An Object-Oriented Design for Graph Visualization

M.S. Marshall, I. Herman, G. Melançon

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Email: {M.S.Marshall, I.Herman, G.Melancon[1]}@cwi.nl

SUMMARY

Many applications, from everyday file system browsers to visual programming tools, require the display of network and graph structures. The Graph Visualization Framework[2] (GVF) is an architecture that supports the tasks common to most graph browsers and editors. This article gives a brief overview of the design of the GVF and focuses on the core classes that are used to represent and manipulate graphs. The design of the core classes is justified by the requirements for navigation and visualization.

KEYWORDS: Graph Visualization, Information Visualization, Networks, Data Structure, Framework, Software Libraries.

1. INTRODUCTION

Graph visualization refers to the display of a set of data that has relations between elements and to the interactive facilities that make navigation through the visual representation of that data possible. Graph visualization is an increasingly important area of research within Information Visualization, both because of its everyday applications and because of the need in many fields to view large and complex networks. A few areas that have tasks involving graph visualization are website administration, network administration, computer science, biology, chemistry, medicine, and financial analysis. Although many different graph visualization systems have been built for both research and specialized applications, none of these systems has been able to meet all the requirements necessary for the analysis of graphs.

When faced with the task of analyzing a set of data with relations we are immediately faced with some of the challenges in graph visualization. For instance, we must choose a type of layout before we can draw the graph. If the graph is too large to fit on the screen, we must choose an abstract view of the graph that exposes certain types of information about the graph, yet reduces the amount of information displayed. Both choices influence what we can discover about the graph, because they determine which information is presented and how it is presented. However, our needs don't stop here; we would also like to interact with the graph, changing the view in order to gain insight into the data. All these features require a system that can easily adapt to our needs and quickly change the way a graph is presented.

In order to draw a *large* graph[2], it is sometimes necessary to reduce the data set in some way because even the best of layout techniques eventually runs out of space and pixels. This principle is thoroughly documented in the literature such as, for example, in T.R. Henry's thesis[1]. Simple filtering techniques based on attributes don't always suffice, so more sophisticated techniques such as clustering are often used. The process of clustering involves discovering groups in the data. A fundamental technique in graph visualization displays the groups, or clusters, of a graph using a special type of node called a *meta-node* to represent clusters or subgraphs in the graph. This technique makes it possible to represent a graph by displaying fewer elements, allowing the user to control the level of detail by "opening" and "closing" meta-

---

[1] Guy Melançon can now be reached at Guy.Melancon@lirmm.fr .

[2] Available at http://www.cwi.nl/InfoVisu/GVF

[3] We refer to *large* graphs as graphs containing more than a thousand elements (nodes + edges). In contrast, we will somewhat arbitrarily define a *huge* graph to contain 10,000 or more elements. See Section 2.1 for details.

nodes. Such an approach requires a way to store and manipulate graphs whose nodes may represent subgraphs.

Users wishing to explore a graph must be able to manipulate it, in addition to navigate it. A user should be able to interactively apply criteria that result in different sets of clusters within the data. In fact, the user should also be able to apply clustering proccesses to subgraphs or clusters resulting from previous activities — a process called *hierarchical clustering*. The process of clustering creates additional structural information about the graph that the user will want to compare with other processes. These types of activities demand a system that can store multiple levels of complex information about the graph, such as the auxiliary structures resulting from clustering, yet keep the original structural information about the graph intact. We will call a graph with such cluster information a *multigraph*[4], or a graph where a node may be shared among multiple graphs. In such a clustered graph, each node belongs to the original graph and may also belong to certain subgraphs that result from clustering. In the case of hierarchical clustering, the node may belong to a series of nested subgraphs. If information about several clustering operations is stored, a node may belong to subgraphs or clusters, each resulting from a different clustering proccess. As we shall see in Section 2.3, multigraphs require a different kind of design than simple graphs.

Other tasks besides clustering make support for constantly changing graphs necessary. For example, the ability to edit a graph can be an important part of a graph visualization system. A user experimenting with layout may want to add or delete parts of the graph to see the effect it has on a particular layout, or simply edit the properties of a given element to see the resulting effect. Systems that update graphs with real-time information also require a way to handle constantly changing structures. All of these tasks make support for dynamic graphs an important requirement for graph visualization systems.

In 1999, our research group at CWI started looking for an environment in which it was possible to experiment with a variety of graph visualization techniques. The experimentation would include the interactive definition of nested clusters and the use of visual elements to represent graphs and their properties[5]. Although we have looked at other class libraries for graphs in addition to many complete systems[4], none fulfilled our needs and so we decided to develop our own. We wanted to create a system that is general enough that it can be embedded in information visualization applications but also be used to create a standalone application. Because our goal was also portability and ease of maintenance, we decided to implement it in Java. Although the development of the Graph Visualization Framework (GVF) was done in Java, we believe that the solutions we have found are of general interest for object-oriented programming.

2. THE FRAMEWORK

2.1. An Overview

Although the goal of this paper is to describe the graph class library rather than the entire framework or our particular application, we will describe some general aspects of the GVF and our application. We have built an application called “Royere” based on the GVF. Royere is used by ourselves and others as a testbed for experimentation with graph visualization. Royere has been used to visualize citation graphs, phylogenetic trees (from biology), database structures, web structures, and randomly generated graphs. Royere currently reads several file formats including GML[5] and GraphXML[6], can print and can export to PNG, BMP, JPG, and SVG[7] formats. Several types of layout have been implemented in Royere including Reingold-Tilford, radial, and a barycentric layout. It is very easy to add a new layout algorithm and reasonably simple to add new functionality such as a parser, or even an entire new View Module (e.g., this has been demonstrated by our creation of viewers that use Java3D (not supported in the current application), OpenGL, and Java2D as graphic engines). It is possible to have several different views of one or more

[4] We do not use the term as it is used in graph theory.

[5] Some earlier reports[2, 3] describe the kinds of developments that interest us.

graphs, each in its own window. Several types of interaction such as zoom and pan, as well as fisheye[8] are available, as well as selection and one type of linked view.

