A System for Graph-Based Visualization of the Evolution of Software

Christian Collberg[1*] Stephen Kobourov[1†] Jasvir Nagra[2‡] Jacob Pitts[1] Kevin Wampler[1†]

[1] Department of Computer Science,
University of Arizona, Tucson, AZ 85721.
{collberg,kobourov,jpitts,wamplerk}@cs.arizona.edu
[2] Department of Computer Science,
University of Auckland, Auckland, New Zealand.
jas@cs.auckland.ac.nz

Abstract

We describe GEVOL, a system that visualizes the evolution of software using a novel graph drawing technique for visualization of large graphs with temporal component. GEVOL extracts information about a Java program stored within a CVS version control system and displays it using a temporal graph visualizer. This information can be used by programmers to understand the evolution of a legacy program: Why is the program structured the way it is? Which programmers were responsible for which parts of the program during which time periods? Which parts of the program appear unstable over long periods of time and may need to be rewritten? This type of information will complement that produced by more static tools such as source code browsers, slicers, and static analyzers.

1 Introduction

There are many situations when a programmer is faced with having to learn and understand an existing large and complex software system. Consider, for example, the following scenarios where Bob is a programmer and P is a large legacy program:

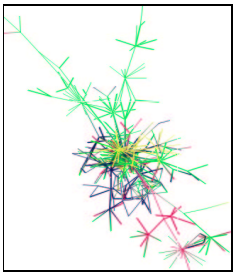
- Bob is asked to add new functionality to P;
- Bob is asked to fix bugs in P;
- Bob is asked to determine whether algorithms exist in P that violate intellectual property rights;
- Bob is asked to rewrite P in a new programming language;
- Bob is asked to port P to a new operating system or architecture.

In many cases Bob will find that the program is undocumented, unstructured, and poorly written. Worse, the original developers may not be available to explain how the system works. Before he can start modifying the program he therefore needs to build a mental model of its structure. To aid in this discovery process he can run the program, examine the source code, and read any available documentation. Various tools such as source code browsers and static analyzers may be helpful in this respect.

In this paper we will describe a new tool — GEVOL — that aid in the discovery of the structure of legacy systems. GEVOL discovers the *evolution* of a program by visualizing the changes the system has gone through. In particular, GEVOL extracts information about Java programs that are stored within a CVS version control system. It then extracts inheritance graphs, call graphs, and control-flow graphs of the program and displays the changes the graphs have gone through since the inception of the program. GEVOL allows Bob to visualize

- when particular parts of the program were first created;
- during which periods which parts of the program were most heavily modified;
- which parts of the program seem to have been unstable for a long period of time and therefore may be in need of being rewritten;
- which programmers have modified which parts of the code when;
- which parts of the program have grown in complexity over a long period of time.

GEVOL is not intended as a stand-alone system. Rather, our ultimate goal is to integrate it with other tools such



*Partially supported by the NSF under grant CCR-0073483 and by the AFRL under contract F33615-02-C-1146.
†Partially supported by the NSF under grant ACR-0222920.
‡Partially supported by the New Economy Research Fund of New Zealand.

as source code browsers. This will allow a programmer to examine the source code, control-flow, inheritance structure, and call structure of a program — *as they change over time* — in order to understand every aspect of the system.

GEVOL is in active development. We are currently in the process of integrating several software complexity metrics [6,15,16,19,25] within the system. This will allow the graph visualizations to be driven by how the complexity of a class or a method is changing over time. Figure 1 shows an overview of the design of GEVOL.

The remainder of this paper is structured as follows. In Section 2 we present the types of visualizations our system is capable of. In Section 3 we discuss the TGRIP temporal graph visualization system on which GEVOL is based. In Section 4 we describe how information is collected from CVS repositories. In Section 5 we present related work, in Section 6 we discuss our findings, and in Section 7 we summarize our results.

2 Temporal Visualization Models

We are hoping GEVOL will be a useful tool when learning about a new code-base. Not only will we be able to view a current snapshot of the code, we will be able to visualize the entire history of the development process. This may lead to interesting insights that could not otherwise be gleaned from examining the mere source.

Our goal is to develop a system that allows the visualization of *all* evolutionary aspects of a program. We are therefore extracting all available information from the CVS repository of a Java program, expressing it as graphs, and using a temporal graph drawing system to visualize the information. We are currently extracting the following data:

1. The author of each change of each file.
2. The control-flow graphs of each method in the program.
3. The change in each basic block in the control-flow graphs.
4. The inheritance graph of the program.
5. The call-graphs of the methods of the program.
6. The time of each change to each file.

Every piece of information is collected for every time-slice. The temporal granularity is configurable but in our current system the size of each slice defaults to one day.

This information allows us to visualize the evolution of a program in several useful ways:

- We color-code nodes depending on how long they have been unchanged. All nodes start out being red, then grow paler and paler for every time-slice they have remained unchanged until they are finally drawn a pale blue. When a node changes again it returns to red and the process repeats. As the user moves through the time-slices this will draw his attention to parts of the system that are in flux at different points in time.
- When the user notices an interesting event (say, a code segment changing heavily for a long period of time) he can click on a node to examine the set of authors who have affected these changes.
- If the user notices that an area of the graph remains constantly red, but does not grow significantly, this may mean the area is a site of constant bug fixes and may need to be redesigned or better tested.

3 Visualization of Large Evolving Graphs

In theory, every problem can be encoded as a graph problem, by representing the input/output in binary and treating them as graphs (adjacency matrix or list). In this case, the problem becomes that of finding the transformation that takes the input graph into the output graph. While this approach is not practical in many applications, it does make sense in visualizing programs, in particular, inheritance graphs, call graphs, and control-flow graphs. Visualizing such graphs can lead to discovery of unsuspected relationships, patterns, and trends.

