An Adaptable Software Architecture for Rapidly Creating Information Visualizations

Rick Kazman
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
rnkazman@cgl.uwaterloo.ca

Jeromy Carrière
Nortel
Ottawa, Ontario, Canada
jayc@bnr.ca

Abstract

While data visualization is an increasingly important analysis tool, both in research and commercial communities, the process of creating these visualizations is still quite complex. Visualizations tend to be hand-crafted, each one different from the previous. This paper presents VANISH, a system created to ease the rapid creation of arbitrary data visualizations. VANISH simplifies the creation of visualizations in two ways: by providing a special-purpose visual language, called VaPL, which maps semantic domains to visual domains; and by easing the integration of new semantic and visual domains. The software structure of VANISH emphasizes separation of concerns: we follow the Arch/Slinky metamodel of user interface software by not only separating the underlying semantic domain to be visualized from the dialog and presentation components, but by providing virtual semantic domain and presentation layers. In this way, it is simple to port from one visualization domain to another, and from one presentation component to another. We demonstrate that it is simple to create abitrary visualizations by implementing several well-known visualization styles such as cone-trees, tree-maps, fisheye views.

Keywords: information visualization, software tools, visual programming languages

1 Introduction

Over the past decade or so there has been a growing emphasis the use of visualization techniques in a wide variety of information technology applications. Visualization is used whenever the data to be understood and managed is large and complex. For example, there has been substantial activity in the areas of software visualization [1], financial data visualization, scientific data visualization [6] and database visualization [7]. The majority of research in visualization has been on developing better presentation techniques. The way in which these visualizations are implemented has received relatively little attention. In particular, few special-purpose tools exist to ease the burden of creating complex visualizations. Such visualizations are typically hand-crafted using a general-purpose high level programming language, typically Fortran, C, or C++.

This paper presents the VANISH (Visualizing And Navigating Information Structured Hierarchically) system and architecture. VANISH is a data visualization programming environment designed to support the creation of arbitrary visualizations over arbitrary semantic domains. VANISH, unlike most visualization tools, contains a general-purpose visual programming language call VaPL (VaNISH's Programming Language). VaPL is just one layer—the *dialog* layer—of a system designed with one dominant software engineering concern in mind: integrability. VANISH is structured to simplify the integration of many different semantic domains and presentation toolkits. In addition, although VaPL contains all of the functionality of a general-purpose programming language, it provides additional functionality specifically aimed at easing the creation of visualizations. These two considerations—the special features of VaPL which simplify the creation of visualization, and the software structure of VANISH, which eases the integration of new semantic domains and presentation toolkits—are the main contributions of this paper.

2 The Software Structure of VANISH

This section will discuss the software organization and structure of VANISH. VANISH is an instantiation of the Arch model of user interface software [12] (a refinement of the Seeheim model [2]). The Arch model divides all user interface software into the following functional categories:

- **Functional Core**: This is the underlying functionality that the system exists to expose (it is often called the *application*, or *domain-specific component*). In the case of visualization systems, the functional core is whatever semantic domain is to be visualized.
- **Functional Core Adapter**: The functional core adapter mediates between the dialogue and functional core by providing a unified, generic view of the functional core to the dialogue. For example,

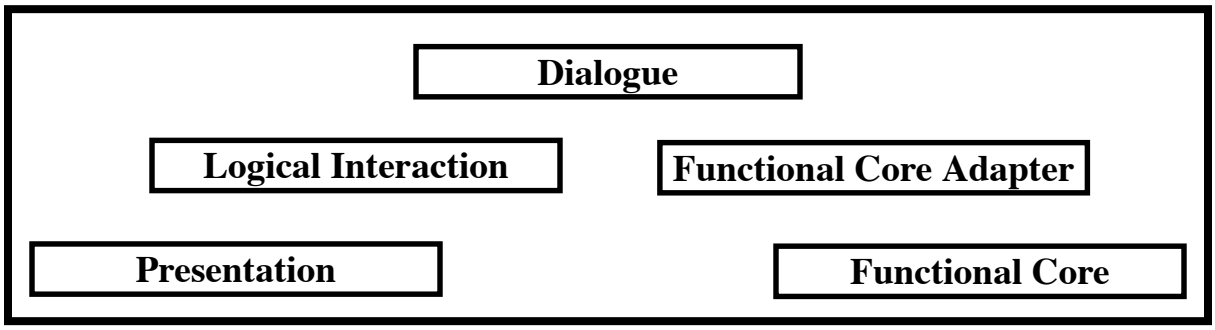

Figure 1: The Arch Model of Interactive Software

when accessing a federated database, a user should not be concerned with the details of the database—whether it is hierarchical, relational, or object-oriented, or with what form of SQL the query engine (which resides in the functional core) uses. The functional core adapter provides a *unified* view of multiple functional cores. Results from the domain-specific component are passed through to the dialogue and onward for presentation to the user.

- **Dialogue**: This function mediates between domain specific and presentation specific functions. It ensures consistency (possibly among multiple views of data), controls task sequencing and context management.
- **Logical Interaction**: This function mediates between the presentation and the dialogue. It provides a set of logical interaction objects (sometimes called virtual objects) to the dialogue. For example, several commercial virtual toolkits, such as XVT [9] and Galaxy, provide a common set of presentation objects that are mapped to specific toolkit objects, depending on the target platform for the application.
- **Presentation**: This function implements the physical interaction between the user and the computer. Presentation deals with input and output devices and is typically realized as a user-interface toolkit and/or a proprietary interface library.

