FlowViz: a Visualization Toolkit to Support Visual Programming Language Development

Alexander Fiannaca
Mobile Accessibility Lab
University of Washington
fiannaca@cs.uw.edu

Sonya Alexandrova
Human-Centered Robotics Lab
University of Washington
sonyaa@cs.uw.edu

ABSTRACT

Developing a flow-based visual programming language (VPL) is a tedious task due to the fact that it requires writing a large amount of code for simply rendering programs to the screen on top of the code for managing the syntax of the language. While a large number of VPL's exist (e.g. Scratch [9]), there are no existing toolkits for the development of new VPL's; meaning that developers of these languages are re-writing boiler-plate rendering and syntax management code. To address this issue, we present FlowViz, a library which supports the development of VPL's by handling all of the low level visualization and language graph details by default. FlowViz handles the internals of managing the visual language graph, only requiring developers to specify the types of nodes and constraints between nodes in their VPL. Being targeted at flow-based VPL's, this specifically enables the development of end-user programming systems in which the "programming language" consists of a series of interconnected nodes.

Author Keywords

Visual Programming Language, Developer Toolkit

INTRODUCTION

Visual programming languages (VPL) have become very popular over the last two decades as a programming abstraction that can both make programming accessible to novice end users, and make programming faster by abstracting large amounts of code into simple visual tokens. Given these benefits, a massive number of VPL's have been developed in recent years with applications ranging from programming education (e.g. VPL's on Code.Org), to languages for domain experts in engineering (e.g. LabVIEW).

CS 512, June 8, 2015, Seattle, Washington.

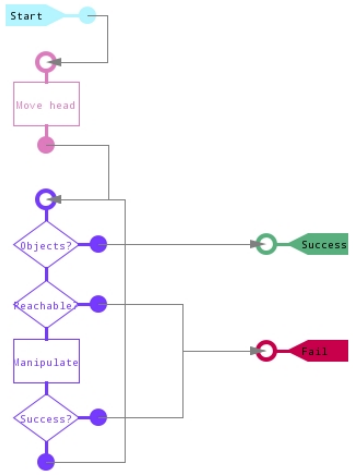

Figure 1. Original version of RoboFlow, an example of a VPL designed to allow end-users to program complex robots.

Of particular interest are VPL's which allow non-technical end users to program complex systems. An example of this type of VPL can be seen in work by Alexandrova et. al. [2] in which non-technical users could use a simple visual language to program the PR2 robot to perform manipulation tasks (Figure 1). Given the popularity of visual programming languages, it is surprising that no general framework exists which makes it easy to create new VPL's. In this course project, we address this problem by presenting a novel toolkit, called FlowViz, that enables developers to rapidly prototype and test new VPL's without wasting time writing boilerplate code which is not directly related to the intended functionality of the VPL. Additionally, we present three example languages written with FlowViz, including a web-based reimplementation of RoboFlow.

RELATED WORK

End-User Programming aims to enable non-programmers to create custom programs that suit their needs. Two widespread examples are spreadsheets and webpage programming. A particular subarea of end-user programming which has

gained popularity in the last several years is Visual Programming Languages (VPLs). VPL's have become especially popular in tools for teaching programming, as can be seen through websites like Code.Org [1]. There are several types of VPLs, such as form-based, flow-based or rule-based. We focus on flow-based VPLs [6].

Some notable examples of VPLs include Scratch [9], Blockly [5] and Lego Mindstorm [8]. Each of these VPL's allow children to code by providing visual languages that encapsulate complicated execution logic into simple composable blocks. Several tools also exist for creating flow-based programs [3, 10] using JavaScript. However, all of the above mentioned tools only allow the creation of programs in a specific VPL, whereas our proposed system will enable users to create their own VPLs.

Since flow-based VPLs are essentially graphs with nodes representing pre-programmed blocks of logic and edges representing program flow between the blocks, specific graph libraries can be used to visualize them. Sigma.JS [7] and D3.js [4] are both popular javascript libraries that can be used to visual graph information; however neither provide a simple interface for end users to create graphs. End user tools have to be built on top of these visualization libraries.

LIBRARY DESIGN

This project was developed on top of Node.Js, using Browserify to package the Node.Js code for use in a standard web browser. The design of FlowViz is heavily influenced by standard Node.Js design patterns in that FlowViz is structured as a series of modules each with a single goal. FlowViz consists of three main parts:

1. **Core Modules:** These modules provide core the core functionalities dealing with configuration parsing, managing the language graph, and rendering the language graph.
2. **Validators, Events, and Interactions:** These key structures act as the entry point for developers to customize the default functionality and interactions baked into FlowViz.
3. **Add-On Modules:** These additional modules provide simple default implementations of common VPL IDE interface components.

