Information visualization with JHigraph — a progress report

This is a revised and up-dated version.

Theodore S. Norvell and Michael Bruce-Lockhart
Computer Engineering
Memorial University of Newfoundland
Email: theo@mun.ca mpbl@mun.ca

***Abstract*—Imagine a graph. Now imagine arranging its nodes in some sort of hierarchy. Now think of a forest and imagine adding edges between various pairs of nodes of the forest. Either way you get the same sort of thing: a higraph. A higraph is a set of nodes that are both linked by edges and arranged in a hierarchy. In 1987, David Harel introduced higraphs as a visual formalism, initially for system modelling. Higraphs are suitable for modelling many kinds of data: computer programs, linked storage structures, class diagrams, mind maps, system models, state charts, data-flow graphs. In this paper we discuss a new software library for information visualization and manipulation inspired by and based on higraphs. Rather than committing to a single visual formalism, we separate visualization concerns from modelling concerns. Each application can depict hierarchy and edges in its own way.**

CONTENTS

I. INTRODUCTION

Imagine a graph. Now imagine arranging its nodes in some sort of hierarchy. Now think of a forest and imagine adding edges between various pairs of nodes of the forest. Either way you get the same sort of thing: a higraph. A higraph is a set of nodes that are both linked by edges and arranged in a hierarchy. In [1] and [2], David Harel introduced higraphs as a visual formalism, initially for system modelling. In this paper we discuss a new software library for information visualization and manipulation inspired by and based on higraphs. Rather than committing to a single visual formalism, we separate visualization concerns from modelling concerns. Each application can depict hierarchy and linkages in its own way.

Section II give the mathematical definitions and gives examples of things that can be modelled with higraphs. Section III presents our software system. Finally section IV gives conclusions and presents ideas for further work.

II. HIGRAPHS

A. Definitions

A *directed graph* or *digraph* $(N, E, \overleftarrow{\ }, \overrightarrow{\ })$ consists of a set of *nodes* N, a set of *edges* E, and *source* and *target* functions ($\overleftarrow{\ }$ and $\overrightarrow{\ }$) such that $E \subseteq \operatorname{dom} \overleftarrow{\ } \cap \operatorname{dom} \overrightarrow{\ }$ and, for all $e \in E$, $\overleftarrow{e}, \overrightarrow{e} \in N$.

A *labelled digraph* $(N, E, \overleftarrow{\ }, \overrightarrow{\ }, \lambda_{\mathrm{node}}, \lambda_{\mathrm{edge}})$ is a directed graph with the addition of two label functions such that $N \subseteq \operatorname{dom} \lambda_{\mathrm{node}}$ and $E \subseteq \operatorname{dom} \lambda_{\mathrm{edge}}$.

An *ordered forest* $(N, \downarrow)$ is a set of nodes N and a *children of* function $\downarrow$. We think of $\downarrow$ as a function from nodes to sequences of nodes. We write $\downarrow$ as a postfix operator so that $a\downarrow$ is the sequence of children for node a and $a \downarrow i$ is item i in that sequence. We require four conditions on $(N, \downarrow)$ for it to be an ordered forest; the first three follow.

- For all $a \in N$, $a\downarrow$ is a finite or infinite sequence of nodes indexed from 0. Formally: $N \subseteq \operatorname{dom}(\downarrow)$ and for all $a \in N$, either $\operatorname{dom}(a\downarrow) = \mathbb{N}$ or there is a $j \in \mathbb{N}$, such that $\operatorname{dom}(a\downarrow) = \{i \in \mathbb{N} \mid 0 \le i < j\}$; furthermore, for all $a \in N$ and $i \in \operatorname{dom}(a\downarrow)$, $a \downarrow i \in N$.
- Each node has at most one parent and the sequences contain no duplicates. Formally: for all $a, b \in N$, and for all $i, j \in \mathbb{N}$, if $a \downarrow i = b \downarrow j$ then $a = b$ and $i = j$.
- No node is an ancestor of itself. (The term 'ancestor' will be defined shortly.)