The name Arch is based upon a visual metaphor of the software organization: the presentation and functional core are the foundations of the arch. The dialogue is the keystone. Each of these three components existed in the Seeheim model. The Arch model adds two "virtual" layers to a Seeheim software structure,[1] in order to increase the separate modifiability of the components, and to ease porting a system between presentation toolkits and application domains. The model is depicted in Figure 1.

Clearly the separation of concerns promoted by the Arch model has important implications for the visualization domain, because visualization faces the same software problems as other interactive systems: the desire to reuse as much software as possible, the desire to minimize the effort of porting, the desire to minimize the difficulty of system modification and maintenance, and to localize the effects of a change.

Given these objectives, we need to precisely determine how the Arch model maps to VANISH in order to understand the ways in which VANISH supports the creation, modification, and portability of visualizations. We will first describe the mapping in broad brush, and then describe each function in more detail.

VANISH's presentation layer (a user interface toolkit or rendering package) displays the data visualization to the user. The logical interaction layer, called **LPresentation** provides a toolkit-independent set of abstractions for visualizing data. VaPL provides the dialog layer—a way to specify the representation of the visualization, independent of the particular domain being visualized and the toolkit or rendering engine being used. The functional core adapter, called **BaseNode**, provides generic access mechanisms to the individual nodes of the domain and their attributes. The functional core is the domain to be visualized. The details of the implementations of these layers in VANISH will now be described in more detail.

3 Functional Core

One of the objectives of VANISH was to be completely independent of the semantic domains being visualized. We wanted VANISH to be a general-purpose visualization tool, and so our restrictions on the nature of the domain to be visualized had to be minimal.

1. There are other differences between the two models, but these are beyond this scope of this paper.

In order to visualize some domain, the following conditions must be met. The information must be representable as a set of *nodes*, each of which:

1. can be identified uniquely,
2. has a set of attributes,
3. has a computable relationship with other nodes.

These requirements are sufficiently unconstrained to permit the visualization of most data. We have already created visualizations of a Unix file system, an RCS repository, and the World-Wide Web. Each of these will be discussed in the Results section.

VANISH treats all information as though it were structured hierarchically (as do Mukherjea *et al* [7]). A root is defined for the hierarchy and relationships between nodes are treated as parent-child relationships. If the structure to be visualized is actually a network rather than a tree then cycles are automatically broken in order to "treeify" the network. If the structure to be visualized does not have an inherent tree structure, it can still be visualized using VANISH by associating some computable aspect of a node's relationship with other nodes as the "link", or parent-child relationship, between nodes. We demonstrate this point in Figure 6.

4 Functional Core Adapter

The computational mechanism that provides generalized access to information structures is a set of C++ classes, each of which is derived from the **BaseNode** class. The **BaseNode** class serves the role of the Functional Core Adapter in the Arch model. It provides a uniform interface to abstract functionality that is common across *all* information structures, in the same way that a virtual toolkit provides a set of common interaction abstractions, irrespective of the underlying graphics software and hardware. In particular, this class implements parent-child relationships, along with the facilities to manipulate structures of this form. **BaseNode** also provides global naming of nodes within the semantic domain, and a mechanism for defining and maintaining a list of attributes for each of the nodes.

An attribute in a VANISH hierarchy is a (name, value) pair. Values are currently defined to be of integer, floating point, boolean, or string types. Five node attributes are created by **the BaseNode** class constructor for all nodes, irrespective of the type of hierarchy. Two of these are integer attributes: the number of links between the node and the root of the tree (i.e. the node's depth in the hierarchy) and the number of children of the node. The three remaining attributes are boolean: whether the node is the root of the tree, whether it is a leaf, and whether it is the current *focus* node. The focus indicates the user's current position within the hierarchy (for example, the current working directory in a Unix file system). Changes in the focus can trigger changes in the presentation, as will be discussed in Section 5.2.

For VANISH to visualize a new variety of hierarchical information (a new domain), a new functional core for this domain must be integrated with the functional core adapter. To accomplish this integration all that is required is to create a new subclass of **BaseNode**. This subclass, once created, can be reused for any visualization of this semantic domain, and so the integration effort, minimal though it is, can be amortized over many uses of the domain. We give several examples of this process in Section 8.2.

This new subclass must provide an implementation of a function called **MakeNode**, that defines how nodes are related in the structure, what additional attributes the nodes contain, and how those nodes are named. If the information domain to be visualized has no inherent topology (for example, if one wanted to visualize an unstructured database of images), then the implementor of a **BaseNode** subclass must define at least one relationship such that *links* between nodes may be computed (to determine if two nodes are related, and how strongly). In the example of the image database, two nodes might be considered to be linked if they exceeded some threshold of similarity using a measure such as RMS distance.

When the **MakeNode** function is called for a given node within the hierarchy, it will:

1. add the node's additional attributes (those specific to this subclass) to the list that has been created by the **BaseNode** constructor;
2. register the node's unique name by calling a member function in the **BaseNode** class;
3. construct any children of the node, calling the appropriate **BaseNode** member functions to associate, or link, the children with the node.

The notion of "children" is clearly used in an abstract way. The children of a particular node are those nodes whose relationship with that node has been defined for the semantic domain through the **BaseNode** member function. A node's attributes are arbitrary: they can represent any salient aspect of a node's content or structure. A node's attributes will not, however, refer to its appearance. That is the separate domain of the presentation, which actually represents the nodes, and the dialog, which maps semantic attributes to visual attributes.