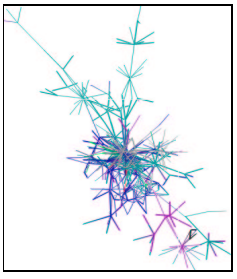
In this paper we consider the problem of interactive visualization of large graphs that have a temporal component. We develop new techniques, models and algorithms that allowed us to implement a prototype system for interactive visualization of large temporal graphs arising from large software development.

The main algorithmic challenge is to develop techniques, models, algorithms and data structures for interactive temporal graph visualization. Consider a graph that evolves through time. The changes in the graph include adding and removing vertices and adding and removing edges. The visualization of such data must ensure that:

- the drawing is *readable*
- the drawing preserves the *mental map* of the underlying structure

A readable layout for a graph is one that shows the underlying relationships. For example, if the graph contains a clique of nodes, we would like these nodes to be uniformly placed on a sphere and not, say along a straight-line segment. The mental map of the user is preserved if the same parts of the graph that appear in different frames remain in the same position. This is usually too restrictive and instead selected landmarks can be chosen that remain in the same position while other parts are allowed to deviate from their previous positions. A naive approach to displaying a sequence of graphs would be to draw each one from scratch. If we were to layout each graph independent of the others, it is unlikely that the mental map will be preserved. Conversely, if we were to layout each graph incrementally from the previous one, we would preserve the mental map but the quality of the layout will likely suffer dramatically when global changes are not allowed.

We propose an approach that combines both readability and mental map preservation. Let $G_1, G_2, \ldots, G_n$ be the sequence of graphs that we would like to visualize as a time-series. Define the aggregate graph, G^*, to be the graph obtained by *adding* all the graphs in the sequence. That is, G^* is a weighted graph in which a vertex has weight that corresponds to the number of frames in which the vertex appears (edge weights are defined analogously). The problem



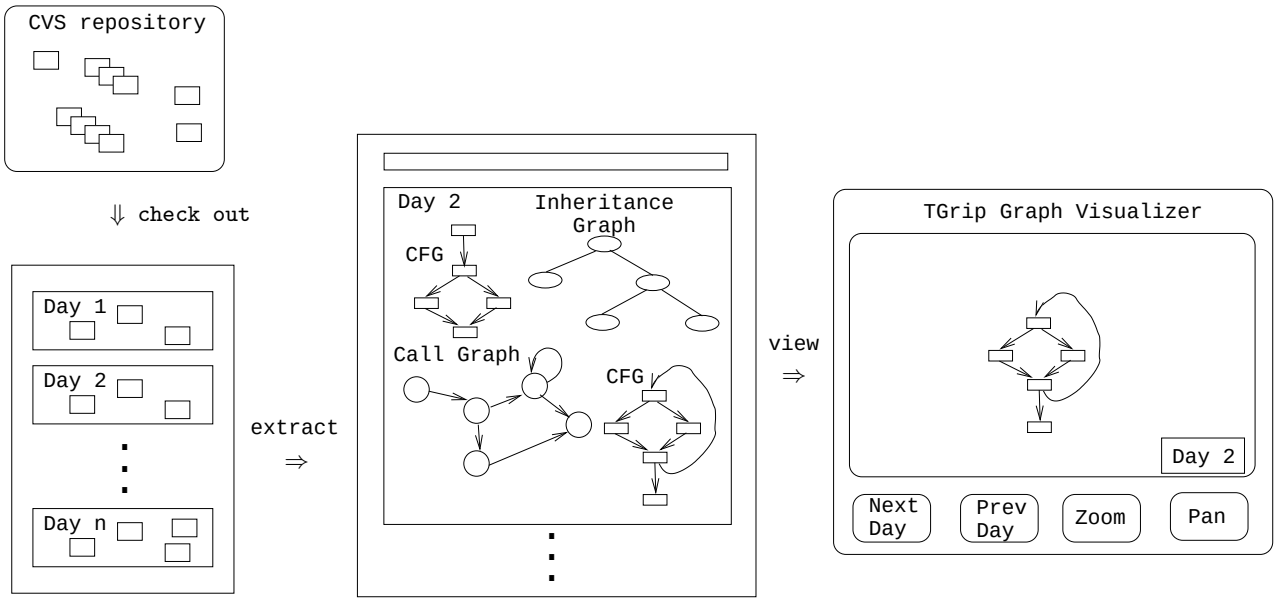


Figure 1: Overview of the GEVOL system.

becomes that of finding a readable layout for the aggregate graph, taking into account the edge and vertex weights and using the placement of the vertices in each time-frame.

The algorithm used to display the various program structure graphs is based on GRIP [13,14]. GRIP can layout very large graphs in reasonable time by computing a hierarchical filtration of the graph. This set of filtrations of a graph G forms a sequence $\{V_n\}$ of subsets of the nodes of G such that for every $V_i, V_j \in \{V_n\}$, $i < j \Rightarrow V_i \subset V_j$. In practice, it is usually the case that $|V_{i+1}| \geq 2|V_i|$, so a filtration does not normally contain very many elements. The filtrations are laid out from smallest to largest (smallest index to largest index) and the layout of V_i is used to provide an outline of the layout of V_{i+1}.

The layout of each filtration proceeds by using an approach related to the spring-embedder of Eades [10] and the force-directed method of Kamada and Kawai [17,18]. The main underlying principle of these methods is that vertices repel each other, while edges, prevent adjacent vertices from getting too far from each other. Thus, for a given node v in G, the displacement of v is calculated by:

$$\vec{F}_{KK}(v) = \sum_{u \in N_i(v)} \left(\frac{\|p[u] - p[v]\|^2}{d_G(u,v)^2 \cdot edgeLen^2} - 1 \right) (p[u] - p[v]) \quad (1)$$

where $p[u]$ is the position of node u, $N_i(v)$ is the the neighborhood of node v, $d_G(u,v)$ is the distance between nodes u and v in graph G, and $edgeLen$ is the predefined optimal edge length. In the last level of the filtration a Fruchterman-Reingold calculation [12] for the force vector is used:

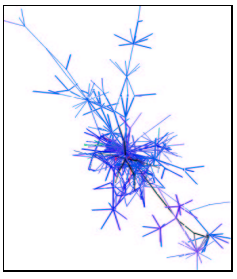
$$\vec{F}_{a,FR} = \sum_{u \in Adj(v)} \frac{\|p[u] - p[v]\|^2}{edgeLen^2} (p[u] - p[v]) \quad (2)$$

$$\vec{F}_{r,FR} = \sum_{u \in N_i(v)} s \frac{edgeLen^2}{\|p[u] - p[v]\|^2} (p[v] - p[u]) \quad (3)$$

The displacement of a node v is then simply $\vec{F}_{FR}(v) = \vec{F}_{a,FR} + \vec{F}_{r,FR}$.