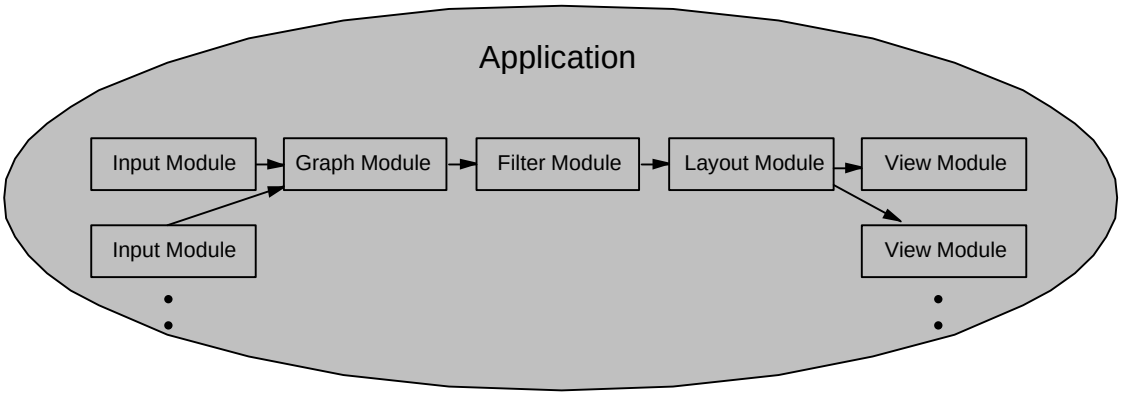

Figure 1. A possible configuration of GVF modules

The largest software components in the GVF are called *modules*. Each module has its own thread of control that cooperates with other threads through a JavaBeans-style notification mechanism[9]. These modules correspond roughly to the functional modules described in the AVS[10] and the information visualization pipeline described by Chi et al[11] (modified version of the Stuart Card's model). The modules can be assembled into a graph visualization application in which each module corresponds to one stage of processing in a pipeline of data flow as in Figure 1. They communicate with one another by sending messages through logical pipelines established by the application. Some user actions will cause the application to start a process early in the pipeline, which can then propagate changes to the View. Because of the notification mechanism and multithreading, the modules could be distributed across processors although we have not yet implemented such a distribution. There are two possible motivations for such a distribution. One is to perform CPU-intensive operations such as layout or graph property updates on a different processor for smoother interaction. Another scenario distributes View Modules to different machines for multi-user collaboration or additional screenspace. It is also possible, of course, to use the modules independently. For example, the Input Module, Graph Module, and the Layout Module have been used to perform layout measurements on sets of randomly generated graphs[12], in a process that does not involve drawing (drawing takes place in the View module).

It is possible to optimize several different aspects of a graph visualization system in the implementation. For example, interaction can be optimized with index structures like quadtrees for speed in picking nodes and edges. As another example of optimization, our application creates a flat list of elements for drawing, which speeds redraw. Many similar types of optimization are possible in an implementation built on GVF but most of these optimizations must take place in the View Module. In this paper, we will focus mostly on what takes place in the Graph Module and how it supports the tasks in the other modules.

The practical size of the graphs that allow real-time interaction by our application is roughly limited to fewer than 10,000 elements on an 866Mhz Pentium machine running Windows 2000 with the Java2D version of the View Module (the OpenGL version of the View Module is still generally faster at drawing). Note that this limit is based on the number of items *interactively drawn*, i.e., items that must be redrawn during a screen refresh. The level of interactivity desired establishes one of the thresholds in graph size that we could call the *interactive limit*. We have witnessed a rapid improvement in the drawing speeds of both hardware and software in the last few years, which has pushed the interactive limit higher. The second, usually higher, *physical display limit* is affected by more absolute constraints, such as screen resolution, which make it physically impossible to draw more than a certain number of elements of a graph. Less well understood cognitive and perceptual limits of the user also limit the practical size of the graph. Therefore, the result of drawing a graph beyond a certain size is simply lots of color on the screen with no discernable structure. Beyond a size threshold that depends on the graph, layout, task, and the display device, it makes no sense to draw the entire graph and we must use entirely different approaches. One such approach is discussed in [13].

As basic support for large graphs, the GVF provides a way to extract the interesting parts of the graph before layout. This can be accomplished by associating a filter or clustering object (see the Filter Module in Fig. 1) with a graph. A `Filter` object is defined to take a `Graph` object as input and produce a `Graph` object as output. Referring to Figure 1, the graph object that arrives through the pipeline at the Layout Module isn't necessarily the original graph but can be a filtered version of the graph. Likewise, the layout algorithm can choose which parts of the graph to show (see Figure 4), reducing the number of elements to be displayed even further. Another approach to filtering is to filter the graph in the View Module. This filtering is applied to the elements that have been handed to the View Module by the Layout Module. This technique is described in [3] and works best on graphs that are below the interactive limit in size.

Although our current application hasn't implemented all the possible techniques available for the exploration of *huge* graphs, the design of the GVF doesn't preclude their use. More sophisticated approaches to the scale problem such as the incremental layout approach found in Nicheworks[14] or the moving logical frames approach described by Huang et al.[15] should be possible to implement in an application built using the GVF. The same holds for the spanning tree extraction and hyperbolic 3D approach used in H3[16].

2.2. Properties and the Graph Classification Problem

In an application of the Decorator Pattern[17], most objects in the GVF can have properties and several mechanisms using properties have been put into place. Properties allow the association of a *value* with a *key* (in what is sometimes called a "dictionary"). Although properties are not unique to the GVF and numerous other systems use them, a description may lend valuable insight into the application of these mechanisms and aid in understanding how the core classes fit into the GVF.

One of the challenges faced by graph system designers is the graph classification problem. How can we design an object hierarchy for the many types of possible graphs? Unfortunately, graph types do not fit neatly into a hierarchy; they are classified according to the various combinations of properties that they have. Multiple inheritance solves part of the problem but doesn't prevent the proliferation of classes, nor potential confusion about class names. For example, a graph with the property "directed" and "acyclic" could be classified as either a DirectedAcyclicGraph or AcyclicDirectedGraph. If the same graph is also "connected", then it is named with a combination of "Connected" "Directed" and "Acyclic". Creating objects for each combination of properties in this manner causes the number of classes to increase exponentially with the number of properties. Even if it were possible to design, such a class type hierarchy would be impractical for dynamic graphs: a single edit of an edge may change the graph's type (e.g. from acyclic to non-acyclic). Such a change would necessitate a dynamic change of the class type of a graph instance, which is impossible in Java or C++. Some researchers have suggested delegation-based systems to overcome these types of limitations[18, 19]. However, our solution to this problem is to use properties to categorize graphs. Properties add a level of dynamism to objects that isn't otherwise possible in Java or C++. Adding a new property to a class instance could be considered functionally equivalent to adding a new field to the class specification. In the case of graphs, the use of properties makes it unnecessary to design a special type hierarchy for the many different types of possible graphs. Rather than instantiate a new type of graph object and transfer information when a graph property changes, we can simply change the value of the affected properties. This practice makes a hierarchy of graph types unnecessary in the GVF. Instead, implementors can use a single class to deal with all type of graphs.