5 Dialogue

When attempting to visualize a large information structure, the most appropriate technique is seldom obvious. Designers must be able to quickly experiment with a variety of presentation mechanisms. Novel mechanisms for presenting specifically hierarchical structures have been developed by, among others: Robertson *et al* [8] and Koike [5], while methods for effectively navigating information structures have been presented by Schaffer *et al* [11], and Mukherjea *et al* [7]. This is the function of the dialogue layer in an interactive system: to mediate the linkage between the functional core and the presentation.

In previous systems, some effort has been devoted to allowing flexible association of visual attributes with information attributes. For example [7] allows a set of pre-determined semantic attributes to be associated with a pre-determined set of visual attributes, via a form-filling interface. Some commercial visualization packages, such as AVS, provide a set of pre-packaged filters and a data-flow language that allows a user to connect filters together. These data-flow languages can only connect filters together; they provide no facilities of general-purpose programming languages.

VANISH allows arbitrary visual and semantic attributes to be defined and related in an algorithmic manner, through a high-level, interpreted, visual data-flow language. VaPL (*VaNISH's Programming Language*) code is implemented in, and interpreted by, the VANISH environment. Being interpreted, VaPL encourages rapid prototyping. This point is uncontroversial. Using a *visual* language to specify the dialogue layer is more controversial. Our experience has been that VaPL simplifies the process of creating data visualizations because it permits only syntactically well-formed data connections to be made, because the language contains functionality specifically tailored to information visualization, and because it keeps individual code chunks—called *mappings*—small (by simply limiting the amount of screen space available to describe a mapping).

Prima facie, it seems strange to call the restriction on screen space for mappings an advantage. However, this is just the enforcement of a sound software engineering principle for controlling the complexity of code: minimizing the size of individual pieces of code, and requiring that complex functionality be broken up into smaller, simpler pieces, each of which has a well-defined interface with the rest of the world. In standard procedural languages the same effect is achieved via guidelines on the maximum size, in lines of code, for a single procedure.

5.1 Structure

VaPL provides a general-purpose programming language. A VaPL program is a set of *mappings*. A mapping is a function: it has a well-defined interface, it accepts inputs, performs processing, and produces outputs. Mappings are named and may call other mappings by name. Sets of related mappings are collected into *mapping libraries*, which can be loaded and saved by the VANISH programming environment. Mappings typically manifest themselves as mappings from node semantics to presentation attributes, however this is not strictly necessary, as will be shown next.

A VaPL mapping comprises a set of *operations*, which are displayed with inputs at the top and outputs at the bottom. A set of arcs connects output terminals to input terminals, specifying a direction of data flow. A node in the graph performs a specific operation on its inputs and sets a value at its output. This data flow model has been used in many visual languages, such as Prograph [1].

VaPL organizes mappings into *cases*. Each mapping is actually a set of cases, arranged in a fixed enumeration. The execution of a mapping begins at the first case and progresses until either a case completes successfully or until no operation can execute in the current case. If a particular case can not complete successfully and more cases exist, then control is transferred to the next case in the enumeration. An example of a multi-case method is shown in Figure 2.

This left-hand side of Figure 2 shows how one would implement the computation of the *n*th Fibonacci number using VaPL. The first case fetches the mapping's input (*n*), and tests for inequality with zero. If the test succeeds, execution continues in the next case—indicated by the disc at the right of the inequality operation. If the test fails, execution continues in the first case and the output is set to zero. The thick dashed line connecting the logical operation to the **SetOutput** operation is a synchronization primitive—it indicates that the former must occur before the latter. The second case is similar to the first. The third case is most interesting: after fetching the input value it subtracts one and calls the mapping (named “fib") recursively. One is subtracted from the result of the previous subtraction and the mapping is called again. The outputs from the two recursive calls are added to compute the final result. The right-hand side of Figure 2 shows the VaPL code for Quicksort (omitting the second case, which only deals with an empty list).

These simple examples show that VaPL is a general-purpose programming language: it includes input, out-