3.1 Dynamic Graph Visualization

GRIP [13,14] is designed to quickly layout graphs with tens of thousands of vertices without assuming any information about the underlying graphs. This makes it a good base for the visualization of graphs that evolve through time. However, before we can employ the aggregate-graph approach to GRIP, we need to modify it so that attributes such as weights on the nodes or edges of a graph are taken into account. To accommodate the kinds of information which is often of interest in software visualization, GRIP was extended to support two additional attributes: weights of nodes and of edges, and time-slice information. The meaning of the weight information is self-evident, and a time-slice is a label associated with each node representing which snapshot of the state of the system being analyzed the node is in. A dynamic graph will then consist of a series of time-slices, each of which is a graph representing the state of the system at a given point in time. Edges can be arranged between the



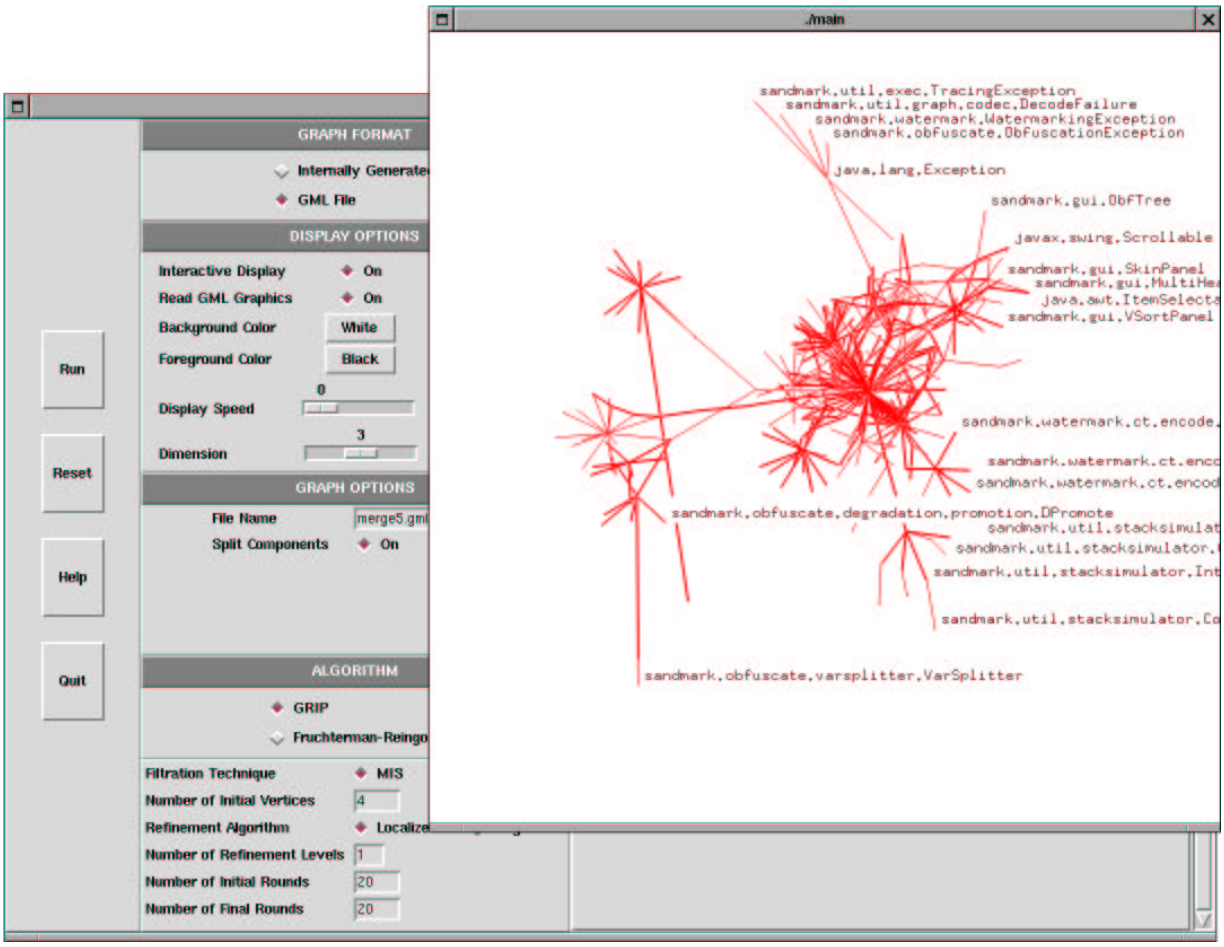


Figure 2: Snapshot of the GEVOL system viewing SandMark's inheritance graph. A subset of the classes have been labelled.

time-slices in various ways depenging on what properties we are interested in.

3.2 Node and Edge Weights

Modification to the forces that act on the nodes were made to accommodate weights and achieve the following goals:

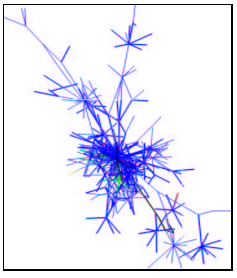
1. Two nodes connected by an edge of weight 0 should behave as if not connected by an edge at all;
2. An edge connecting two nodes, each of weight zero, should have a natural length of zero;
3. Heavy nodes should be placed further apart;
4. Heavy edges should be shorter;
5. If an edge of weight w connects two nodes of weight w, the edge's ideal length should be the same as an edge of weight 1 connecting two nodes of weight 1, but the larger the w, the stronger the connection should be.

