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

Christian Collberg\(^*\)  Stephen Kobourov\(^{††}\)  Jasvir Nagra\(^{†}\)  Jacob Pitts\(^{†}\)  Kevin Wampler\(^{††}\)

\(^{†}\) Department of Computer Science,
University of Arizona, Tucson, AZ 85721.
\\{collberg,kobourov,jpitts,wamplerk\}@cs.arizona.edu

\(^{††}\) 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 a 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 aids 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 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 [Chidamber and Kemerer 1994][Henry and Kafura 1981][Halstead 1977][Oviedo 1980][McCabe 1976] 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.
In order to study the effectiveness of GEVOL the CVS tree of the Sandmark [Collberg 2003] project was used. Sandmark is a software watermarking and obfuscation tool developed jointly by the University of Arizona and the University of Auckland. It consists of over 100,000 lines of Java code and has been edited by twenty six developers over a period of two years for which the CVS tree is available.

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, and
- 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 independently 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 a weight that corresponds to the number of frames in which the vertex appears (edge weights are defined analogously). The problem 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 [Gajer and Kobourov 2000]. GRIP can lay out 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 $\{V_i\}$ of subsets of the nodes of $G$ such that for every $V_i, V_j \in \{V_i\}, i < j \Rightarrow V_i \subset V_j$. 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
The layout of each filtration proceeds by using an approach related to the spring-embedder of Eades [Eades 1984] and the force-directed method of Kamada and Kawai [Kamada and Kawai 1988, Kamada and Kawai 1989]. 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 graph \( G \), the displacement of \( v \) is calculated by:

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

where \( p[u] \) is the position of node \( u \), \( N_i(v) \) is the neighborhood of node \( v \), \( d_G(u,v) \) is the distance between nodes \( u \) and \( v \) in graph \( G \), and \( \text{edgeLen} \) is the predefined optimal edge length. In the last level of the filtration a Fruchterman-Reingold calculation [Fruchterman and Reingold 1991] for the force vector is used:

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

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

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

$$\sqrt{\frac{w_u \cdot w_v}{w_e}}$$ (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 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:

$$\text{opt}_G(u, v) = \sum_{i=1}^{n-1} \sqrt{\frac{w_{p_i} \cdot w_{p_{i+1}}}{w_{p_{i+1}p_i}}}$$ (5)

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

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

To achieve an aesthetically pleasing layout of the graph it is also necessary to employ modified Fruchterman-Reingold forces, as the Kamada-Kawai method does not achieve satisfactory 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
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.

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 function $optD_G(u, v)$ be redefined so that for two nodes $u, v$ with time-slice indexes of $t_u$ and $t_v$ respectively:

$$optD_G(u, v) = \delta_{t_u t_v} \frac{\sqrt{w_u w_v}}{\text{edgeLen}} (p[u] - p[v])$$

where $\delta$ is the Kronecker delta:

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

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

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

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 into a collection of Java class-files. The class-files 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:

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

one inheritance graph per day:

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

and a number of control-flow graphs per day:

$$\{\text{CFG}_{1,m_1}, \text{CFG}_{1,m_2}, \cdots, \text{CFG}_{2,m_3}, \cdots, \text{CFG}_{n,m_1}, \text{CFG}_{n,m_2}, \text{CFG}_{n,m_3}\}.$$
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 B2 above) result in undesirable changes in layout of the graph.
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 repository 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 [Myers, 1986, Myers 1990, Price et al. 1992], Roman and Cox 1993.

5.1 Static Visualization

One of the best known interactive software visualization systems is BALS [Brown 1988] developed at Brown University. BALS annotates the program being visualized with hooks so that “interesting events” such as changes to data structures and subroutine calls and returns can be relayed to the visualization system. This in turn builds up a view that corresponds to these events.

BALS later evolved into Zeus [Brown 1992], 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 [Storey et al. 1997] 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 [Young and Munro 1997] 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 [Mockus et al. 1999] 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 [Ball et al. 1997] 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 [Ball and Eick 1996] 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 [Eick et al. 2002] 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 it is able to examine extremely large programs, up to several million lines of code.

A similar technique called Revision Towers is used by Taylor et al. [Taylor and Munro 2002] that uses color bars of varying thickness and height to represent the current size, changes and authors of a piece of code. These bars are animated over time to show the development of the software repository.
5.3 Dynamic Graph Drawing

Graph drawing techniques for static graphs have been used for dynamic graph visualization. North [North 1996] 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 [Brandes and Wagner 1998]. Diehl and Görg [Diehl and Görg 2002] consider graphs in a sequence to create smoother transitions. Special classes of graphs such as trees, series-parallel graphs and st-graphs have also been studied in dynamic models [Cohen et al. 1995, Cohen et al. 1992, Moen 1990]. 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 6 shows a sequence of snapshots of the SandMark inheritance graph. There are several notable events. In Figure 6A and Figure 6C, 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 6D to Figure 6E 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 4A 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 4B. 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 4C 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 A1, A2 and E4 allows one to deduce that 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. 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 these 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 million 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.

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.

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


Figure 6: 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 (red → green → yellow → blue), while author 2’s nodes will go through (yellow → green → blue).