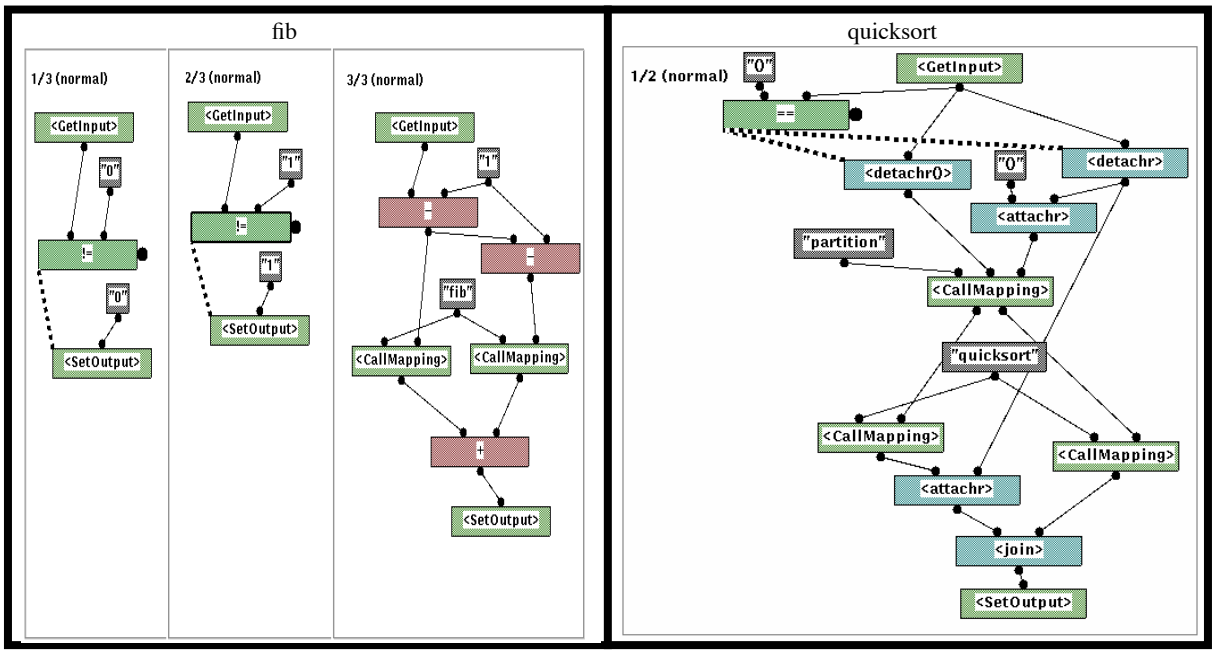

Figure 2: VaPL implementations of the Fibonacci computation and Quicksort

put, assignment, mathematical, boolean and list operations, an adequate (although not rich) set of built-in data types, conditionals, and iteration (not shown in this example). VaPL also has the ability to package functionality together in procedure-like packages (the mappings), and invoke these mappings recursively. Data types are determined dynamically: all data is passed as strings and typed by VaPL only when necessary. Type compatibility is checked by the language at run time. For example, adding a boolean to an integer results in a run-time error.

As stated earlier, mappings provide the means of decomposing a VaPL program into procedural chunks. The **RunMapping** and **CallMapping** operations provide the mechanism by which mappings can call other mappings. **RunMapping** takes as its only argument the name of the mapping to execute, while **CallMapping** takes a mapping name and an arbitrary number of additional parameters. Further differences between these two operations will be discussed below. Two other operations, **GetInput** and **SetOutput**, provide the mechanisms for retrieving input parameters and setting result values, respectively, as demonstrated in Figure 2.

However, VaPL is *not* just another visual programming language, and is *not* just another general-purpose programming language. VaPL includes special functionality and primitives for visualization. It provides a powerful environment for the rapid creation of arbitrarily complex visualization code which we will describe next.

5.2 Visualization Using VaPL

VaPL contains functionality which makes it particularly suited to visualization: it provides generalized visualization capability by executing mappings in the context of hierarchy nodes. Any mapping can execute with respect to a particular hierarchy node and while doing so, it has direct read and write access to the attributes of that node. The mapping code utilizes the node's attributes to compute visual attributes on which rendering is based (as will be described in more detail below).

VaPL contains a rich set of built in *node operations*, such as **GetChildren**, **GetRoot**, and **GetParent**, for navigating a hierarchy. These operations retrieve the children of a node, or the root of the hierarchy, or the siblings or parent of the current focus, and so forth. In addition, VaPL defines a number of *attribute operations* such as **NodeAttr** and **SetNdAttr**, to get and set arbitrary attributes of nodes. Because a node's attributes are defined by the semantic domain (or, more precisely, by the particular subclass of **BaseNode** being utilized as the semantic domain), the VaPL language can refer to any attribute of any semantic domain without modification.

A visualization is thus created by writing VaPL code which is executed for each "relevant" node in the hierar-