The most important graph properties used by the GVF are boolean properties, like "isDirected" and "isAcyclic". Some other available properties are "isFreeTree", "isFreeForest", "isTree", and "isForest"[6]. These properties can be used in a simplified version of a constraint-based negotiation mechanism described in [20] to implement dynamic menus. For example, some layout algorithms will only work with particular types of graphs, i.e. graphs with certain properties. The negotiation mechanism allows us to decide at runtime which layout algorithms are made available on the menu for a particular graph view, depending on the graph's properties. Each layout algorithm can express its requirements as a boolean expression of

[6] Planarity testing is impractical for many large graphs typically found in information visualization applications. A planarity property is not used by our current implementation but could be useful after reducing the size of the graph, as described in Section 2.1.

property comparisons that is evaluated at runtime. This expression is contained in a `Requirements` object. Requirements for a `Layout` object that aren't expressible with the operators and properties available can be implemented by overriding the `meetsRequirements` method with customized logic. The same mechanism aids in deciding which layout to use when a graph is initially loaded for viewing. Such a mechanism makes it possible to add a layout algorithm to the application by defining a single class. The availability of several different types of objects, including layouts, filters, and metrics, can be controlled by entries in a preference file. Note that this dynamic loading is made straightforward by the reflection mechanism of Java, which makes it possible to load object instances based on the name of their class.

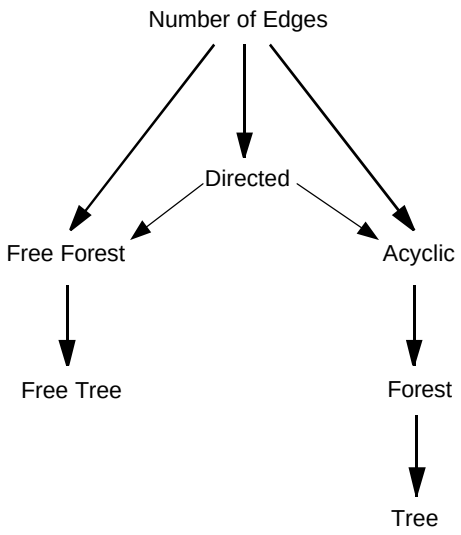

Figure 2. Property dependencies

Because some properties depend on others, it is sometimes necessary to update several properties when a particular property's value changes. To aid in such interactions, we applied the Observer Pattern[17]: class instances can express their "interest" in the change of a specific property by registering themselves. If the property changes, the registered observers (called *listeners* in Java) are notified and can act appropriately. The property change listener facilitates the update of property dependency chains. For instance, in the case of graph properties, if the property "isAcyclic" changes, a registered listener should check to see if the property "isForest" now applies.[7] Likewise, if the property "isForest" changes, then a registered listener should check to see if the property "isTree' applies. This system is employed to ensure that the properties given to a graph remain consistent with the graph during real-time changes to the graph such as those brought about by editing. Consistent properties are crucial to many routines. For example, a layout routine that draws a tree will not work properly on a graph that isn't a tree.

Another use of properties within the GVF is commonly found in many applications that have user preferences. Application behaviors such as how a node is displayed or which type of layout to use can be influenced by the property value that is stored at various levels throughout the application. A simple example is the color of a node. The preferred color for a node can be set through the corresponding visual property located either at the `Node` object, in the containing `Graph` object, in the containing View, or in the application itself.

In another application of properties, we can use the Factory Pattern [17, 20, 21] together with the negotiation mechanism to generate objects, such as specific types of graphs. The negotiation mechanism ensures that the objects are produced with the requested properties but according to user preferences and within system constraints. In this way, it is possible to precisely determine the properties of the objects that such a factory produces at runtime, possibly even making use of information about system performance. A side benefit of using the Factory Pattern is the ability to easily incorporate *object pooling*[8] for the type of object being supplied.

2.3. The graph as a node

Figure 3 shows a simplified class diagram of some essential classes in the GVF. The `Element` object is the most fundamental object of the core classes. It is meant to capture the features common to the two types of

[7] Forests require the "acyclic" property to be "true".

[8] Object pooling refers to a type of memory management that "recycles" objects in order to avoid calls to the constructor (`new` is one of the most expensive calls in Java). See [22] for details.

elements in a graph: nodes and edges. An `Element` represents data before any relational information has been added to it and can therefore represent a generic data element in a data visualization application. All `Element` objects can have properties assigned to them. An `Element` can also have data (i.e. the data that the graph is representing) associated with it, along with any information necessary to view it. For instance, a node (that is also an `Element`) may represent a web page that can be rendered by a specified browser. Elements can be grouped together, as in a graph or a selection, and we can ask an `Element` about the groups to which it belongs. All of these features are available to nodes, edges, and graphs through inheritance.

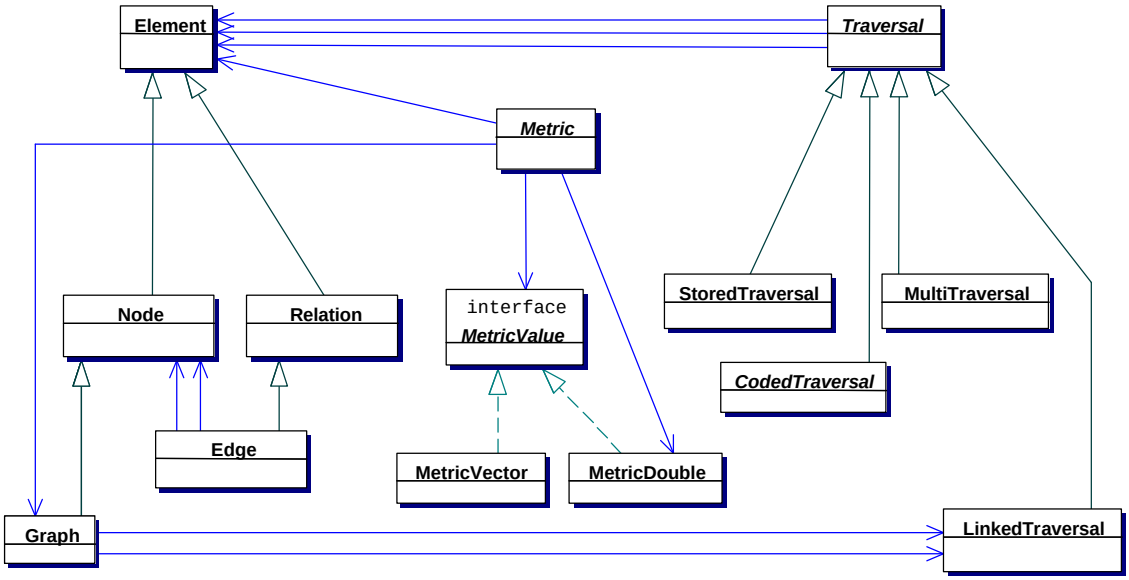

Figure 3. A UML diagram of the most essential classes in the GVF

Some features of the hierarchy in Figure 3 might be surprising at first glance. For example, a designer might be initially inclined to define a `Graph` object as a direct translation of the definition of a graph: a collection of nodes and edges. A `Graph` object, in this case, would store a list of all the nodes and edges that belong to it. Similarly, one might be tempted to define nodes in a graph as a direct container of references to its neighbors, in the form of object references directly represented as fields in the `Node` class. However, this simple approach does not work in view of the requirements that we discussed in the introduction. The use of multigraphs and their dynamic nature requires a somewhat more sophisticated approach that will be outlined in what follows.

