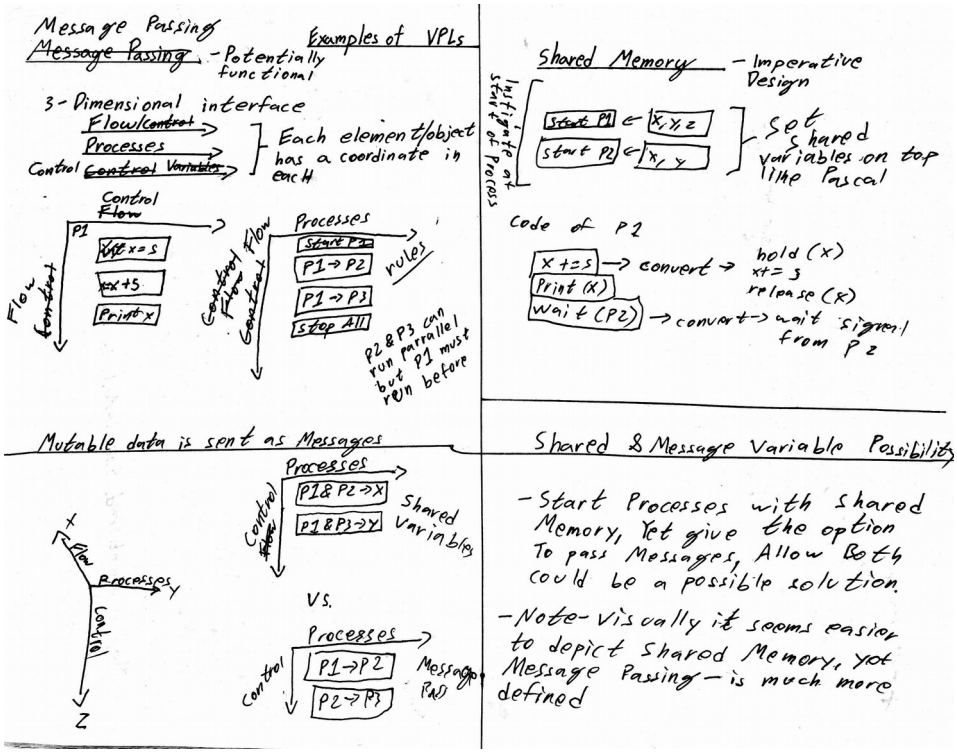
Stankovic, Nenad, and Kang Zhang. "Visual Programming For Message-Passing Systems." *Int. J. Soft. Eng. Knowl. Eng. International Journal of Software Engineering and Knowledge Engineering* 09.04 (1999): 397-423. Web.

Appendices

A. Abstraction of Party Planning- If a certain number of people are planning a party, how would one organize tasks. Examples of shared memory and shared variables



B. Message Passing and Shared Memory Sketches- Potential Designs for Message Passing and Shared Memory.



C. Blueprints for Drag and Drop Design Logic Java FX

```
<?import javafx.geometry.Insets?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<GridPane fx:controller="Controller"
          xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10">
</GridPane>
```

```
import javafx.application.Application;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
public class Main extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception{
        PrimaryScene primaryScene=new PrimaryScene(primaryStage);
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

```
import javafx.event.EventHandler;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.paint.Paint;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
/**
 * Created by rishi on 3/17/16.
 */
```

```
public class Statement extends Rectangle {
    private volatile double screenX = 0;
    private volatile double screenY = 0;
    private volatile double myX = 0;
    private volatile double myY = 0;
    private Pane pane;
    private boolean drawAgain;
    private int initialX;
    private int initialY;
    private Color color;
    private int height;
    private int width;

    public Statement(Color color, Pane pane,boolean drawAgain, int initialX, int initialY, int width, int height){
        super(width,height, color);
        this.color=color;
        this.drawAgain=drawAgain;
        this.initialX=initialX;
        this.initialY=initialY;
        this.height=height;
        this.width=width;
        this.pane=pane;
        this.setLayoutX(initialX);
        this.setLayoutY(initialY);
        this.setOnMouseClicked(new EventHandler<MouseEvent>() {
            @Override
            public void handle(MouseEvent event) {
                screenX = event.getX()-width/2;
                screenY = event.getY()-height/2;
                myX = getMyX();
                myY = getMyY();
            }
        });
```

```
    this.setOnMouseDragged(new EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent event) {
            double deltaX = event.getSceneX() - screenX;
            double deltaY = event.getSceneY() - screenY;
            setLocation(myX+deltaX, myY + deltaY);
        }
    });
    this.setOnMouseReleased(new EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent event) {
            if(isDrawAgain()){
                pane.getChildren().add(new Statement(color, pane,true,initialX,initialY,width,height));
                setDrawAgain(false);
            }
        }
    });
}
public double getMyX(){
    return  super.getX();
}
public double getMyY(){
    return  super.getY();
}
public void setLocation(double x, double y){
    this.setLayoutX(x-width/2);
    this.setLayoutY(y-height/2);
}
public boolean isDrawAgain() {
    return drawAgain;
}
public void setDrawAgain(boolean drawAgain) {
    this.drawAgain = drawAgain;
```

```
    }
}
```

```
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

/**
 * Created by rishi on 3/17/16.
 */
public class PrimaryScene {

    private Pane root;
    private Stage primaryStage;

    public PrimaryScene(Stage primaryStage)throws Exception{
        //Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));
        Pane root= new Pane();
        //code for boxes
        root.getChildren().add(new Statement(Color.BLUE,root,true,10,10,64,64));
        root.getChildren().add(new Statement(Color.GREEN,root,true,10,80,64,64));
        root.getChildren().add(new Statement(Color.RED,root,true,10,150,64,64));
        root.getChildren().add(new Statement(Color.ORANGE,root,true,10,220,64,64));

        this.root=root;
        this.primaryStage=primaryStage;
        primaryStage.setTitle("Hello World");
        Scene scene =new Scene(root, 1024, 1024);
        primaryStage.setScene(scene);
```

```
        primaryStage.show();
    }

    public Parent getRoot() {
        return root;
    }

    public Stage getPrimaryStage() {
        return primaryStage;
    }
}
```