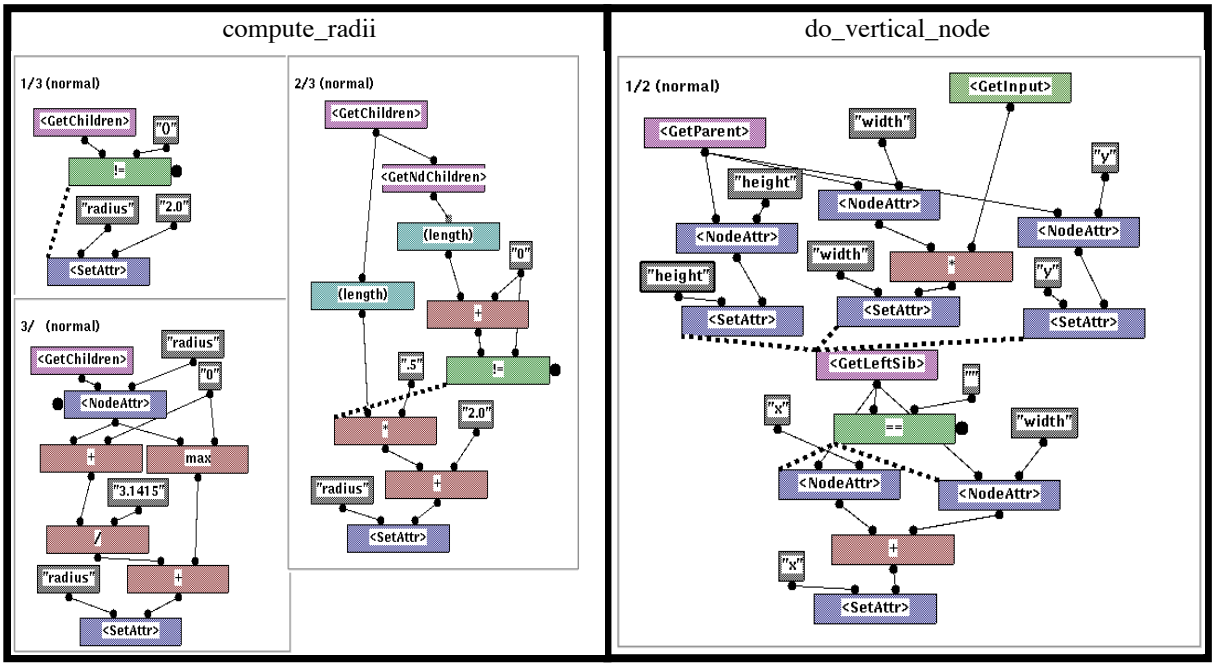

Figure 3: VaPL code to compute cone tree and tree-map layouts

chy. The mapping itself can determine what nodes are relevant. For example, one might only want to visualize nodes within some constant distance from the user's current focus, or only those nodes which pass some semantic restriction.

To give an example: often circumstances arise under which one wishes to compute some visual attribute for a given node based on the attributes of other nodes. For example, one might want to change a node's size based upon its depth in the hierarchy, or its number of children, or the "value" or its children (where value is some dynamically derived measure). By automatically extending the language to include exported attributes from a new semantic domain, a VaPL programmer can simply refer to those attributes and associated them with visual channels.

Some visual attributes can only be built in a bottom-up fashion. For example, the diameter of a cone tree rooted at a node N is dependent upon the number and size of N's children, which are themselves determined by their own set of children, and so forth. When it encounters an undefined attribute, the **NodeAttr** operation provides three courses of action:

1. cause a run-time error and halt execution of the mapping;
2. switch to the next case;
3. recursively call the current mapping in the context of the node being queried, in an attempt to compute the desired attribute.

This third option is precisely what is required to allow a programmer to implement "bottom-up" computations in hierarchical structures. Figure 3 shows two VaPL mappings specific to visualization: the left hand mapping computes the radii of the cones for a cone tree layout [8]. The mapping on the right-hand side of Figure 3 computes the layout for a tree-map visualization [3].

Recall that these mappings will be run for every node in the hierarchy. Consider the cone tree mappings, on the left of Figure 3. The first case retrieves the children of the current node and compares the resulting list to the empty list "()". If the comparison succeeds, execution moves to the next case; otherwise, the node's *radius* attribute defaults to 2.0. The second case calls **GetNdChildren** on the list returned by **GetChildren**, thus producing a list of lists which contain the identifiers of the node's grandchildren. This case, and the next, actually compute the radius of non-terminal nodes in the cone tree. The disc at the left of the **GetNdAttr** operation in the third case indicates that the mapping should be executed recursively for the queried node if its *radius* attribute is not defined.

VANISH and VaPL also provide the functionality for marking a node in the hierarchy as the *focus*. The **SetFo-**

cus operation sets the current node to be the focus and the **GetFocus** retrieves the identifier of the current focus. When a node is set as the focus, VANISH, for each node, computes the distance from the node to the focus and sets an attribute named **distance_to_focus**. This is useful in allowing the user to navigate around the hierarchy.

For example, in a large hierarchy such as is created when visualizing the World-Wide Web, it is imperative to limit the visualization. In our Web visualization, we only visualize nodes which are less than 5 nodes from the current focus—this is specified via a VaPL mapping. By changing the focus (selecting a node), the user can "navigate" around the Web. Also, given that any node attribute can be queried by VANISH, we have the ability to selectively cull the Web visualization. As a simple example we might only display those nodes which contain keywords of interest to the user, or which are smaller than 20,000 bytes, or which contain embedded gif files.

6 Logical Interaction

One of the goals of VANISH was to ease the integration of both new semantic domains and new presentation toolkits. In order to support portability at the presentation level, an environment-independent graphics layer is a necessity. In VANISH, this mechanism is implemented as a C++ class, called **LPresentation**. This class is an abstract description of the functionality provided by the presentation layer. For each supported windowing environment a distinct subclass of **LPresentation** is created. The interface to the **LPresentation** class provides facilities for drawing lines, polygons, circles and text, as well as for controlling graphical attributes such as colour, line width and fill pattern. **LPresentation** provides two-dimensional output and all coordinates are specified as normalized device coordinates. It is a simple matter to support a three-dimensional logical presentation class as well. We have not done so because, at present, this would limit both the performance and the portability of VANISH.

Currently **LPresentation** subclasses exist for both Motif and OpenLook toolkits, under X windows. In addition to insulating VaPL and the rest of VANISH from the output primitives of the toolkit, the **LPresentation** layer provides a set of input event handling primitives, so that none of the other classes require any knowledge of the details of handling events. Because the dialogue layer, VaPL, is insulated from changes in the underlying presentation toolkits, changing between toolkits is trivial. Porting from Motif to OpenLook took 3 hours of programmer effort, and that time included porting VaPL's programming interface (so that one could not only display in either Motif or OpenLook, but could also program using the same interface under either toolkit).

7 Presentation

A very simple presentation layer provides all necessary functionality for display of the information structure being visualized. The presentation engine walks through the hierarchy, displaying nodes as specified by their visual attributes, which have been computed by the execution of one or more VaPL mappings. Because presentation primitives have been kept simple, VANISH can be ported to a wide variety of 2D or 3D graphical environments.