Given these considerations, an edge, e of weight w_e connecting nodes u, v of weight w_u, w_v, respectively, is given an ideal length of:

$$\frac{\sqrt{w_u \cdot w_v}}{w_e} \tag{4}$$

This formula will lead to a division by zero if $w_e = 0$. The resulting infinite distance is indeed the correct ideal distance for the Fruchterman-Reingold force based calculations, since two disconnected nodes have only repulsive forces between them. In practice, however, this is undesirable and thus we ensure that all edges of weight zero are removed.

To account for the layout constraints of weighted graphs, the graph distance between two nodes is replaced with the ideal distance between the nodes. Because of the computational and space requirements of calculating the effects of



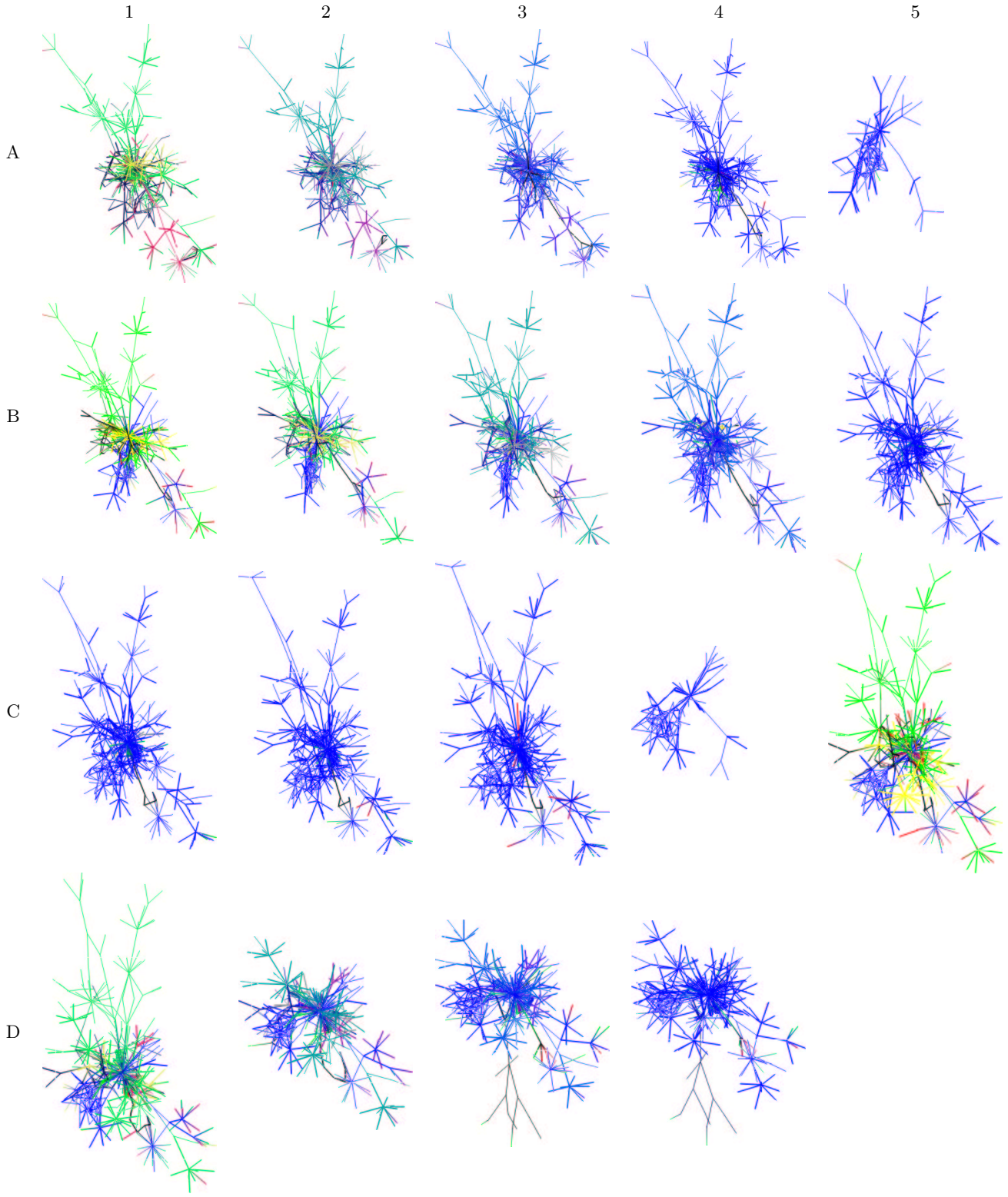


Figure 3: Snapshots of the SandMark inheritance graph. Nodes are colored by author and by latest change. When a node first appears it is given the color of its author. In this example author 1 is red, author 2 is yellow, other authors are green, and author-less classes (such a library or system classes) are black. For every time-step that a node does not change, its color will fade to blue. Nodes belonging to author 1 will go through the color progression ⟨, , , , ⟩, while author 2's nodes will go through ⟨, , , , ⟩.