For $a, b \in N$, if there is an i such that $a \downarrow i = b$, then a is the *parent* of b and b is a *child* of a; a node with no children is called a *leaf;* a node with no parent is called a *root*. If there is a finite sequence of 1 or more indices $i_0, i_1, \cdots, i_n \in \mathbb{N}$ such that $a \downarrow i_0 \downarrow i_1 \downarrow \cdots \downarrow i_n = b$ then a is an *ancestor* of b and b is a *descendent* of a. A root together with all its descendents is called a *tree*. Each node is in at most one tree. I'll add one final condition for being an ordered forests: each node must be in a tree; this restriction is redundant if N is finite.[1]

[1]To see why it is not redundant in the infinite case, consider the set of integers $\mathbb{Z}$ as the node set; each node has one descendant, that being is its numerical successor. Now the all restrictions are satisfied except for the latest. There is no root, so no node is in a tree.

		Target			
		Self	**Desc.**	**Anc.**	**Other**
S	**Self**	Loop	Down	Out	Out
r	**Desc.**	Up	Internal	Deep out	Deep out
c	**Anc.**	In	Deep in	Other	Other
.	**Other**	In	Deep in	Other	Other

TABLE I
CLASSIFICATION OF EDGES

A *hierarchical graph* or *higraph* is a structure $(N, E, \overleftarrow{}, \overrightarrow{}, \downarrow)$ such that $(N, E, \overleftarrow{}, \overrightarrow{})$ is a digraph and $(N, \downarrow)$ is an ordered forest. A *labelled higraph* is a structure $(N, E, \overleftarrow{}, \overrightarrow{}, \downarrow, \lambda_{\text{node}}, \lambda_{\text{edge}})$ such that $(N, E, \overleftarrow{}, \overrightarrow{}, \lambda_{\text{node}}, \lambda_{\text{edge}})$ is a labelled digraph and $(N, \downarrow)$ is an ordered forest. You can think of a higraph either as being a forest augmented by edges or as being a digraph augmented with hierarchy.

With respect to a node a, each node b can be classified as *self* (i.e. equal to a), *descendent*, *ancestor*, or *other*. Based on this classification we can classify an edge e with respect to a node a, based on the classification of e's source and target with respect to a. This classification is shown in Table I.

Given a higraph $G = (N, E, \overleftarrow{}, \overrightarrow{}, \downarrow)$ a *downward closed* set of nodes is a subset S of N such that

$$a \in S \wedge a \text{ is an ancestor of } b \Rightarrow b \in S, \quad \text{for all } a, b \in N$$

A *downward closed subgraph* of higraph G is a pair (G, S) where S is a downward closed subset of N. Any downward closed subgraph is uniquely defined by its set of *top nodes*,those nodes that are in S but that don't have a parent in S, i.e., that are either roots or that have a parent but one that is not in S. Let $E_S = \{e \mid \overleftarrow{e}, \overrightarrow{e} \in S\}$ then $(S, E_S, \overleftarrow{}, \overrightarrow{}, \downarrow)$ is itself a higraph. For the rest of this paper, the only subgraphs of interest are those that are downward closed and from here on we'll use *"subgraph"* and "downward closed subgraph" synonymously.[2] With respect to a subgraph (G, S), an edge e is *internal* iff $\overleftarrow{e}, \overrightarrow{e} \in S$, *entering* iff $\overleftarrow{e} \notin S$ and $\overrightarrow{e} \in S$ *leaving* iff $\overleftarrow{e} \in S$ and $\overrightarrow{e} \notin S$, and *unrelated* iff $\overleftarrow{e}, \overrightarrow{e} \notin S$.

One particular subgraph is (G, N), which is termed the *whole graph*. Thus each higraph can be seen as a subgraph and each subgraph can be seen as a higraph. From a software engineering point of view, this circularity raises the question: which class should inherit from which? We chose Higraph as the base interface with interfaces WholeGraph and Subgraph both extending from it. (See Fig 4.)