7.1 Visual Attributes

The visual attributes, or channels, that specify how an information structure will be displayed are identical (from a VaPL perspective) to attributes specified by the functional core. Thus, defining the presentation of a hierarchy is simply a matter of determining which attributes are to be represented in the visualization and then mapping these attributes to the available visual channels. VANISH currently provides the following visual channels (but adding more channels, or changing the definition of an existing channel is a trivial matter): Position, Colour (as *R, G*, and *B* values), Node size, Node shape (currently one of *triangle, rectangle*, or *circle*), Line style, Line width, Name.

7.2 Interaction

VANISH provides the ability to tailor user interaction in the form of a *callback* that is automatically associated with each node, and is called when a mouse button is pressed over a node. VaPL defines a **SetCallback** operation, which takes as its only argument the name of a mapping that should be called when an event occurs. After a hierarchy has been rendered, a button press causes the registered callback mapping to be executed, receiving as input a list which indicates which button was pressed, and the (x,y) position of the mouse pointer when the event occurred.

8 Results

In this section we will discuss the uses that we have put VANISH to, concentrating on what changes were required to the system to accommodate the various uses, and how much effort these changes required. These

results demonstrate that VANISH addresses important software engineering concerns in the creation of visualizations.

8.1 Integration Time

As a means of demonstrating the generality, portability, and usefulness of the separation of concerns in VANISH, we have applied it to visualizations in four distinct semantic domains: a (Unix) file-system, a revision control system (RCS), a C++ class browser, and the World-Wide Web. In order to apply VANISH to each of these domains the only changes to VANISH were the creation of new **BaseNode** subclasses: **FileNode**, **RCSNode**, **ClassNode**, and **WWWNode**.

The **FileNode** subclass uses Unix *inodes* as its unique identifiers and defines several domain specific attributes for its nodes: the user and group identifiers of the file's owner, the access, change and modify times of the file and the file's size.

The **RCSNode** subclass allows RCS log files (produced with the *rlog* command) to be loaded for visualization by VANISH. A user can visualize a large RCS project database, seeing file revision history, version branches, and so forth. These log files describe the version history of files maintained using RCS. Attributes specific to the RCS hierarchy include the per-version RCS-defined date, author, and state.

The **WWWNode** and **ClassNode** subclasses use URLs and C++ class names, respectively, as unique identifiers. Currently, these domains are visualized only structurally, as no semantic attributes have been defined for these classes.

The important fact to note about the inclusion of these semantic domains is that their integration with VANISH was extremely quick. The integration of the three subclasses of **BaseNode** took about 1 hour of programming time each.

Because the **BaseNode** class acts as a functional core adapter in VANISH, no other changes to the system were necessary to begin writing VaPL programs to visualize each of the new domains. The inclusion of attributes from the semantic domains as resources in VaPL is automatic. This means that VANISH can not only visualize static data (such as a database of RCS information), but can act as an interactive front end for any system where visualizing the results of user interaction is of value (such as a Web browser).

8.2 Re-implementing "Standard" Visualization Techniques

As mentioned above, many techniques exist for visualizing hierarchical information. With VANISH, changing the visualization technique is a matter of creating or loading a new set of mappings. One way of proving VANISH's generality as a visualization system is to show that it is sufficiently powerful to re-implement the kinds of visualizations reported in the current research literature. VANISH has been used to implement several well-known visualization algorithms: a simple 2D tree layout, two varieties of cone trees [8], "spiral" trees, and two varieties of tree-maps [3]. Each of these has been created in 30 to 90 minutes of coding time.

The technique employed to visualize structural information is orthogonal to the underlying domain being visualized. That is, one could visualize the structure of any semantic domain using any set of visualization mappings. If one wants to visualized domain-specific semantics then special-purpose mappings must be written.

To perform visualization based on hierarchy-specific semantic attributes, small changes are made to the VaPL code to bind these attributes. For example, we might want to relate a node's size in the visualization to the corresponding file's size in a file-system hierarchy (as determined by information from the **FileNode** subclass). Figure 4 shows a visualization of a small file-system hierarchy (approximately 350 nodes), visualized as a cone tree, where file size is mapped to node size and where colour is mapped to file age. Because VANISH's current presentation classes are all 2D, a top-down orthogonal view of a cone tree was implemented, adjusted so that no node completely obscures any other. VaPL mappings are used to compute the radii of the cones, to compute the angles of the arcs, and to compute the (x,y) positions of the nodes.

After implementing this visualization, the technique of graphical fish-eye views [10] was added to the cone tree layout in the following way: after the cone tree layout was computed, a callback was set which marks the selected node as the focus node. Each time the focus node changes new node positions are computed based on the fish-eye technique. The code required to compute the fish-eye node positions was minimal (19 VaPL nodes).

We have also implemented two versions of tree-maps [3] in VANISH, and used this technique to visualize a file-system. Tree-maps are a space-filling approach to visualization. A semantic attribute—in this case the size

of the hierarchy rooted at this node—is mapped to each rectangle in the tree-map. If the rectangle has children, their sizes are proportional to their relative size with respect to their parent. The example shown on the left side of Figure 5 is a file system with approximately 300 nodes. File size is also represented (redundantly) using the colour of the node.

In the other version of tree-maps which we've implemented (shown on the right side of Figure 5) we mimicked the output of the Unix utility xdu, which shows a hierarchy by drawing a series of fixed-width rectangles from left to right, with the root on the left. A given node is drawn to the right of its parent and its height varies with its size relative to its parent.