In dynamic graphs, where clusters are constantly created and disbanded, and where edits or updates may change a graph, flexibility is of the utmost importance. Not only support for dynamic graphs but also support for multigraphs is necessary. Multigraph support is especially important for clustering. During clustering, we often introduce new relations into the data, both between elements of the original graph and between the clusters themselves. We would like to build clusters and store the induced relations, yet retain information about the graph that contains them.

Another place where multigraphs are needed is during layout. Often, layout algorithms extract a subgraph (for example, by temporarily disregarding certain edges) and perform layout on the extracted elements, replacing the missing elements in the view. In this case, it is advantageous to regard the elements as belonging to both the original graph and the extracted graph, i.e. multiple graphs.

In all these examples, there is one thing in common: whereas the set of nodes and edges may be constant, a graph could be considered as a special “view” of these entities, and a system might need several such “views” concurrently on the same set of entities.

In the GVF, we store the structural information about a graph in the `Node` object. In other words, the `Node` object can be queried about the edges that it has *in a particular graph context.* This makes it possible

for a Node to belong to multiple graphs while retaining a unique identity. For example, the following lines could be used to find out what edges NodeA has in Graph1 and then Graph2:

```
edgeSet1 = NodeA.getEdges(Graph1);

edgeSet2 = NodeA.getEdges(Graph2);
```

Notice that the Node object is queried with the graph context as a parameter. More importantly, the node can change graph membership by simply changing what is stored in the Node object; there is no need to find and edit an entry for the Node in lists maintained in the containing graphs. The node-centric model of the GVF makes this possible.

Another motivation of our project besides the ability to handle multigraphs is the need to treat a Graph object as a node when it is being used to represent a subgraph or cluster. Because many of our Graph objects will be treated as meta-nodes, the Graph object is defined as a subclass of the Node object. This definition makes several things possible. In this way, a Graph can be handled as a Node when it represents a subgraph, which simplifies the implementation of nested graphs. Furthermore, a Graph can have a relation with another Graph or Node, making it possible to express one-to-many relations and relations between clusters.

```
Graph bigGraph = new Graph();

Graph cluster1 = new Graph();

Graph cluster2 = new Graph();

cluster1.add(Node1);

bigGraph.add(cluster1); // cluster1 is treated as a node in
bigGraph

bigGraph.add(cluster2); // cluster2 is treated as a node in
bigGraph

Edge edge = new Edge(cluster1, cluster2);

bigGraph.addEdge(edge);
```

Several features of Element can also be used by the Graph object. For example, a Graph object can have properties just like any other element. Also, a graph can have data associated with it.

The alert reader might now ask why we haven't used the Decorator pattern here again instead of static subclassing to accomplish nesting. Afterall, the GVF uses the Decorator pattern for graph properties. Couldn't we make the nesting of a graph inside a node another (dynamic) property of a node? There are several reasons why we do not. In the case of graph properties, the benefit of using the Decorator pattern outweighs the costs of programming "outside the language" i.e. decorating it. One disadvantage is that we don't get type checking or compile time checking for our graph properties – we must rely on good programmer behavior. Another is that implementors must find the documentation that defines the semantics of these properties. These disadvantages are outweighed by the benefit that we have only one class for a graph instead of a confusingly large number graph classes. However, in the case of nesting, the benefits of applying the Decorator pattern are unclear. However, there are several benefits to statically subclassing from Node. We get compile-time checking and are able to include the several methods that are specialized to Graph without complicating the Node object. The fact that we often would like to represent a subgraph as a node also means that this particular subclassing fits our model. A graph is, in this case, a specialized form of a node[23].

The discussion until now has only used the term "edge" to refer to the relations between nodes. The word "edge" implies a binary relation. However, *hypergraphs* are also important in some application areas. They involve relations between *sets* of nodes that can be best represented by tuples. The Relation object in the GVF stores such a tuple (and degenerates in the simplest case to a binary relation or edge). The

`Relation` object makes GVF potentially useful for systems that use hypergraphs such as VLSI and PCB design[24] while remaining inconspicuous to implementers concerned only with binary relations.[9]

2.4. Traversals

During the manipulation of graphs, it is often necessary to visit a set of nodes, graphs, or edges, or a mixture of these elements. The order of the visitation may be important for a particular application, such as a traversal that describes a path from one node to another. In the GVF, a `Traversal` is an object that can produce a set of `Elements` in an order that describes the visitation of a graph. The `Traversal` object is meant to provide a single interface to the application programmer for the visitation of a graph. The intended method of access for a `Traversal` is through the Iterator Pattern[17], familiar to many readers through the STL library of C++ and collections in Java 1.2. The `Iterator` allows the user to iterate over a collection of objects without requiring special knowledge about the contents of the collection. Because different circumstances require different implementations of the `Traversal`, the factory pattern is applied to produce the type of traversal that best suits the situation.

It is important to note that the iterators supplied by `Traversals` return `Element objects`. In this way, visits of both nodes and edges, and even graphs, can be included in a single traversal. This provides us with a unified and comprehensive approach to graph traversal. This is especially convenient during the layout of several connected clusters, where we would like to treat each cluster as an `Element` along a traversal:

```
Traversal clusterTraversal =
          theTraversalFactory.create("BreadthFirstSearch", graph);
Iterator clusters = clusterTraversal.iterator();
while (clusters.hasNext()) {
  Node cluster = (Node)clusters.next();
  ...
}
```

In general, the `Iterator` should make the implementation details of most traversals irrelevant to the user. However, the user may occasionally want a special type of traversal that isn't available from the factory and find it necessary to write her own. An explanation of how the Factory Pattern is used for Traversals should clarify its value and help the reader understand what is involved in implementing a traversal[10].

The `Traversal` is implemented in several different ways. A `Traversal` can be statically defined or stored, or can be dynamically defined by the logic that implements the iterator. The choice among implementations is made by the factory and can be determined by the tradeoff between speed and storage. If the factory has previously built and stored the requested traversal, it simply returns it. However, if the user requests a traversal that hasn't been built yet, the factory may return a traversal whose iterator is dynamically defined. Rather than throw away the traversal information, this iterator stores the returned elements in the traversal while it is being used. The next time that particular type of traversal is requested for that particular graph, the cached version can be supplied. Of course, more sophisticated logic may be used by the factory. For example, the factory could consider several factors such as available memory, probability of a particular type of traversal being used again, and so forth. In some situations, it may be useful for the user to indicate to the factory that the traversal will only be used once and this information could be used to help decide what sort of object to return. As with most user interactions with the factory, the preferences can be contained in a `Requirements` object.