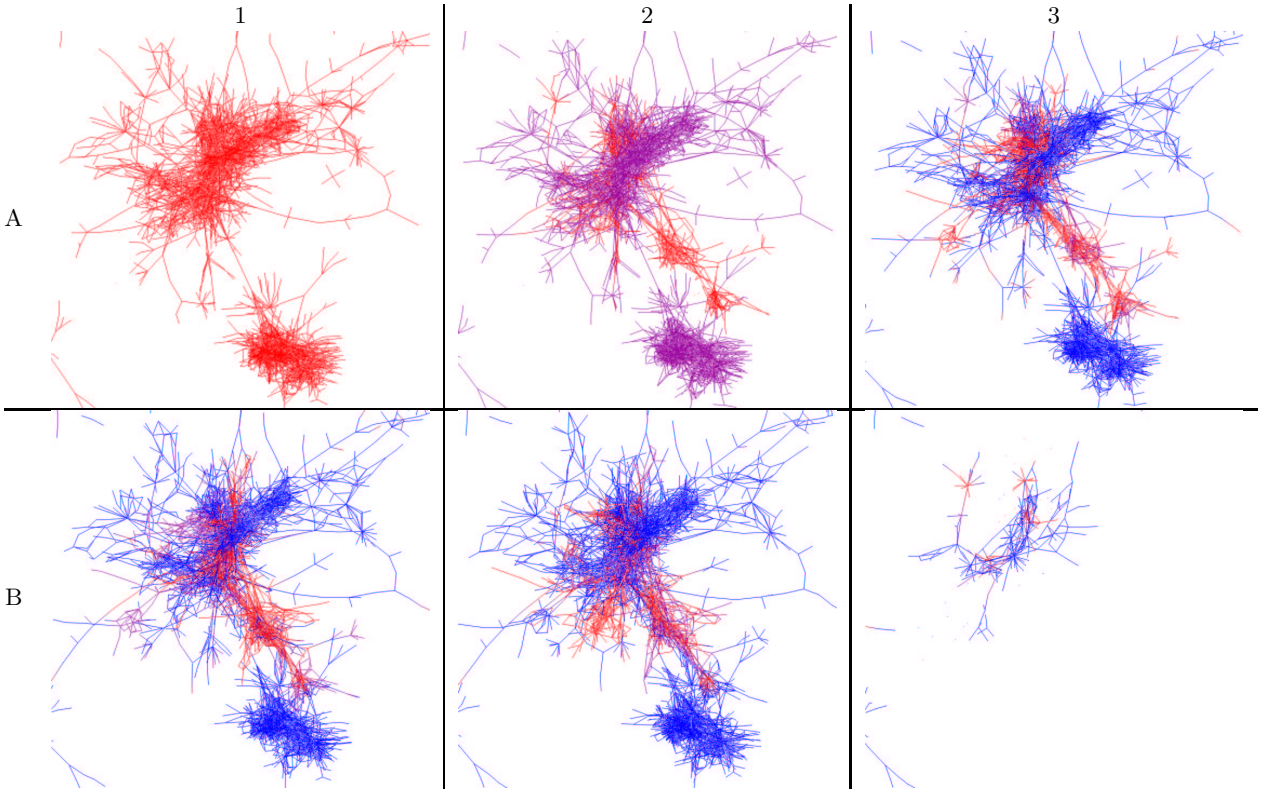


Figure 4: Snapshots of the SandMark call-graph. Nodes start out red. As time passes and a node does not change, it turns purple and, finally, blue. When another change is affected the node again becomes red.

all paths between two nodes, or of computing the shortest weighted path between them, an approximation is used. Let $p_1, p_2, \ldots, p_n$ be the sequence of nodes in the shortest unweighted path in G connecting two nodes, u and v. Then we define:

$$optD_G(u,v) = \sum_{i=1}^{n-1} \frac{\sqrt{w_{p_i} \cdot w_{p_{i-1}}}}{w_{e_{p_i p_{i-1}}}} \tag{5}$$

In pratice this approximation works both quickly and well. The final force calculation used in the modified Kamada-Kawai method is:

$$\vec{F}_{KK}(v) = \tag{6}$$
$$\sum_{u \in N_i(v)} \left(\frac{2\|p[u]-p[v]\|^2 \cdot (p[u]-p[v])}{(edgeLen \cdot optD_G(u,v))^2 + \|p[u]-p[v]\|^2} \right) -$$
$$- \sum_{u \in N_i(v)} (p[u]-p[v])$$

To achieve an aesthetically pleasing layout of the graph, it is also necessay to employ modified Fruchterman-Reingold forces, as the Kamada-Kawai method does not acieve satasfactory methods by itself, but rather creates a good approximate layout so that the Fruchterman-Reingold calculations can quickly "tidy up" the layout. The modifications needed to support weighted graphs are simple:

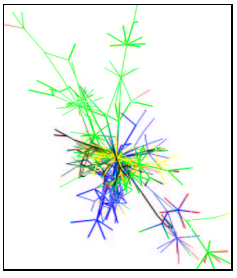
$$\vec{F}_{a,FR} = \sum_{u \in Adj(v)} \frac{w_e \cdot \|p[u]-p[v]\|^2}{edgeLen^2} (p[u]-p[v]) \tag{7}$$

$$\vec{F}_{r,FR} = \sum_{u \in N_i(v)} s \frac{edgeLen^2 \cdot \sqrt{w_u \cdot w_v}}{\|p[u]-p[v]\|^2} (p[v]-p[u]) \tag{8}$$

3.3 Graph Time-Slices

The modifications needed to support time-slices in the Kamada-Kawai method are quite simple. In equation (6) the only alteration required is that the the function $optD_G(u,v)$ be redfined so that for two nodes u, v with time-slice indexes of t_u and t_v respectively:

$$optD_G(u,v) = \delta_{t_u t_v} \cdot \frac{\sqrt{w_u \cdot w_v}}{w_e} \tag{9}$$