Core Modules

At the core of FlowViz there are a set of modules responsible for critical tasks involved in managing the visual program composition process and the rendering of visual programs to the screen.

Configuration Parser

The first thing a developer must do to create a VPL using FlowViz is define a configuration file that specifies the node types (language tokens) and the constraints between types (defining allowable edges). Therefore, the Configuration Parser module is by far the most important module in our entire system. This module is responsible for taking a configuration specification and building out a type hierarchy and it's set of associated constraints. This hierarchy of node types is specified by the developer by providing a list of subtypes of any given type. It is important to note that only the node types that are leafs in this hierarchy are actually instanciated as usable tokens in the VPL. This leads to the question: why specify a hierarchy of types instead of simply listing the different node types? The answer to this question lies in a key feature of the configuration parser: the parser cascades settings down the type hierarchy. This feature, inspired by the Cascading Style Sheets used to style HTML applications, is critical in that it allows developers to specify settings once for entire groups of node types, rather than repeat settings in multiple locations. This both makes it easier to make changes to a configuration file, and makes sure that configuration files are as succinct as possible.

In addition to cascading properties, the configuration parser has one other key feature: on top level types, unspecified properties are set to default values (where possible) before cascading values through the hierarchy. This feature allows developers to completely leave out properties from their configuration files when they are willing to accept the default value provided by FlowViz. This feature has the same benefit as the cascading properties feature in that it reduces the amount of configuration code a developer must specify. The only property which is mandatory for the developer to provide is the unique name of each node type in the hierarchy.

Graph Manager

The next critical module in the FlowViz library is the graph manager. The graph manager maintains a list of the nodes and edges that are currently in any given language graph and provides facilities for adding and removing nodes and edges. In essence, the graph manager is in charge of managing all of the data describing a given program. In addition, the graph manager is critical in that it administers all changes that occur to the layout of nodes and the drawing of new edges. This means that the graph manager provides a single point of access to all changes occurring in the language graph as an end user creates a program. The final important feature of the graph manager is that it enforces constraints on allowable edges between nodes by verifying all edges added to the graph are valid according to the constraint checking

module. This enforces the syntax of a VPL developed with FlowViz, ensuring that end-users cannot create syntactically invalid programs. Note, however, that FlowViz is completely agnostic to the actual semantics of a given program. FlowViz has no concept of what a program is actually supposed to be used for, it only views programs as a graph with constraints over edges. That being said, FlowViz does provide a method by which developers can enforce the semantics of their language, as will be discussed in the Validators section below.

Constraints

As the configuration parser generates the type hierarchy, it uses the provided node type names to generate namespace-like specifiers for each type. For instance, given type *Input* and subtypes *Keyboard* and *Mouse*, the configuration parser would generate the namespace-like specifiers *Input*, *Input.Keyboard*, and *Input.Mouse* respectively. These specifiers are important because they form the basis of the constraint declaration system in FlowViz. Constraints represent both the type and number of incoming and outgoing edges required for a node of a particular type. The type of an edge is represented as the pair $(StartNodeType, EndNodeType)$. In order to specify what the constraints are for a given node type, developers provide FlowViz with a list of possible incoming edge specifiers and outgoing edge specifiers in addition to the range of edge counts possible. For example, the constraints on operators in a simple calculator VPL might look like this:

```
{
  "incoming": {
    "range": [2,2],
    "types": {
      "Terminals.Number": [0,2],
      "Operations.*": [0,2]
    }
  },
  "outgoing": {
    "range": [1,1],
    "types": {
      "Operations.*": [0,1],
      "Output": [0,1]
    }
  }
}
```

Note the use of the $*$ in constraint specifications as a wildcard for all subtypes of a given type. This example says that an operator can have incoming edges from either other operations or from simple numbers and can have outgoing edges either to an output node (the equals sign) or to other operators. In addition it limits the number of incoming edges to exactly 2 edges and the number of outgoing edges to exactly 1 edge. This simple method of defining constraints is powerful for defining the syntax of a language in a brief declarative manner. The constraint checker module is in charge of interpreting these constraints and checking to verify that new edges are allowable given the constraints, in addition to determining what edges are missing given the constraints. By determining missing edges the constraint checker enables a powerful feature of FlowViz: auto-suggestion of the next possible nodes to add to a program and autocompletion of a program in the case that there is no ambiguity in which node must come next according to the constraints. The auto-suggest feature is demonstrated in Figure 2.