---

[9] Hypergraph relations have not yet been exploited in our visualization environment, and are the subject of future work.

[10] The traversal factory can also make use of reflection to dynamically load new traversal types as is done in other factories.

One stored implementation of `Traversal` is called `LinkedTraversal`. The design model for the `LinkedTraversal` is very similar to the model for storing graph information at the `Node` object except that the storage for `LinkedTraversal` is in the `Element` object. In the `LinkedTraversal`, the elements can be considered part of a "switching network" in which each element points the way to the next element. In such a traversal, we may also decide to change to a second traversal when we are part way through the first one (this requires access to the actual `Traversal` object rather than only the `Iterator`). However, the implementation of `LinkedTraversal` is constrained because it cannot store a path that crosses through the same element more than once. Another stored type of `Traversal`, called `StoredTraversal`, is less dynamic but faster and more storage efficient. The `StoredTraversal` is implemented as an `ArrayList` of `Elements`, which makes iteration faster.

In order to build most types of traversals, we must be able to visit all the nodes of a graph. When a graph is being initially built, a special `LinkedTraversal` called a `nodeTraversal` is simultaneously built that visits all the nodes. It is easy to demonstrate that if such a traversal were implemented with an array, changes to the graph would require more work to update the `nodeTraversal`. This can become a crucial issue when dealing with large dynamic graphs. Similarly, property change listeners can be employed that ensure that traversals stored by a factory are updated as a result of graph edits.

The `LinkedTraversal` was designed with dynamic graphs in mind. One of the advantages of the `LinkedTraversal` is that during deletes, we can ask an element which traversals it belongs to and then repair those traversals. This way, a `LinkedTraversal` can be updated when changes are made to the graph. In a list model, each traversal would have to be checked after each deletion to see if it contains the element being deleted.[11] Another potential advantage of the distributed model is the ability to use parts of existing traversals to create new ones. Changing a given traversal is also quite simple. However, for certain temporary tasks the `StoredTraversal` object is better because it stores the traversal in a single local object that can then be returned to the system after use. The `StoredTraversal` access and iteration times are also faster than that of `LinkedTraversal`, where each access and iteration is accomplished through a `HashMap` object at the element. In the case of our current implementation, comparing `StoredTraversal` access and iteration to that of `LinkedTraversal is simply a matter of comparing the speed of those operations for their respective Java implementations, i.e. ArrayList` vs. `HashMap`.

The implementation of `Traversal` where the returned elements are dynamically defined is called `CodedTraversal`. The `nextElement`() returned by the iterator of a `CodedTraversal` is defined by logic in a particular implementation rather than a stored order. This makes the `CodedTraversal` more storage efficient than fast. It can be used to implement a state machine definition of a traversal. In a situation where traversals are used on a very large graph, it may actually be necessary to use the `CodedTraversal`. For a very large graph, it may be impossible to store the order of several traversals without running out of memory. However, not all traversals can be implemented as a `CodedTraversal` because the definition of some types of traversals requires storage anyway.

The last class in the toolkit of traversals is the `MultiTraversal`. A `MultiTraversal` can be used to build a traversal from several other traversals. The `MultiTraversal` can transparently manage iteration across all the traversals that it contains. In cases where the user has collected several paths that, together, span some part of the graph, a `MultiTraversal` may be used to follow each path successively until all elements have been visited.

2.5. Layout

Layout is an important part of viewing a graph. Different layouts reveal different information about the data so it is important to have a variety of layouts for different tasks. Although layout uses information about the graph structure, it is not part of the graph itself but, rather, it is part of a view of the graph. A designer might

---

[11] However, an addition will invalidate most traversals associated with a graph in either model except for `nodeTraversal`.

be initially inclined to simply store the coordinates for a given layout in a class variable. However, such a design doesn't support the storage of the coordinates from a layout for re-use. Nor does it support concurrent layout, where a graph is drawn with different layout methods in parallel. Neither does it support the storage of the same type of layout for different graphs in which the node is a member. For instance, this would necessitate running a new layout procedure every time the user selects a different layout or a different graph.

In the GVF, the coordinates of a `Node` or `Edge` are stored in the `Layout` object. These coordinates are associated with a particular graph and can only be retrieved by providing the relevant `Element` and `Graph` objects as parameters. With the dual mapping of coordinates using both element and graph as keys, we accomplish two things: it is possible to store a particular type of layout for several different graphs and it is possible to store multiple layouts for a particular graph. This allows us to have multiple graphs, each with a different layout and seamlessly switch between the layouts without waiting for a new layout process to finish. Furthermore, layout is often a costly process that we would like to have done in the background, if possible. Because the layout process is independent of the view, we can perform a different layout in the background on the same graph that is being viewed.