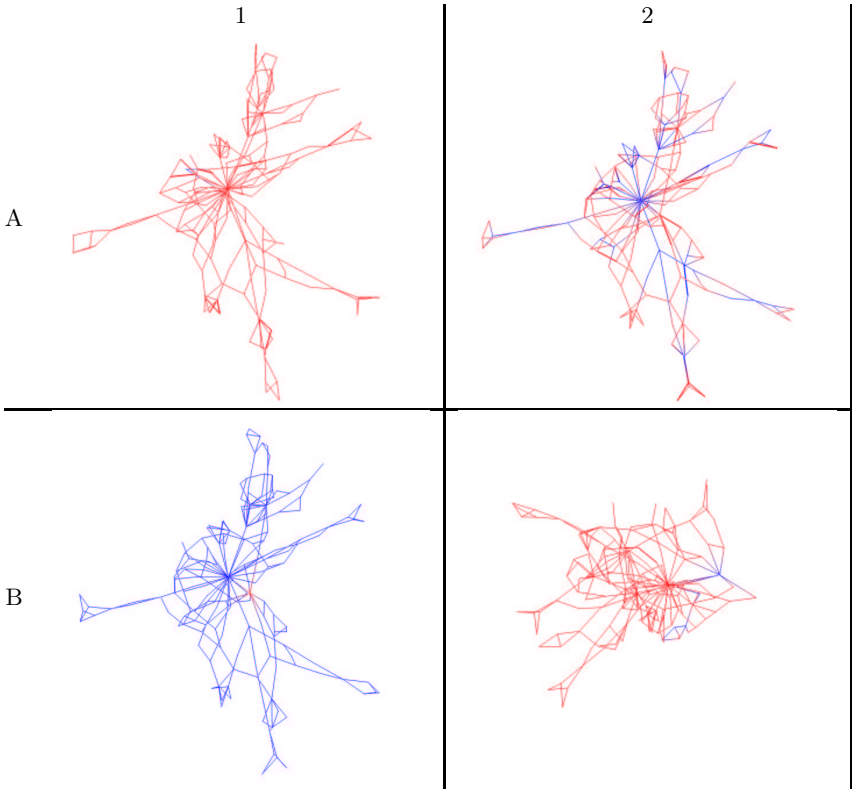


Figure 5: The SandMark control-flow graph. As with the call-graph in Figure 4, changed nodes start out red and gradually fade to blue. Note that in the current system, changes a large number of nodes of the graph (such as shown in B_2 above) result in undesirable changes in layout of the graph.

Where δ is the Kronecker delta:

$$\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

The modifications needed for the Fruchterman-Reingold calculations are similar: repulsive forces are eliminated outside of a given time-slice:

$$\vec{F}_{a,w,t,FR} = \vec{F}_{a,w,FR} \tag{10}$$

$$\vec{F}_{r,w,t,FR} = \delta_{t_u t_v} \cdot \vec{F}_{r,w,FR} \tag{11}$$

4 Extracting CVS Information

As shown in Figure 1, the Gevol system will check out consecutive versions of the code for the Java program under study. The program is compiled to a collection of Java classfiles. The classfiles are loaded into Gevol and control-flow graphs, call graphs, and inheritance graphs are built. Each graph is stored in an individual file which can later be loaded by the Tgrip viewer.

Thus, the result of the extraction step is a sequence of files, one per generated graph. Let n be the number of days in the CVS repository. There is one call graph per day:

$$\langle \text{Call}_1, \text{Call}_2, \cdots, \text{Call}_n \rangle,$$

one inheritance graph per day:

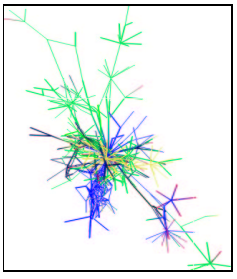
$$\langle \text{Inher}_1, \text{Inher}_2, \cdots, \text{Inher}_n \rangle,$$

and a number of control-flow graphs per day:

$$\left\langle \begin{array}{lll} \text{CFG}_{1,m_1}, & \text{CFG}_{1,m_2}, & \\ \text{CFG}_{2,m_1}, & \text{CFG}_{2,m_2}, & \text{CFG}_{2,m_3}, \\ & \cdots & \\ \text{CFG}_{n,m_1}, & \text{CFG}_{n,m_2}, & \text{CFG}_{n,m_3} \end{array} \right\rangle.$$

Constructing inheritance graphs is straight-forward since each Java class-file indicates the parent of the class and the interfaces it implements. Since a Java class can extend one class (Java is a single-inheritance language) but implement several interfaces the inheritance graph is a DAG.

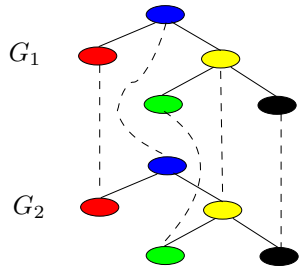
Constructing call-graphs is slightly more complicated. The target of a method invocation `p.m()` will depend on the runtime type of `p`. We do a conservative type-based analysis of the potential targets of method invocations by considering



the inheritance graph. A more precise data-flow based analysis would be possible but is not necessary for our purposes. The call-graph will typically be a forest of directed graphs. The reason is that most Java programs are multi-threaded (if not explicitly then implicitly through the use of graphical user interfaces) and many calls appear "spontaneously" through actions of the Java runtime system.

Control-flow analysis is complicated by the fact that most Java bytecode instructions can throw an exception. As a result control-flow graphs are very dense with exception edges and hence become highly unreadable. We therefore omit many edges such as those generated by possible null-pointer exceptions.

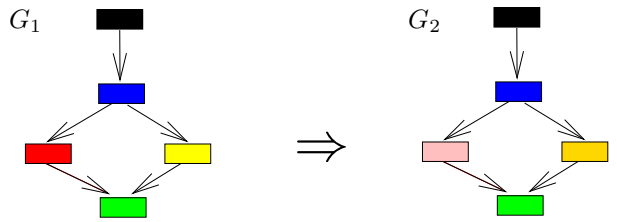
Once daily graphs have been constructed they must be merged into a "time-slice-graph." To merge two graphs G_1 and G_2 (where G_2 is a modified version of G_1) we identify which node n from G_1 corresponds to which node m from G_2 and add an edge $n \rightarrow m$:



(Here, colors indicate node identities.) TGRIP knows that these (dashed) time-slice edges should be treated specially. In particular, TGRIP will attempt to place the same node from two slices in approximately the same location. This will allow for smooth transitions as the user navigates through time.

For inheritance graphs and call-graphs it is straightforward to add time-slice edges. The reason is that every node can easily be given a unique identity. In the case of the inheritance graph each node is identified by the fully qualified class name. In the case of the call graph each node is identified by *class-name:method-name:method-signature*. It is necessary to include signatures in the identifier since Java allows method overloading.