We say that an edge e is *governed by* a node v if e is Up, Down, or Internal with respect to v but not Up, Down, or Internal with respect to any descendant of v. A loop e, i.e. an edge that has the same source and target, is governed by the parent of its endpoint, if there is such a parent. Any other edge e, i.e. a nonloop, is governed by the least common anscestor of its source and target, if such exists. Thus each edge is governed by at most one node. Edges that aren't governed by any node are said to be *governed by* the higraph. These will be edges that are either loops on roots or that run between distinct trees of the forest. In the extreme case where no node has a child (i.e. we have an ordinary digraph), all edges are governed by the higraph. An edge e in E_S is said to be *governed by* the subgraph (G, S) if it is not governed by any node in S; these will be edges that are either loops on the top nodes of (G, S) or that run between trees defined by two different top nodes of (G, S).

[2]One could conceive of subgraphs that are not downward closed, but in such cases the same "child of" function could not be used. For example we could take an arbitrary subset S of N and use a modified 'child of' function that omits from each node's child sequence any node not in S.

B. Applications

Structures that can be modelled by higraphs are common in computing. Anything that can be modeled by a graph, a tree, or a forest can be modelled by a higraph. Here are some examples that use both hierarchy and edges. (0) Consider a set of HTML documents. Each is an ordered tree of HTML elements, text nodes, comments, etc. Certain of the elements (in particular those with A and IMG tags) link to elements of the same or other trees. These links are directed edges. (1) Consider a snapshot of the objects in an execution of a C program. Each object is a node. Structures (structs) and arrays are parents and induce a hierarchy. Pointer objects are the sources of edges that link nodes. (2) In a file system, drives are roots and directories and drives are the parents of files and directories. Symbolic links are modelled by edges.[3] (3) Consider a computer program written in a structured language such as C. Each subroutine is a tree of statements. Go-to statements provide links (edges) between statements. We could also use edges to provide links from nodes representing declarations to the nodes representing the uses of named entities. (4) Almost any engineered product can be viewed hierarchically, for example an automobile has an engine, which has cylinders, which have spark plugs. At the same time there are connections that pass through hierarchical boundaries. E.g. the electrical system must connect the spark plugs. Similarly a program may consist of a hierarchy of modules, but we might have a call from one module to a distant cousin. (5) Data flow graphs in the ProGraph [3] language can be seen as higraphs in which each 'frame' is a node. 'In' and 'out' edges connect the frame to its child nodes. (6) A project plan has tasks divided into subtasks. Edges connect tasks that are dependent. (7) Statecharts, as found in UML, are higraphs [4]. Each state, pseudostate, and region is a node. Each state is parent to its regions and each region is parent to its substates. Transitions are edges. See Figure 1 In fact, historically, statecharts [1] came first and higraphs were invented by David Harel [2] later to help formalize state-charts. Harel's formalization is a bit different in detail from that presented above.[4]

[3]Unix file systems don't quite fit this model because a file (other than a directory) in unix may be hard linked into more than one directory. For unix we could model hard links to nondirectory files by edges. This way each file (other than a directory) is both a root and a leaf. The . and .. links in unix are also nonhierarchical and would have to be modelled by edges.

[4]Firstly, Harel's formalism divides the children of a node among a number (1 or more) of partitions. A node a with a child b in partition x, in a Harel higraph, can be modelled in our formalism by a node a with a child x, which has b as a child.

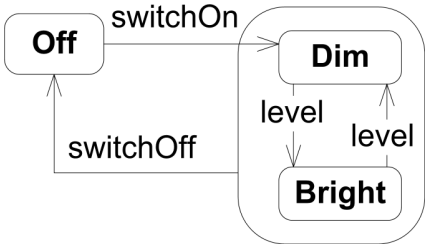

Fig. 1. A statechart.

III. JHIGRAPH

Our interest is in information visualization and manipulation. A number of projects that we were working on all had similar needs and it was decided to create one underlying visualization package, based on higraphs. The projects were as follows.