The implementation of these two forms of tree-maps took a total of 3 hours of programming time.

8.3 Non-hierarchical Data

Although all of the preceding examples of VANISH's use have been on semantic domains which are inherently hierarchical, this is not a requirement of the system. For example, in Figure 6 we have visualized a set of source files, completely disregarding any hierarchical relationship they may have within the file system hierarchy.

The visualization is a scatterplot which maps each file on the basis of the number of lines of code in the file, the number of functions in the file, and the number of times that the file has been checked into an RCS code repository. Lines of code is mapped to the y axis, number of functions is mapped to the x axis, and shading indicates the number of times that the code has been checked in. In addition, node shape is used to redundantly encode RCS information. Colours indicate the source subdirectory. In this example there are four subdirectories, and they have been assigned the colours red, green, blue, and grey. The shade of each of these colours is varied according to the number of times that the file has been checked in: darker colours indicate larger numbers of checkins. As the number of checkins goes to 1, the node colour tends toward white. In addition, node shape is used to redundantly encode RCS information. Circles indicate the bottom third of nodes that have been checked in (in terms of frequency), squares indicate the middle third, and triangles indicate the top third of the frequency distribution.

We can draw the following conclusions by examining Figure 6:

- there is a gradual increase in the number of checkins as we move from the upper left-hand corner of the visualization to the lower right. This is expected because files in the upper left-hand corner have few lines and few functions, and are arguably the least complex in the system. Files toward the lower right-hand corner have many functions and many lines, and so we would expect them to be modified, and therefore checked in, more frequently. This information is indicated in two ways: the colours at the upper left-hand corner are light, and tend to get darker toward the bottom right. This information is also apparent from the shapes found: circles predominate in the upper left-hand corner, whereas triangles predominate toward the lower right.
- files which are more complex are checked in more frequently. This can be seen by considering an imaginary line running from the upper left-hand corner to the lower right-hand corner. Files below this diagonal have relatively longer functions than files above the diagonal, and so we would expect these functions to be more complex. This is corroborated by the checkin information presented: squares and triangles predominate in the region below the diagonal. Circles predominate above the diagonal.

There is, however, one apparently anomalous point in the scatter-plot. There is one point which is substantially below the diagonal, but which is a circle. Upon examining this point, it turns out that it represents a file generated by Lex. The generated code is quite complex, consisting of few functions with many lines. However, the Lex source file is quite simple, and seldom changes. Thus the generated code file is seldom checked in.

A final note: in each of the above example visualization techniques, the implementation was relatively simple, and the interactive nature of VaPL made the prototyping of different visual alternatives for presenting information simple, and greatly eased the debugging of the VaPL code.

9 Conclusions/Future Work

The VANISH environment supports the creation of arbitrary visualizations over arbitrary domains. Our evidence for this claim comes from the fact that we have been able to create visualizations of object hierarchies, file systems, RCS databases, and World-Wide Web sites, in two different presentation toolkits: Motif and OpenLook.

The ease with which we can integrate new semantic domains and visual domains derives entirely from the underlying software architecture, based upon the Arch model. This architecture promotes appropriate separa-

tion of concerns. In particular, the creation of *virtual* presentation and semantic layers (called Logical Interaction and Functional Core Adapter) eases the task of integrating new presentations and functional cores with VANISH. Our proof of the efficacy of this software structure is that we are able to quickly integrate new presentation and semantic domains. Each integration task has taken around 1 hour.

The VaPL language also supports software engineering considerations that are important for visualization: being interpreted and dynamically typed, it encourages rapid prototyping. Being a visual data-flow language, it has no global variables, it keeps individual mappings small, and all control structures are immediately visible.

Our future work with VANISH is to extend the LPresentation class to include 3D structures and to create a LPresentation subclass for OpenGL. Finally, we expect to continue to apply VANISH to new semantic domains and to experiment with novel visualizations.

10 References

[1] P. Cox, F. Giles, T. Pietrzykowski, "Prograph: A Step Towards Liberating Programming from Textual Conditioning", *IEEE Workshop on Visual Languages*, 1989, 150-155.

[2] M. Green, "Report on Dialogue Specification Tools", in G. Pfaff (ed.). *User Interface Management Systems*. New York: Springer-Verlag, 1985, 9-20.

[3] B. Johnson, B. Schneiderman, "Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Structures", *Proceedings of IEEE Visualization*, 1991, 284-291.

[4] G. Karsai, "A Configurable Visual Programming Environment", *IEEE Computer*, March, 1995, 36-44.

[5] H. Koike, H. Yoshihara, "Fractal Approaches for Visualizing Huge Hierarchies", *Proceedings of 1993 IEEE/CS Symposium on Visual Languages*, 1993, 55-60.

[6] B. McCormick, T. DeFanti, M. Brown, "Visualization in Scientific Computing", *IEEE Computer Graphics and Applications*, 7(4), 1987, 61-70.

[7] S. Mukherjea, J. Foley, S. Hudson, "Visualizing Complex Hypermedia Networks through Multiple Hierarchical Views", *Proceedings of CHI'95*, 1995, 331-337.

[8] G. Robertson, S. Card, J. Mackinlay, "Cone Trees: Animated 3D Visualizations of Hierarchical Information", *Proceedings of CHI'91*, 1991, 189-194.