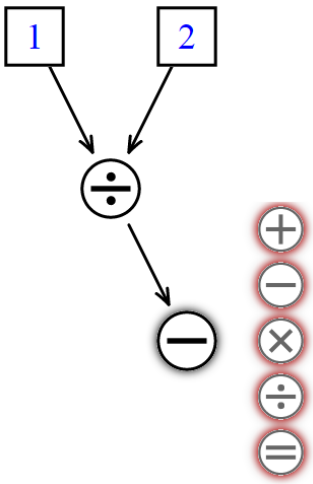

Figure 2. The auto-suggest feature informs end-users of possible nodes that could come next in their program. The current node is highlighted with a black drop shadow while suggestions are highlighted with red drop shadows.

Layout Manager and Renderer

The final core set of modules in FlowViz are the layout manager and the renderer. Together, these modules are responsible for the visual presentation of the language graph to the screen. This is accomplished through heavy use of both D3.js and Snap.svg.js. The layout manager also takes advantage of Dagre.js which is a javascript implementation of the GraphViz style (Sugiyama style) graph layout algorithms. Note that as part of the configuration file developers provide to FlowViz, developers are required to include SVG images for each of the node types. Due to this requirement, FlowViz is completely agnostic to the actual appearance of the nodes in the program graph.

Validators, Events, and Interactions

A major design consideration in the development of FlowViz was the fact that there is a great deal of variability between existing VPL's, meaning that for any

library to be useful in developing VPL's it needs to be general enough to allow for this variability, but still provide specific enough functionality to be a useful tool. To address this consideration, we designed the core of FlowViz to handle the common features between most VPL's and then we designed a series of extension points to FlowViz to handle the differences between languages. These extension points consist of custom data and graph validators, an eventing system, and an interaction management system.

Data and Graph Validators

The constraint declaration system handles general syntax validation for VPL's developed with FlowViz, but it does not allow for any concept of semantic validation. For instance, in the calculator example of the previous section, the constraints ensure that the division operator has incoming edges from two numbers, but it does not validate that the denominator is non-zero. In order to allow developers to capture this more expressive type of validation, we integrated the concept of custom validators into FlowViz. Custom validators are functions which take in information about a node or the entire graph and return either true or false to indicate whether the node or the graph is currently valid. This allows for arbitrarily complex validation over the semantics of any VPL created with FlowViz.

System Events

It is possible that FlowViz could be used as a base layer for a complete VPL IDE, however this would require the ability to run higher level code above FlowViz whenever certain events happen in FlowViz. For instance, an IDE could display summary information about a program such as the number and type of nodes. This would require the code that displays this information to be notified everytime a change occurred in the FlowViz graph. For this reason, we added a complete eventing system to FlowViz, which fires Node.Js EventEmitter events whenever certain events occur. This currently includes 17 events fired for things like the addition or removal of nodes and edges, the updating of the renderer, and changes in data stored on nodes.

Interactions

One of the most important features of FlowViz is in the way it handles interactions with the visual display of programs. By default, FlowViz defines interactions such as using single-click to select a node, or double-click to start drawing a new edge between nodes. It is very likely that these interactions will not be the best interactions to use for all VPL's built with FlowViz. For this reason, FlowViz was designed to allow developers to extend or override it's default interactions in addition to adding completely new interactions that FlowViz is not aware of by default. This

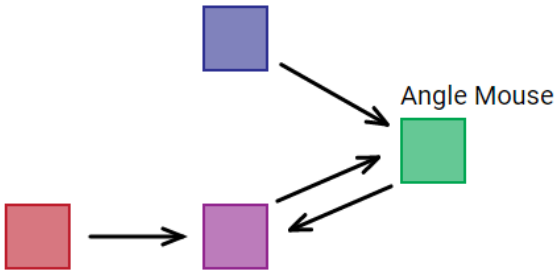

Figure 3. The MapAll language demonstrates how custom interactions can be added to to FlowViz-based VPL's. Notice the node label that appears when the cursor hovers over the green node.

is accomplished by maintaining a listing of a series of classes of interactions and, within those classes of interactions, dictionaries of events and callbacks. The current classes of interactions include the following: node-based and edge-based interactions, interactions for creating new edges, and interactions for moving items in the interface. These four classes cover the currently available functionality of FlowViz, allowing developers to completely redesign the interface FlowViz generates while at the same time still leveraging the core functionality of FlowViz.