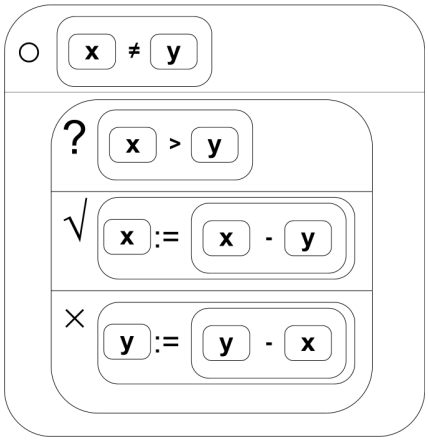

Fig. 2. A command in the PLAY language

- In the Teaching Machine [5] we were working on programmer defined visualization (PDV) of data. For PDV the programmer adds statements to their code constructing a visualization of the programs data state. This allows the Teaching Machine to show more abstract views of program state than can be automatically generated. For example, if one is using an adjacency matrix to represent a graph, the TM can automatically generate a visualization of the matrix, but to display the graph requires some explicit instruction. Using PDV the programmer adds instructions to the program to build a visualization of the graph and to change the visualization of the graph each time the matrix is changed.
- Both the TM and the PLAY programming system display the data state of the program as a graph with pointers linking together objects.

Secondly, Harel's original formalism does not rule out that a node may have multiple parents. A node with multiple parents is not easily modeled with our formalism.

- In the PLAY programming system, the program is represented as an abstract syntax tree, presented to the user using a boxes-in-boxes visualization. See Fig 2. The programmer edits their program by direct manipulation (e.g. drag and drop) of the tree. For example to add a new command to a subroutine, the programmer drags from a palette of statement types and drops a new box within the box representing the subroutine's body.
- Other systems in early stages include a system for editing and executing state-charts and a system for editing regular expressions presented as "railroad diagrams".

The first two of these only display data. The last two demand that the user may manipulate (e.g. via the mouse or a touch interface) the visual representation causing changes to the underlying data structure.

Based on the needs of these projects and the assumption that others would have similar needs, we started the JHigraph project. JHigraph provides the basic models, views, and controllers needed to build various GUI systems based on higraph models. Because the Teaching Machine and PLAY projects are written in Java, we chose Java as an implementation language. Initially JHigraph is layered above the Swing GUI library, but it is intended that it be portable so that it can be used with SWT and perhaps other libraries.

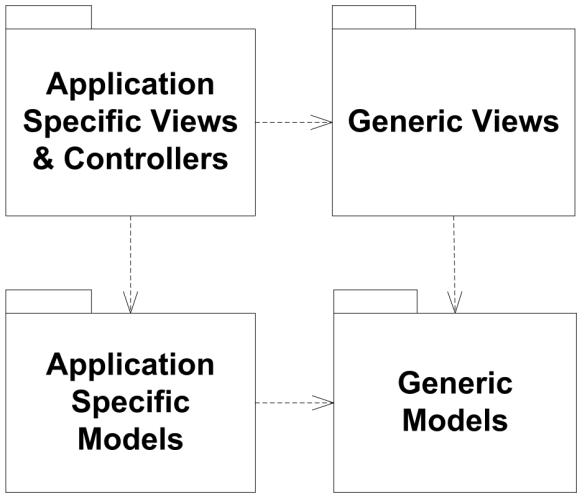

Fig. 3. Package structure of a JHigraph application.

The structure of a JHigraph based application is shown in Fig 3. At the right are the models, views and controllers provided by JHgraph. These classes and interfaces are generic. The application provides specializations of these classes and implements any abstract methods in order to obtain classes that are specific and concrete. These classes are shown at the left of Fig 3. The system is thus layered along two orthogonal axes. Views and controllers depend on models, but not the other way around; and application specific classes depend on the generic classes, but not the other way around.

A. Models

JHigraph provides interfaces for nodes, edges, higraphs, whole graphs, and subgraphs. For example the interface WholeGraph represents mutable objects whose state represents a higraph. WholeGraph objects also acts as a factory objects for creating objects representing nodes, edges, and subgraphs. Only Node, Edge, and Subgraph objects created by the same WholeGraph may be composed. Node objects are initially created as roots of the higraph. Operations on node objects include

- delete: The node represented by the object and all its descendents are removed from the higraph and all subgraphs, as are all edges that have such a node as source or target. After deletion, none of these objects can be used for anything.
- duplicate: The node and all its descendents and all edges that have such a node as its source or target are duplicated. The duplicate of the node becomes a new root.
- insertChild: Moves another node to become a child of this node.
- replace: Replaces this node with another.
- detach: Make a node become a root.
- permuteChildren: Rearranges the children of a node.