[9] M. Rochkind, "An Extensible Virtual Toolkit (XVT) for Portable GUI Applications", *Digest of Papers, COMPCON* Spring, 1992, 485-494.

[10] M. Sarkar, M. Brown, "Graphical Fisheye Views", *Communications of the ACM*, 37(12), 1994, 73-84.

[11] D. Schaffer, S. Zuo, L. Bartram, J. Dill, S. Dubs, S. Greenberg, M. Roseman, "Comparing Fisheye and Full-Zoom Techniques for Navigation of Hierarchically Clustered Networks", *Proceedings of Graphics Interface '93*, 1993, 87-96.

[12] UIMS Tool Developers Workshop, "A Metamodel for the Runtime Architecture of an Interactive System", *SIGCHI Bulletin*, 24(1), 1991, 32-37.

.

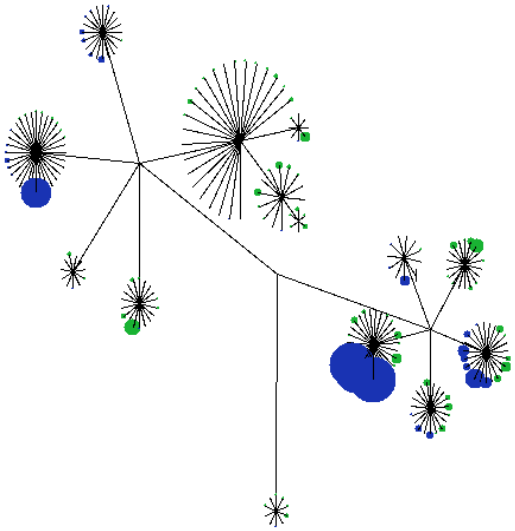

Figure 4: A Cone Tree File System Visualization

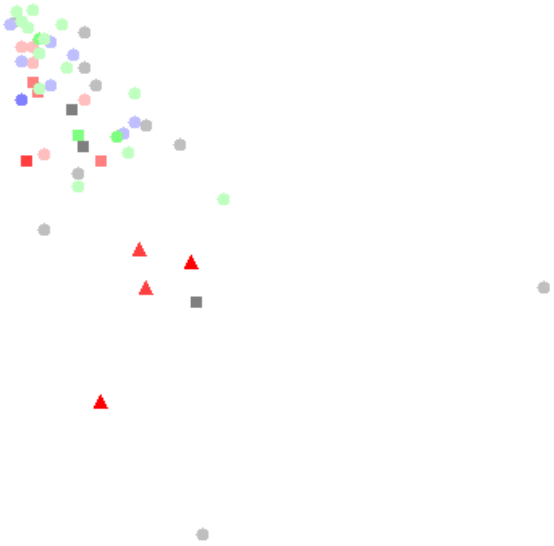

Figure 6: A Scatterplot Layout of a Code Repository

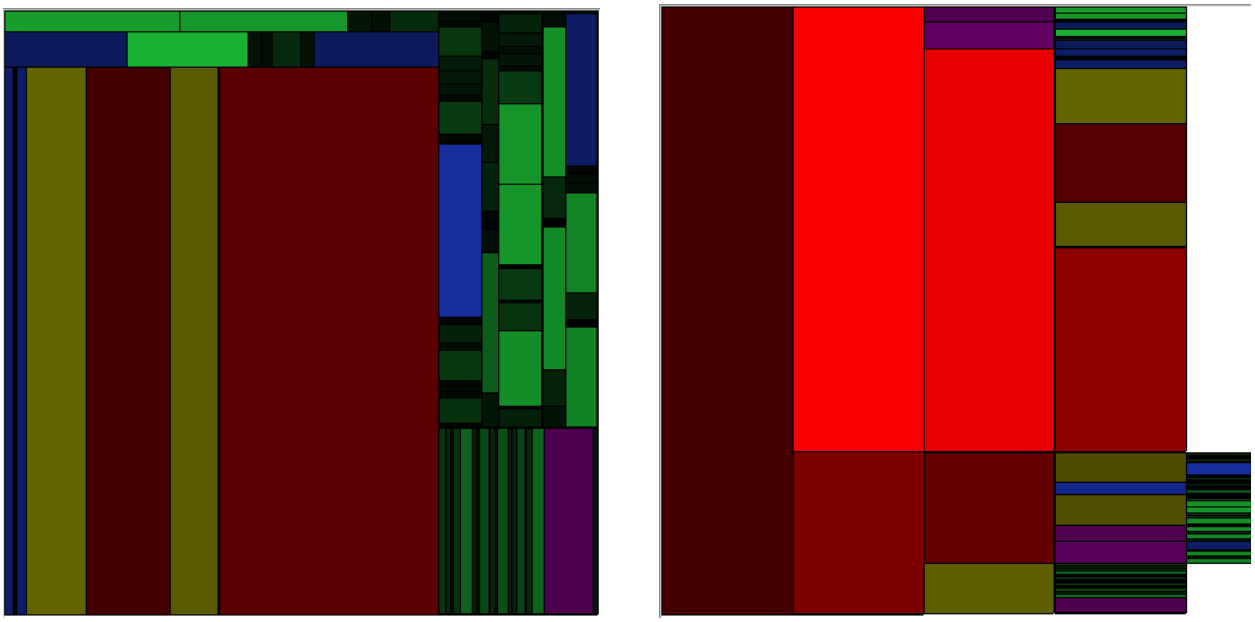

Figure 5: Tree-Map and xdu Visualizations of a File System