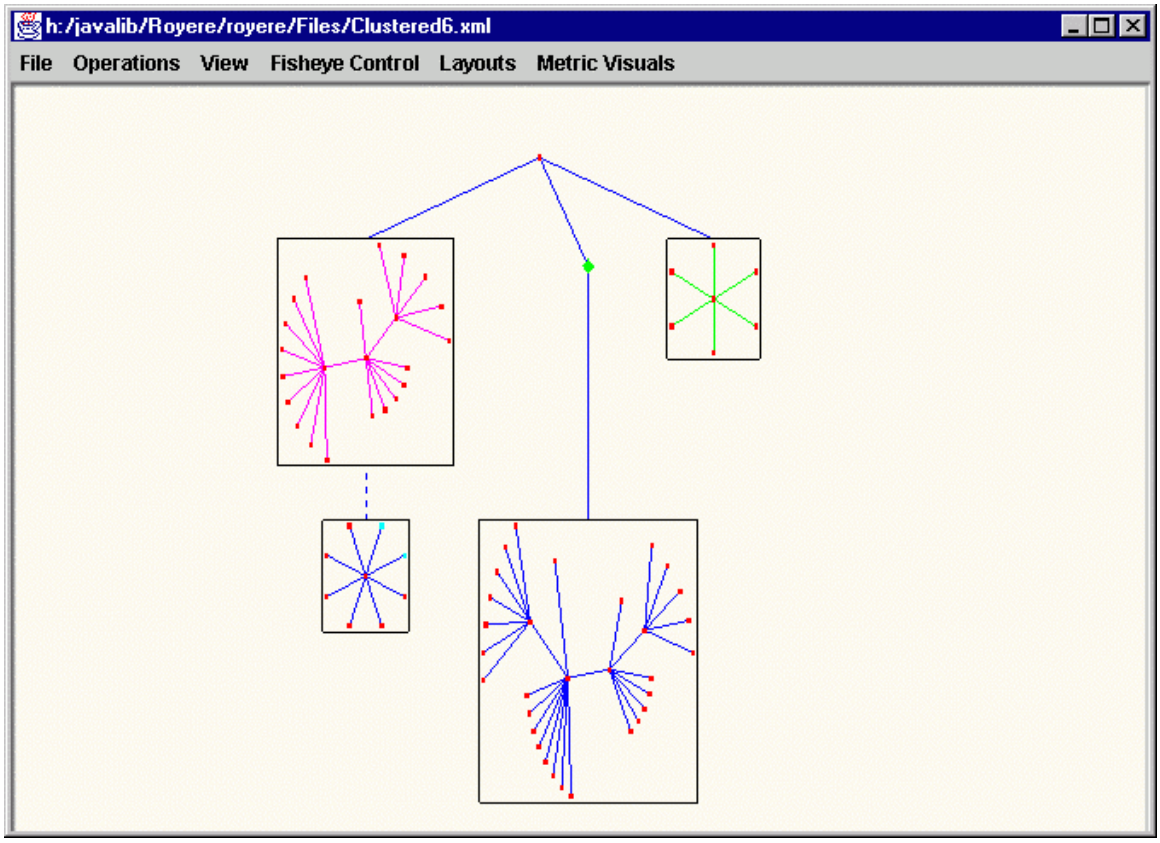

Figure 4. A nested layout with mixed styles of layout

One of the advantages of the way the coordinates of a layout are accessed can be illustrated by the following example. While building a view, we may take advantage of the way that coordinates are stored and show a mixture of several layouts. The mixture can be of both the scaling and type of layout. For instance, when encountering a meta-node during a traversal of a clustered graph, we may choose to show the nodes in the subgraph (depending, of course, on the preferences for that view). The nodes in the

subgraph may have the same layout type as in the supergraph or a different type. As an example, we may have a tree laid out using the conventional hierarchical Reingold-Tilford algorithm, but a subtree laid out using a radial algorithm (see Figure 4). The negotiation mechanism that we mentioned previously can be used to determine the type of layout based on the layouts that are possible for the given graph type and the user preferences.

2.6. Metrics

As with any form of analysis, we need a set of measures that we can apply to graphs. These measures allow us to find interesting features in the data by mapping the measures to visual attributes or using them for control. Many graph theoretic properties are well-known measures to those familiar with graph theory. The degree of a node, which is equal to the number of edges connected to that node, is a simple example. Although most algorithms rely on conventional metrics, such as the degree of a node or other metrics based on distances in a graph, metrics can also be defined by the composition of other metrics.

Certain measures or metrics are useful for providing visual cues when rendering a graph[2, 3, 25]. One technique renders an edge with continuously shaded color that reflects the metric values of the nodes at its endpoints. The overall effect is the emphasis of certain edges in the graph. In another technique, the metric value for a node determines whether the node and its corresponding edges are displayed. When applying these techniques, different metrics create different effects on the view of a graph and emphasize different types of information. Visual cues aren't the only way to use metrics. For instance, some types of layout algorithms use metric values to aid in positioning[26, 27]. Metrics are also essential to clustering operations, during which some type of value must be assigned to an `Element` in order to decide its cluster membership. Each different area of application may make use of metrics and these metrics take on many forms.

A simple approach would be to make a class variable of type `double` to store a metric value at the `Node`. However, this option is limited and supposes that the set of necessary metrics and their type is known ahead of time. One of the challenges in the design of metrics was to provide a uniform approach to defining a metric, which could be of any algebraic type and could be dynamically created. In order to support multiple metrics with unknown algebraic properties, we must have a set of mappings to objects that implement the necessary operations. Therefore, in the GVF, we have made it possible to define a `Metric` object and associate it with an `Element` (node or edge). There may also be several different metrics for any given `Element`, each one associated with a particular graph that contains the `Element`. In order to support such associations, we must provide a dual mapping based on the metric type and the graph. We may retrieve the `Metric` object for a given `Element` by providing both the name of the metric and a graph context.

The class hierarchy for metrics is divided into two separate branches under the top-level objects `MetricValue` and `Metric`, to clearly separate the set of operations that can be carried out on the metric values from the algorithms we must implement to compute the value associated with an element in a graph. In an attempt to keep the possible applications open, we have made `MetricValue` into an interface that defines several operators for manipulation and comparison. Only the basic operations (addition and multiplication by a scalar) necessary to compute the minimum, maximum and average values of a metric are required. In its current form, the GVF offers two classes `MetricDouble` and `MetricVector` that implement the `MetricValue` interface using `double`'s and a finite dimensional vector of `double`'s. The `MetricDouble` implementation offers additional operations specific to real numbers. Of course, it is also possible to implement the `MetricValue` interface with a different internal representation. The `Metric` class can be thought of as the class containing the knowledge of how the metric should be computed. It stores the value of its associated element through composition with `MetricValue` in a field called `metricValue`. The `Metric` also contains methods to store properties (such as its name) and requirements that must be satisfied before the metric can be created and computed. Just as with the `Layout` object, these requirements are stored in a special `Requirements` object that contains a boolean expression of property comparisons that are evaluated at runtime.

The example below shows how a simple metric can be implemented with the use of a sucessor iterator. In the example, the metric is meant to be applied to a tree (or more specifically, a forest) where the width of an element is defined as the number of leaves that are reachable from a given node. The restriction to trees would be expressed in the requirements of the metric. The value can thus be obtained by iterating over the node's successors in the tree and summing up their values (except for leaves for which this sum is 0.0; it is set to 1.0 by taking the maximum of the sum with the value 1.0).

```
Iterator successorIterator
        = ((Node) this.element).getSuccessors(this.graph);
MetricDouble sumMetric = new MetricDouble(0.0);
while (successorIterator.hasNext()) {
  Node successor = (Node) successorIterator.next();
  Metric successorMetric
        = successor.getOrMakeMetric(WidthMetric.Name, this.graph);
  sumMetric.add(successorMetric.getOrCalculateValue());
}
sumMetric.max(new MetricDouble(1.0));
metricValue = sumMetric;
```

Observe that the metric object associated with an element can only be obtained through the `getOrMakeMetric()` method to avoid creating duplicates. In the same manner, the value of a metric is requested through the `getOrCalculateValue()` method to ensure that the value is computed only if necessary. As with Traversals, property change listeners can be employed to ensure that metrics are updated or discarded as a result of changes to the graph that they measure.

3. COMPARISON WITH OTHER SYSTEMS

Although many graph visualization systems exist, most of them have been developed for specific tasks rather than as libraries for general application. However, a few general libraries are currently available and we will compare some of them to the core classes of the GVF here. Our comparison focuses on the functionality afforded by the design rather than the feature set of the applications that have been built using them. It is important to note that all of the systems in the comparison are works-in-progress and that our comparisons have been made based on the available documentation for these systems. Indeed, because of a lack of publicly available documentation, we were unable to examine certain other systems in the detail necessary for a meaningful comparison with the GVF. Proprietary systems from Bell Laboratories[28], Tom Sawyer Software[29], and JViews[30] are examples of systems that we might have otherwise included in this comparison. Another system that we haven't attempted to compare is the widely used package of graph visualization tools from AT&T[31]. This comparison is therefore not a complete survey of systems.