Add-On Modules

The final major component of FlowViz is a series of add-on modules that implement commonly used interface elements that build on top of the core functionality of the FlowViz library. Currently, this includes a module for generating a default legend (a listing of all of the node types), a controls module that generates an HTML block with buttons to run things like graph validation or auto-completion, and a data editor which provides an interface to edit data items attached to nodes and edges in a program. These add-on modules further the goal of this project to make it possible to rapidly prototype VPL's without wasting time on boilerplate code. These add-ons also serve as a starting point for developers who wish to develop customized interface components.

EXAMPLE VPL'S

As a method of demonstrating the capabilities of FlowViz, we implemented three completely unrelated demo VPL's. These include two active research projects in the Human-Centered Robotics Lab (MapAll and RoboFlow) and a third classical beginning compilers course example (Calculator).

MapAll

MapAll is a simple programming language which allows end users to define tranformations on input from human-input devices such as keyboards and mouses.

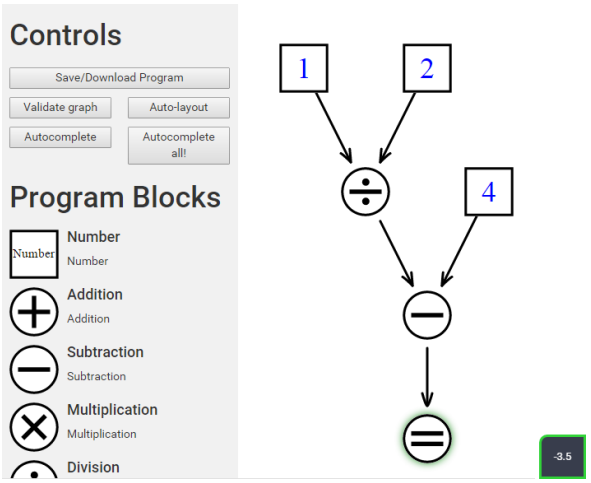

Figure 4. The Calculator language demonstrates how system events can be subscribed to in order to allow for running custom javascript code at the appropriate times. The blue text in the number nodes of this example is set when the data for the node is updated.

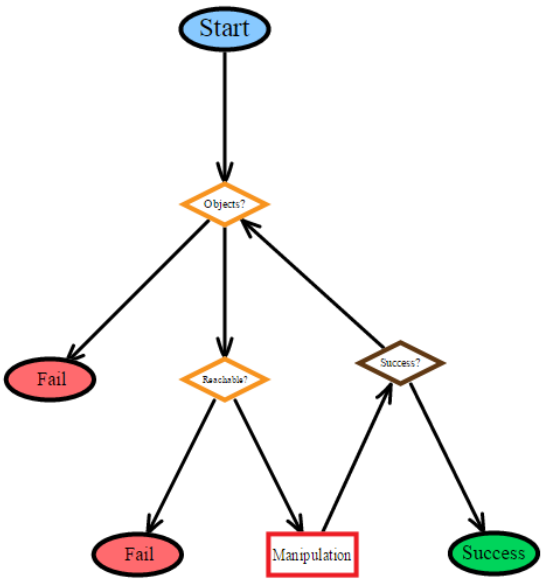

Figure 5. The RoboFlow language demonstrates how expressive a FlowViz-based VPL can be given only a set of edge constraints and accepting the FlowViz default parameters.

These transformations are mappings from input to either transformed input (using a keyboard to move the cursor), system-level operations (using a button to start a program), or application-level operations (using a button to have an email client open a new email to a particular person). This system is useful for allowing people with physical disabilities who cannot use standard input devices to reconfigure the way their input devices work. This VPL was partially reimplemented using FlowViz. In this example, the MapAll tokens are purely symbolic, being squares of varying colors. In order to make it clear what each colored square represents, we attached a new hover interaction to nodes in the graph (Figure 3). When any of the nodes are hovered over, the name of the node type appears above the node. This was accomplished with only several lines of javascript, demonstrating the how easy FlowViz makes the interaction customization process.

Calculator

In an ode to the classical compilers course assignment of developing a simple calculator language, we implemented a basic calculator which demonstrates how system events can be leveraged to add custom extensions to the FlowViz-generated interface. The calculator VPL app listens for data attached to number nodes to change and updates the SVG representing the number node to show the value attached to the node. Additionally, the app listens for the interaction of the output operator being clicked to run the calculation represented by the user's program. Upon this click event, the calculated value is displayed to the user (Figure 4).