Node objects carry a label object (a 'payload' in the terminology of the system) which may be replaced.

Subgraph objects are mutable objects whose state is a subgraph of the associated WholeGraph. We can add and remove nodes from each subgraph object, subject to the constraint that each subgraph remains downward closed. This constraint is enforced by limiting the mutator operations on subgraph objects to (0) adding a node not in the subgraph as a top node and (1) removing a top node. Subgraph objects are useful for representing parts of the whole graph. For example in PLAY we might represent each subroutine with its own subgraph. Nodes "cut" by the user can not be deleted (otherwise there could be no "paste" operation), rather they are moved to a cut-buffer subgraph. Similarly a trash-can for nodes would be represented by a subgraph.

Edge objects are very simple: they can be deleted; their source and target can be changed; their payload (i.e., label) can be changed; and that is about it.

The notion of well-formedness is important in a number of applications. Some higraphs are well formed, while others, maybe, are not. For example, in the case of a higraph of HTML elements, we might require that a node representing a TR element must have as children only nodes representing TH or TD elements. In the PLAY language there are similar restrictions, e.g., 'while commands' must have two children: an expression and a block. We would like to maintain as an invariant of each higraph that it is well formed; as a precondition to each mutator, there is a check that the operation will leave the higraph well formed. For that reason, each mutator is accompanied by a query that asks whether the mutator would succeed or fail. For example there is a canInsertChild query that should be called before a call to insertChild. These queries are particularly useful for drag and drop operations. Suppose the user drags a depiction of a node from one place in a depiction of the graph to another; one can use the queries to feed back (e.g. by changing the mouse pointer shape) to the user whether a drop at the current mouse location will succeed or fail.

The application may provide its own implementations of these interfaces. We provide two implementations that can be used as base classes for the application's implementations. The first implementation has no notion of well-formedness. It returns true for all "can" queries, except where there are purely structural reasons for returning false — for example making a node its own child. The second implementation, based on the first, has a notion of well-formedness based on each node having a tag and well formedness depending on the sequence of tags of each node's children. This is very similar to the notion of well-formedness in XML. The routine that determines whether a sequence of tags is acceptable is left abstract and can be filled in at the application specific level of the model.

B. Views

Each node, edge, whole graph, and subgraph object can be depicted graphically on a computer's screen. Indeed we may want multiple depictions at the same time — for example to support a zoomed view and an overview. To represent these depictions we use view objects. Each view object is responsible for maintaining information about a depiction, such as where it is on the screen, what colour it is, what shape, whether it is collapsed, etc., and is capable of repainting the depiction as needed. Each WholeGraph or Subgraph object can have associated to it zero or more HigraphView objects. For each HigraphView hv and each node a in its higraph, there is at most one view object $v_{a,hv}$. See Fig 4.

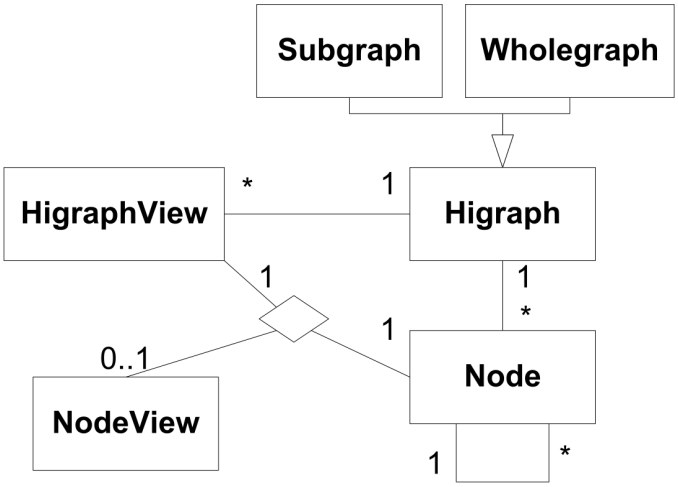

Fig. 4. View and model classes.