The systems that we investigated have several features in common with the GVF. All the systems are written in an object-oriented language and implement graph traversal with a storage object and some form of iteration. Most systems have a way to associate data or properties with either a node or an edge, although explicit support for metrics was generally absent. Most systems are also capable of specifying the visual attributes of elements and provide several choices for layout.

EDGE[32] was the first object-oriented graph editing toolkit to be published and seems to exemplify the design goals of many future systems, including several mentioned here: to provide a portable and extendible graph editor that can be adapted for use with many types of applications. EDGE was written in C++ with a graphical user interface based on the X window system. A graph description language (GRL) allows storage and data exchange of graphs and their attributes. A noteworthy feature of GRL is the ability to specify constraints that EDGE can use to affect layout. Another unique contribution of EDGE is the use of an (albeit expensive) algorithm to reduce the number of edge crossings by redirecting edges through "edge concentrator" (i.e. dummy) nodes. However, customization of EDGE requires recompilation and, in the case of a menu extension, the use of a "program generator". Such extensions are simplified by the

reflection mechanism in Java, which makes it possible to make components available via a menu in Royere by compiling a single object written to the GVF API and adding the object location to the preferences file.

Some libraries, such as AGD[33] and GDT[34] have impressive collections of algorithms but focus on tasks such as layout rather than interaction. However, we will discuss the GDT in the context of the solution that they provide to the graph classification problem presented in Section 2.2. The GDT is based on a set of libraries for combinatorial computing called LEDA[35, 36] and is written in C++. In [19], Pizzonia and Battista propose extensions to C++ called ECO C++ (Extender and Classer Oriented C++) that implement a delegation-based system. The design, which is applied within the GDT, overcomes the problem of creating a class hierarchy for dynamic graphs but requires a pre-compiler for the language extensions. The experience some of us have had with similar precompilers [18] confirms that using such tools makes it difficult to deploy such systems on a larger scale. Furthermore, the implementation of graph algorithms using the GDT requires familiarity with a large and specialized graph class hierarchy.

LINK[37] was developed as a portable tool for interactive graph manipulation and computation. It has an object-oriented Scheme command-line interface, with direct access to the Tk graphical user interface and an underlying set of C++ libraries. Functionality for types of manipulation common in discrete mathematics is built on Collections and Iterators. *LINK* is successful at providing a high degree of flexibility and interactivity. One of the biggest drawbacks to *LINK* is probably its speed. According to the authors, loading a few thousand objects for a graph view took several minutes on a Sparcstation 5 in 1997. Although it is unclear if datasets of such size still present a problem to the latest version of LINK, datasets of comparable size do not present a problem in Royere for either loading or interactivity.

The Graph Foundation Classes (GFC)[38] for Java became available from IBM's alphaWorks site in March 1999. The GFC contains several ideas in common with the GVF and is implemented in Java, although the current implementation is limited to JDK 1.1 features, which lacks, for example, the collections library of JDK 1.2. As with the GVF, layout and drawing are separate from the core classes. There are classes called a Walk, Trail, and Path that can be used to traverse a graph, although they are limited to traversals of strictly alternating nodes and edges. Clustered graphs can't be traversed in the straightforward way that is possible with the GVF because it isn't possible to have graph objects on a Walk, Trail, or Path. It is also possible for nodes and edges to be shared among multiple graphs in the GFC. However, different method signatures must be used in the base classes depending on whether there are multiple graphs or not.

The Graph Template Library (GTL)[39] was developed at the University of Passau in C++ based on the Standard Template Library (STL). The model for the API is based on LEDA[35, 36]. The GTL is used in order to implement a toolkit for graphs called Graphlet[40] and can be used to implement other systems. The GTL consists of a set of data structures for describing graphs and a set of algorithms for manipulating them, including several useful algorithms such as planarity testing. It is possible to handle multiple graphs in the GTL using a feature called "hiding". It is possible to manipulate a subgraph by hiding all nodes and edges that don't belong to it and performing operations on the graph. To use the entire graph again, you are required to "unhide" nodes and edges. Although this approach makes it possible to manipulate subgraphs without building entirely new graphs, it doesn't effectively solve the problem of manipulating nested multiple graphs.

Perhaps the most noticeable difference between the GVF and the systems discussed above is the GVF's use of properties to distinguish between graph types such as trees and directed graphs. All the other systems except for the Graphics Template Library employ a hierarchy of graph classes. We have already pointed out the disadvantages of such an arrangement for dynamic graphs (see Section 2.2). Although there is limited support for multigraphs in the GFC, other systems did not appear as capable of handling the nested multigraph structures that the GVF was designed to handle. This was generally due to the fact that graph objects have their own separate branch in the object hierarchy and keep lists of the nodes and edges that they contain.

The node-centric design of the GVF provides all the features needed to visualize nested multigraphs in a flexible way. Although nested multigraph structures aren't necessary for all tasks, they are essential for the representation of multiple clusterings. We feel that this simple idea, together with our application of

properties to overcome the problems of a graph type hierarchy, make the GVF a unique foundation for the visualization of graphs.

4. CONCLUSIONS

The special requirements of fully interactive graph visualization have been used to create a framework that is uniquely equipped to handle such things as dynamic and nested multigraphs. Although the libraries discussed here are still fairly new, they have proved to be useful in implementing a graph visualization system that is capable of meeting the requirements that we have discussed. The flexibility and extensibility of the GVF has been tested by a small group of researchers in the field.

There is an artificial division between different types of visualization systems because of the specialized knowledge and tools that are currently necessary to implement them. However, it is not difficult to imagine that graph visualization could be useful to view relations that have been induced in the data, such as the relations resulting from clustering. Because clustering is so frequently used in information visualization applications that are not necessarily graph-oriented, it could be argued that the core classes would be useful as a base class for representing such data.

Aside from the obvious resources such as memory and CPU, huge graphs require special measures. We would like to be able to view graphs with hundreds of thousands of nodes with possible real-time changes. Future work could involve methods that allow us to incrementally navigate such structures, for example, storing a graph in a database or dynamically retrieving graph information from a remote process. An additional challenge in such a system would be to provide graph editing facilities that are just as effective and natural as those found in graph editing systems that are available for smaller graphs.

The current `Requirements` object is a boolean expression of property comparisons. These expressions could be used to define interactive filters. Future work could involve allowing the user to use either a GUI widget or command shell, or both, to define a `Requirements` object that is then applied to the graph as a filter.