RoboFlow

The final demo VPL implemented with FlowViz is a reimplementation of RoboFlow, a VPL that allows end-users to program the PR2 robot (see Figure 1). This example demonstrates how expressive of a language can be generated using the FlowViz defaults and a properly defined set of constraints in the provided configuration file. It should be noted that the visual appearance of the reimplementation of RoboFlow has several key visual differences from the original implementation. These differences are expounded upon in the Future Work section below.

DISCUSSION

In designing and implementing FlowViz, we came across several interesting challenges. The first was in determining the best way to both create a powerful library that makes development of new VPL's as painless as possible, but also allows for a high level of customization. In the end, this was accomplished by balancing baked-in defaults and the providing the capability to easily override those default, be it through the configuration file, or through interactions, event handlers, and validators. This ended up generating a library which is both powerful and customizable. Discovering the best way to accomplish this in terms of implementation was a challenging task that required several iterations of the major core modules.

Another interesting challenge which we discovered during this project was that the layout of nodes and

edges in a visual program can significantly effect the readability of the resulting program. Unfortunately, we did not have enough time to address these issues in the scope of this project; however, we are currently working to address layout intricacies for the next release of FlowViz (see Future Work).

FUTURE WORK

While we are excited about the progress that has been made with FlowViz to date, there are a significant number of improvements and extensions which should be made in the future. First and foremost, the current auto-layout system uses the Sugiyama style layout algorithm resulting in classical GraphViz-like node layouts. This type of layout is useful in some contexts, although for many VPL's different layout styles would be preferable. For instance, compare the layouts from the original implementation of RoboFlow (Figure 1) and the FlowViz-based implementation of RoboFlow (Figure 5). The original layout is more intuitive, and would be preferable as an option in future versions of FlowViz.

In addition to node layout, paths in FlowViz are currently only straight paths between nodes. This is very limiting in terms of the expressivity of edges and the possible set of layouts for any given program. In the future, we plan to implement an A*-type path planning algorithm in order to allow for paths with varying types of routes. Also, FlowViz currently attaches edges anywhere along a circle circumscribing language tokens rather than allowing for distinct connection points on the view of a token. We plan to extend FlowViz to allow for different node types to allow edges to be connected to them in different ways.

Finally, we received a good a piece of feedback from the poster session which we would like to implement in the future. Several people asked us if it is possible to observe the execution of the visual program through the interface we generate. While it is clear that FlowViz is a library for prototyping the visual and syntactical components of VPLs, and not a VPL compiler, it would be interesting to provide a layer which enables the creation of flow animations on top of any given program in a FlowViz-based editor.

CONCLUSION

In this work, we presented FlowViz, a visualization toolkit for the development of visual programming languages. FlowViz enables rapid development, prototyping, and testing of new VPL's aimed at end-user development. This toolkit is specifically designed to provide smart default interactions, but at the same time, provide the ability for developers to override or adapt the smart defaults with as little requisite effort as possible. FlowViz also provides both add-on modules for common interface elements, such as a legend, in order to allow developers to prototype VPLs quickly without having to worry about boilerplate code. Three example applications were presented, two of which are currently active research projects.

REFERENCES

1. Accessed 6/9/2015. *Code.Org*. website. http://code.org/.
2. Sonya Alexandrova, Zachary Tatlock, and Maya Cakmak. 2015. Visual Robot Programming for Generalizable Mobile Manipulation Tasks. In *Proceedings of the Tenth Annual ACM/IEEE International Conference on Human-Robot Interaction Extended Abstracts*. ACM, 163–164.
3. Henri Bergius. Accessed 6/9/2015. *NoFlow*. website. FlowHub, http://noflojs.org/.
4. Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D^3 data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on* 17, 12 (2011), 2301–2309.
5. N. Fraser. Accessed 6/9/2015. *Blockly*. website. Google, https://developers.google.com/blockly/.
6. Daniel D Hils. 1992. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing* 3, 1 (1992), 69–101.
7. Alexis Jacomy. Accessed 6/9/2015. *SigmaJS*. website. http://sigmajs.org/.
8. Seung Han Kim and Jae Wook Jeon. 2007. Programming LEGO Mindstorms NXT with visual programming. In *Control, Automation and Systems, 2007. ICCAS'07. International Conference on*. IEEE, 2468–2472.
9. Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and others. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.
10. IBM Emerging Technology. Accessed 6/9/2015. *Node-RED*. website. IBM, http://nodered.org/.