Assuming they all exist, the node views, for any higraph view hv, thus form a forest such that $v_{a,hv}$ is parent to $v_{b,hv}$ if a is parent to b. Being able to navigate this forest is important,

for example for layout or repainting. However, explicitly linking the views together would parallel the structure already supported by the model and create the problem of keeping the two parallel forest structures synchronized. Instead views are implicitly linked via their model objects. Each node view $v_{a,hv}$ can ask its model object, a node object, for that object's children and then can ask each child node object b for its view relative to the same higraph view, i.e., for $v_{b,hv}$. If such a view does not exist, then it is created on the fly using a factory object associated with the higraph view. Consider what happens when a node b is created and then added under a parent a. The next time the views are refreshed, a view $v_{a,hv}$ of node a will at some point ask for all views of a's children associated to the same higraph view hv. This will cause a new view $v_{b,hv}$ to be created. Similarly if a node b leaves a higraph with view hv, the view $v_{b,hv}$ will become irretrievable, for the time being.

A similar mechanism is used to implicitly link node views and edge views: if $\overleftarrow{e} = a$ or $\overrightarrow{e} = a$, then $v_{e,hv}$ and $v_{a,hv}$ should be able to find each other via their model objects.

In addition to views for higraphs, nodes, and edges, we also have views corresponding to drop zones (places where nodes and other things can be dropped), labels, and connection points. These additional views are linked directly to their parents, typically node or edge views.

C. Layout Managers and Transitions

Layout managers are associated with higraph views and node views. These managers are responsible for placing all views in the view tree beneath the associated view. The layout manager for a higraph view will place the views of the top nodes. The layout manager of a node view places its child node views and so forth. If an edge is governed by a node in the higraph, it's view is placed by the layout manager for that node's view; otherwise, if the edge is governed by the subgraph or the whole graph, it's view is placed by the layout manager for the HigraphView.

The job of the layout manager is to find the next position and size for each node view (and other views, such as drop zones) and routes for its edge views. After each round of laying out the views then make an animated transition to their new positions. Views that have become invisible, shrink to nothingness; views that have become visible expand to their new size. These animated transitions reenforce the continuity of identity between the original and the new depictions of each view.

D. Input

The handling of input events is highly application dependent. We provide an interface HigraphEventObserver that the application may implement, if it needs to. Events sent to the application via this interface include: clickedOn (when the mouse is clicked over a subgraph view), mouseOver (when the mouse moves over a subgraph view), importData (when a drag-and-drop operation ends over the subgraph view), and others. Since views within a subgraph view may overlap in extent —consider a node whose children are drawn within its bounding box, or an edge that passes in front of a node—, for each event, the application is passed a list of views that intersect the mouse position, in reverse draw order. It is up to the application to decide which view is the intended recipient of the event, although usually it is the first on the list that the event could affect.

IV. Conclusions and further work

The work represents a rigorous separation of concerns between model and view. Mathematical formalization of the model was crucial to creating a clean model.

Although there is still work to be done on JHigraph, we are reaching a near a release of programmer defined visualizations for the Teaching Machine. Animated transitions are the main piece required to be done to support this. Work on the PLAY system —with its higher demands for input event processing— is continuing as time permits.

References

[1] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, June 1987.

[2] D. Harel, "On visual formalisms," *Commun. ACM*, vol. 31, pp. 514–530, May 1988.

[3] P. T. Cox, F. R. Giles, and T. Pietrzykowski, "Prograph: A step towards liberating programming from textual conditioning," in *IEEE Workshop on Visual Languages*, pp. 150–156, 1989.

[4] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual*. Addison-Wesley Professional, 2nd ed., 2010.

[5] T. S. Norvell and M. P. Bruce-Lockhart, "Taking the hood off the computer: Program animation with the teaching machine," in *Canadian Electrical and Computer Engineering Conference*, May 2000.