REFERENCES

[1] Henry T R. Interactive Graph Layout: The Exploration of Large Graphs. *PhD Thesis*, in Department of Computer Science, Tucson: University of Arizona, ftp://ftp.cs.arizona.edu/reports/1992/TR92-03.ps, 1992.

[2] Herman I, Melançon G, Delest M. Tree Visualisation and Navigation Clues for Information Visualisation. *Computer Graphics Forum*, 1998; **17** (2): 153–165.

[3] Herman I, Marshall M S, Melançon G, Duke D, Delest M, Domenger J-P. Skeletal Images as Visual Cues in Graph Visualization. *Proceedings of the Data Visualization '99, Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization.* Vienna, 1999; 13–22.

[4] Herman I, Marshall M S, Melançon G. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 2000; **6** (1): 24-43.

[5] Himsolt M. *GML - Graph Modelling Language*. http://infosun.fmi.uni-passau.de/Graphlet/GML/, 1997.

[6] Herman I, Marshall M S. GraphXML — An XML-based graph description format. *Proceedings of the Symposium on Graph Drawing*, 2000; 52-62.

[7]. *Scalable Vector Graphics (SVG).* http://www.w3.org/Graphics/SVG/, 2000.

[8] Sarkar M, Brown M H. Graphical Fisheye Views. *Communications of the ACM*, 1994; **37** (12): 73–84.

[9] Horstmann C S, Cornell G. Chapter 8: JavaBeans. in: *Core Java 2 , Volume 2: Advanced Features*: Sun Microsystems Press/Prentice Hall, 2000.

[10] Upson C, Thomas Faulhaber J, Kamins D, Laidlaw D, Schlegel D, Vroom J, Gurwitz R, Dam A v. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics & Applications*, 1989;July 1989): 30-42.

[11] Chi E H, Barry P, Riedl J, Konstan J. A Spreadsheet Approach to Information Visualization. *Proceedings of the IEEE Symposium on Information Visualization*, 1997; 17-24.

[12] Melançon G, Dutour I, Bousquet-Melou M. Random generation of Dags for graph drawing. Centre for Mathematics and Computer Sciences, Amsterdam INS-R0005, 2000, ftp://ftp.cwi.nl/pub/CWIreports/INS/INS-R0005.pdf.

[13] Marshall M S, Herman I, Melançon G. Automatic Generation of Interactive Overview Diagrams for the Navigation of Large Graphs. Centre for Mathematics and Computer Sciences, Amsterdam INS-R0014, 2000, http://www.cwi.nl/InfoVisu/papers/Hierarchical.pdf.

[14] Wills G J. Niche Works - Interactive Visualization of Very Large Graphs. *Journal of Computational and Graphical Statistics*, 1999; **8** (2): 190-212.

[15] Huang M L, Eades P. A Fully Animated Interactive System for Clustering and Navigating Huge Graphs. *Proceedings of the Symposium on Graph Drawing GD'98*. Berlin, 1998; 374-383.

[16] Munzner T. Drawing Large Graphs with H3Viewer and Site Manager. *Proceedings of the Symposium on Graph Drawing GD'98*. Berlin, 1998; 384–393.

[17] Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns*, Addison-Wesley Professional Computing Series, 1995.

[18] Arbab F, Herman I, Reynolds G J. An Object Model for Multimedia Programming. *Computer Graphics Forum*, 1993; **12** (3): 101-113.

[19] Pizzonia M, Battista G D. Object-Oriented Design of Graph Oriented Data Structures. *ALENEX*, 1999.

[20] Duke D J, Herman I, Marshall M S. *PREMO: A Framework for Multimedia Middleware: Specification, Rationale, and Java Binding*, Springer, 1998.

[21] Eckel B. *Thinking in Java*, Prentice Hall, 1998.

[22] Sosnoski D M. *Java performance programming, Part 1: Smart object-management saves the day*. http://www.javaworld.com/javaworld/jw-11-1999/jw-11-performance.html, 1999.

[23] Coad P, Mayfield M. *Java Design: Building Better Apps and Applets*, Prentice Hall, 1997.

[24] Alpert C J, Kahng A B. Recent Developments in Netlist Partitioning: A Survey. *Integration: the VLSI Journal*, 1995; **19** (1-2): 1-81.

[25] Herman I, Marshall M S, Melançon G. Density Functions for Visual Attributes and Effective Partitioning in Graph Visualization. *Proceedings of the IEEE Symposium on Information Visualization*. Salt Lake City, Utah, U.S.A., 2000; 49-56.

[26] Wilson R M, Bergeron R D. Dynamic Hierarchy Specification and Visualization. *Proceedings of the IEEE Symposium on Information Visualization (InfoVis '99)*, 1999; 65–72.

[27] Brandes U, Shubina G, Tamassia R. Improving Angular Resolution in Visualizations of Geographic Networks. *Proceedings of the Second Joint Eurographics and IEEE TCVG Symposium on Visualization*. Amsterdam, 2000; 23-33.

[28] He T. Internet-Based Front-End to Network Simulator. *Proceedings of the Data Visualization '99*. Vienna, Austria, 1999; 247–252.

[29]. *The Graph Editor Toolkit for Java White Paper*. http://www.tomsawyer.com/get/paper-java.html, 1999.

[30] ILOG. http://www.ilog.com/products/jviews/graphlayout/.

[31] Gansner E R, North S C. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 2000; **30** (11): 1203-1233.

[32] Paulisch F N, Tichy W F. EDGE: an Extendible Graph Editor. *Software - Practice and Experience*, 1990; **20** (S1): 63-88.

[33] Mutzel P, Gutwengwer C, Brockenauer R, Fialko S, Klau G, Kruger M, Ziegler T, Naher S, Alberts D, Ambras D, Koch G, Junger M, Bucheim C, Leipert S. A Library of Algorithms for Graph Drawing. *Proceedings of the Symposium on Graph Drawing GD'98*. Berlin, 1998; 456-457.

[34]. *Graph Drawing Toolkit*. http://www.dia.uniroma3.it/~gdt/, 1999.

[35] Mehlhorn K, Näher S. *LEDA: A Platform for Combinatorial and Geometric Computing*, Cambridge University Press, 1999.

[36]. *LEDA Research*. http://www.mpi-sb.mpg.de/LEDA/, 1999.

[37] Berry J, Dean N, Goldberg M, Shannon G, Skiena S. Graph Drawing and Manipulation with LINK. *Proceedings of the Symposium on Graph Drawing GD'97*, 1997; 425-437.

[38] Cesar C L. *Graph Foundation Classes for Java*. http://www.alphaWorks.ibm.com/tech/gfc, 1999.

[39] Forster M, Pick A, Raitner M. *Graph Template Library*. http://infosun.fmi.uni-passau.de/GTL/, 1999.

[40] Himsolt M. Graphlet: design and implementation of a graph editor. *Software — Practice and Experience*, 2000; **30** (11): 1303-1324.