Adding time-slice edges for control-flow graphs is significantly more difficult. To see why, consider the following example:



Here, two nodes (corresponding to the *then* and *else* branches of the if-statement) of the control-flow graph have changed. However, it will in general not be possible to determine which node in G_1 changed into which node in G_2. We might heuristically identify the two nodes with the smallest edit distance, but at best this can only be an educated guess. Our current version of the system employs a very conservative estimate of which nodes correspond to which nodes across slices. In particular, it identifies nodes by calculating a hash on the instruction body of the node and linking nodes with identical hashes across time-slices. It assumes that nodes that have changed and thus have new hash values are in fact new nodes. This means that changed nodes may not appear close to the same node over different time-slices.

In practice, this is not a significant problem if only a few nodes change since these other nodes fix the position of the new node relatively close to the original, and such that it is perceptively obvious that the new node is an altered version of the old one.

In addition to the information extracted from the program code we also incorporate information from the CVS respository itself into the graphs. This includes time-stamps and author information.

After all pieces of information have been gathered and the graphs have been merged we are left with three graphs: an inheritance graph, a call-graph, and a set of control-flow graphs. Each graph has n (number of days) layers, where each node in one layer is connected by a time-slice edge to the corresponding node in the next layer.

5 Related Work

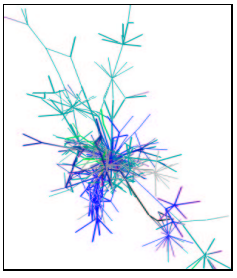
Many program visualization tools have been proposed in the past. The aim of these tools is to improve the understanding of computer programs by humans by portraying them in a form that is more readable than mere source code. In this section we will briefly review some software visualization tools. For more in-depth information we refer the reader to one of the many available visualization taxonomy studies [22,23,26,27].

5.1 Static Visualization

One of the best known interactive software visualization systems is BALSA [4] developed at Brown University. BALSA annotates the program being visualized with hooks so that "interesting events" such as changes to data structures and subroutine calls and returnss can be relayed to the visualization system. This in turn builds up a view that corresponds to these events.

BALSA later evolved into Zeus [5], a system that shows multiple synchronized views of a running program. Zeus allows a developer to interrupt the running program and edit it using any one of many available data structure representations. The changes are propagated to update all other views. Furthermore, Zeus allows a user to use sound and color to enhance the visualization.

SHriMP [28] is a more recent system that offers a variety of different graphical views of a software system. For example, class and inheritance hierarchies as well as aggregation can be visualized. A programmer trying to understand how various components of a software system fit together can



zoom in or out of particular components as well as focus on specifics such as relevant documentation or source code.

One major problem with visualizing call-graphs is their density. Young [29] attempts to overcome this problem by abandoning the standard graph view for a *CallStax* view. This lays out each call chain as a stack of cubes. The view is examined in a virtual reality environment.

5.2 Visualizing Evolving Software

Real-world software changes over time and software becomes better or worse because of the changes made to it. There are many tools available for analyzing such changes. These usually extract historical information stored by change management systems such as CVS and SCCS. SoftChange [20] is such tool that extracts complexity, size, purpose and author of changes made to a program and summarizes this information in textual web-based reports. The authors note that "to study software changes it was essential to handle large and complex data sets. The volume, complexity, and lack of structure of software change data overwhelm standard statistical analysis tools."

Ball [2] describes a tool that attempts to deduce a better understanding of a program from its development history. The system attempts to synthesize views of the requirements of the software, the implementation technology, the development process and the organization of developers based on the version control system logs and the source code.

Ball [1] describes a system that visualizes many different aspects of software using three different types of representation: *Line representation* shows program source at three scaling levels, giving both detail and overview. *Pixel representation* shows each line of code as an individual pixel. *Hierarchical representation*, finally, is used to model statistics for structured data such as file systems. In all cases the text or pixels are color coded to show a particular statistic of interest. Particularly relevant to our work is the fact that the system collects information about *code age*.

Eick [11] visualizes software changes using mostly traditional views, such as bar-graphs, pie-charts, matrix views, and cityscape views. A large number of different types of statistics can be displayed, allowing changes to the system to be viewed from many different perspectives. The most significant strength of this system, however, is that is able to examine extremely large programs, up to several million lines of code.

5.3 Dynamic Graph Drawing

Graph drawing techniques for static graphs have been used for dynamic graph visualization. North [24] studies the incremental graph drawing problem in the DynaDAG system. Brandes and Wagner adapt the force-directed model to dynamic graphs using a Bayesian framework [3]. Diehl and Görg [9] consider graphs in a sequence to create smoother transitions. Special classes of graphs such as trees, series-parallel graphs and st-graphs have been also been studied in dynamic models [7,8,21]. Most of these approaches, however, are limited to special classes of graphs and usually do not scale to graphs over a few hundred vertices.

6 Discussion

Figure 3 shows a sequence of snapshots of the SandMark inheritance graph. There a several notable events. In Figure $3_{A,5}$ and Figure $3_{C,4}$ one author "broke the build," i.e. checked in code that would not compile properly. This problem was fixed in the next time-slice. Going from the time-slice in Figure $3_{D,1}$ to Figure $3_{D,2}$ a large code-segment (almost 10,000 lines of code shown as two green tendrils stretching towards the top of the page) was removed.

It is also interesting to note that different authors can be seen to play distinct roles. Author 2 (yellow) is obviously more involved in the core architecture of the software. The nodes (classes) he introduces lie close to the center of the inheritance tree and other classes extend them. Author 1 (red), although as prolific in generating new classes as author 1, introduces classes along the fringe of the graph. They are specializations of core classes and presumably implement actual functionality. Thus it is reasonable to conclude that author 2 is a system architect and author 1 a programmer.

Figure 4 shows snapshots of the SandMark call-graph. Figure $4_{A,1}$ shows that an early part of the system consisted of two main parts, the *gui* (top) and the *obfuscation algorithms* (bottom). In June of 2002 a new structure was created (`sandmark.util.controlflow`) which became a *mediation-point* between the two structures. This is shown in purple in Figure $4_{A,2}$. Initially, the *gui* calls the *obfuscation algorithms* directly but over time, `sandmark.util.controlflow` comes into existence between the two parts and acts as an intermediary. Figure $4_{B,3}$ shows another instance of the build being broken.

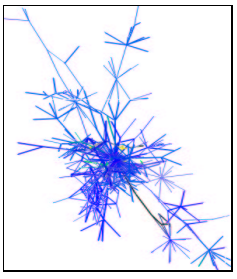
Figure 5 shows the control-flow graph for a method `sandmark.util.stacksimulator.StackSimulator.execute()` from SandMark. The large size of the graph itself makes it stand out among the control-flow graphs of other methods and identifies it as a good candidate of refactoring. Furthermore, the relative absence of blue indicating unchanged basic blocks in $_{A,1}$, $_{A,2}$ and $_{B,4}$ allows one to deduce that the most of the method is being rewritten during this period.

It is important to note that for reasonable size programs the generated graphs can be huge. Our current test case is the SandMark system which consists of approximately 90,000 lines of code developed over 200 days.[1] The generated call graphs have a total of 760,201 nodes and 2,216,034 edges over all the time-slices. The inheritance graphs have a total of 100,722 nodes and 123,145 edges.

The control-flow graphs consist of a total of 3,091,105 nodes and 3,294,038 edges. Visualizing graphs of this magnitude is a daunting task.

One of the techniques Gevol uses for making this graph more manageable is to preprocess them before displaying them to contain only those nodes that the user is currently interested in. The system allows the user to specify (using a regular expression) the range of values for a particular field of a node that the user wishes to view. For example, although the control-flow graph contains well over three mil-

[1] The actual development time is longer than that but 200 days is the extent of the CVS record.



lion nodes, the user may only be interested in those nodes that occur in a particular package or by a particular author.

7 Summary

We have presented a system for visualization of the evolution of software using a novel graph drawing technique for visualization of large graphs with a temporal component. Three different types of graphs were considered: inheritance, control-flow, and program call-graphs.

Throughout the paper on the bottom right hand side we have included a series of inheritance graphs that can be "animated" in a flip-book fashion to give an idea of the GEVOL system in action.

Acknowledgments: The extraction of some of the CVS graphs was done by Christopher Brue and Abin Shahab. Kelly Heffner helped in analyzing the temporal views of SandMark.

References

[1] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[2] T. Ball, J. Kim, A. Porter, and H. Siy. If your version control system could talk. In *ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering*, May 1997.

[3] U. Brandes and D. Wagner. A bayesian paradigm for dynamic graph layout. In G. Di Battista, editor, *Proceedings of the 5th Symposium on Graph Drawing (GD)*, volume 1353 of *Lecture Notes Computer Science*, pages 236–247. Springer-Verlag, 1998.

[4] M. Brown. Exploring algorithms using Balsa-II. *IEEE Computer*, 21(5):14–36, 1988.

[5] M. H. Brown. Zeus: A system for algorithm animation and multi-view editing. Technical Report 75, 28 1992.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[7] R. F. Cohen, G. D. Battista, R. Tamassia, I. G. Tollis, and P. Bertolazzi. A framework for dynamic graph drawing. In A.-S. ACM-SIGGRAPH, editor, *Proceedings of the 8th Annual Symposium on Computational Geometry (SCG '92)*, pages 261–270, Berlin, FRG, June 1992. ACM Press.

[8] R. F. Cohen, G. Di Battista, R. Tamassia, and I. G. Tollis. Dynamic graph drawings: Trees, series-parallel digraphs, and planar ST-digraphs. *SIAM J. Comput.*, 24(5):970–1001, 1995.

[9] S. Diehl and C. Görg. Graphs, they are changing. In *Proceedings of the 10th Symposium on Graph Drawing (GD)*, pages 23–30, 2002.

[10] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[11] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *Software Engineering*, 28(4):396–412, 2002.

[12] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Softw. – Pract. Exp.*, 21(11):1129–1164, 1991.

[13] P. Gajer, M. T. Goodrich, and S. G. Kobourov. A multi-dimensional approach to force-directed layouts. In *Proceedings of the 8th Symposium on Graph Drawing (GD)*, pages 211–221, 2000.

[14] P. Gajer and S. G. Kobourov. GRIP: Graph dRawing with Intelligent Placement. In *Proceedings of the 8th Symposium on Graph Drawing (GD)*, pages 222–228, 2000.

[15] M. H. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

[16] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, Sept. 1981.

[17] T. Kamada and S. Kawai. Automatic display of network structures for human understanding. Technical Report 88-007, Department of Information Science, University of Tokyo, 1988.

[18] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inform. Process. Lett.*, 31:7–15, 1989.

[19] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[20] A. Mockus, S. Eick, T. Graves, and A. Karr. On measurement and analysis of software changes, 1999.

[21] S. Moen. Drawing dynamic trees. *IEEE Software*, 7(4):21–28, July 1990.

[22] B. A. Myers. Visual programming, programming by example, and program visualization: A taxonomy. In *ACM SIGCHI '86 Conference on Human Factors in Computing Systems*, pages 59–66, Apr. 1986.

[23] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, Mar. 1990.

[24] S. C. North. Incremental layout in DynaDAG. In *Proceedings of the 4th Symposium on Graph Drawing (GD)*, pages 409–418, 1996.

[25] E. I. Oviedo. Control flow, data flow, and program complexity. In *Proceedings of IEEE COMPSAC*, pages 146–152, Nov. 1980.

[26] B. A. Price, I. S. Small, and R. M. Baecker. A taxonomy of software visualization. In *Proc. 25th Hawaii Int. Conf. System Sciences*, 1992.

[27] G.-C. Roman and K. C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, 26(12):11–24, 1993.

[28] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Muller. On integrating visualization techniques for effective software exploration. pages 38–45, 1997.

[29] P. Young and M. Munro. A new view of call graphs for visualising code structures, 1997.