An environment for creative problem solving by visualization of massive dynamic data-sets — putting software on the map

Christian Collberg, Stephen Kobourov, John H. Hartman[*]

1 Introduction

In many engineering disciplines a project life-cycle is clearly separated into an early, creative, design phase, and a later, more mechanical, implementation phase. Because of the complex nature of software, the software engineering life-cycle is different. During its lifetime a program may be re-designed multiple times and one version of the design may bear little resemblance to another. Thus, software engineers are involved in a never-ending creative cycle that continuously examines the current product, analyses it for design flaws, logic flaws, or performance flaws, and creates a new design which is more beautiful, more efficient, or less prone to faults.

This proposal is concerned with gathering information — static, dynamic, and historical — about real world software systems and presenting it in a simple and coherent way to engineers in order to facilitate creative problem solving during maintenance and re-design efforts. Our vision is a system that seamlessly integrates what is traditionally thought of as disparate activities: re-factoring, browsing program text, debugging, and performance evaluation and tuning.

Many problem solving endeavors are *open-ended* — we don't initially know exactly what we're looking for. An environment which purports to support creative problem solving must therefore be *flexible* — we can't know ahead of time what problem a user will need to solve. In our domain we must therefore be prepared to collect *all* kinds of information about a program (static, dynamic, and historical) and be prepared to present any combination of these to the engineer.

To support creative problem solving, our system $\mathfrak{C}^5$ (*Creative Cartoghraphic Comprehensive Comprehension of Code*) will therefore present the state and performance characteristics of the program under study, its overall static structure, the relationships between different parts of the code, and the historical development of the program. Without careful consideration, this would present to the engineer a bewildering morass of data from which no useful information could be gleaned. We therefore propose to visualize all aspects of a program under study using one single familiar and powerful metaphor, the *geographic map*.

Problem solving for large software systems is by necessity a collaborative effort. Often, the expertise of many — original designers, performance experts, user interface experts, implementers — need to be harnessed in order to identify the source of a problem and devise an appropriate solution. Our geographic metaphor will allow all experts, regardless of background, to share a common language and common imagery in which to discuss the system under study. We will furthermore design $\mathfrak{C}^5$ to support the use of large multi-touch displays which will allow multiple engineers to simultaneously interact with the visualization.

Our map metaphor will allow us to visualize the development history of the program (its version control database), its current static structure (the packages, modules, methods, and their interconnects), as well as its dynamic (performance) behavior. This is in contrast to previous work, in which different aspects of a program are visualized in different and *ad hoc* ways. We expect $\mathfrak{C}^5$ to be used in many problem

[*]Department of Computer Science, University of Arizona, {collberg,kobourov,jhh}@cs.arizona.edu.

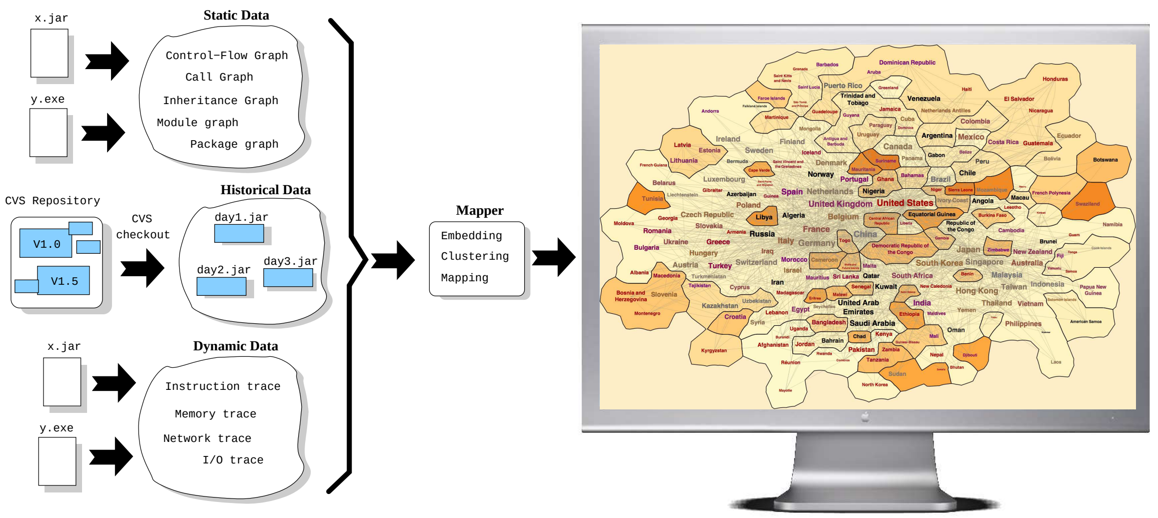

Figure 1: Overview of the proposed $\mathfrak{C}^5$ system.

solving scenarios. For example, an engineer newly transferred to a project can use $\mathfrak{C}^5$ to learn about the program's static structure; a maintenance engineer can use $\mathfrak{C}^5$ to track down bugs in unfamiliar code; a reverse engineer can use $\mathfrak{C}^5$ to gain a quick understanding of legacy code, including an understanding of its development history; a performance tuning engineer can use $\mathfrak{C}^5$ to gain an accurate view of the performance characteristics and bottlenecks of a program; and a re-factoring engineer can use $\mathfrak{C}^5$ to re-design the program from the ground-up. In all cases the engineer will need to learn only *one* tool and be presented with only *one* metaphor of the program and its behavior. The goal is for this unified view to reduce the information load on the engineer and thereby allow him/her to be more creative and efficient in their problem solving.

There are many technical and design challenges in building a system such as $\mathfrak{C}^5$. First of all, collecting detailed run-time information of a program requires tremendous resources. To get an accurate view of a program we need to do *vertical* data collection — profiling and tracing information need to be collected from multiple levels, including the CPU, the instrumented operating system, an instrumented virtual machine, an instrumented program, network packet sniffers, disk performance monitors, etc. One goal of this project is to devise algorithms which allow the collection of every type of performance data, at every level of a computing system, without sacrificing accuracy or interactivity. Figure 1 shows an overview of $\mathfrak{C}^5$.

To summarize, there are five main aspects of this proposal that set it apart from previous work:

1. **Creative:** We will study ways to provide software engineers with a single environment that will support all manners of interactive, collaborative, and creative problem solving.

2. **Cartoghraphic:** We will study how to use the intuitive and familiar metaphor of a geographic map to present the large amounts of data the system collects about a program.

3. **Comprehensive:** We will study how to collect performance data about a software system at all levels, without sacrificing accuracy.

4. **Comprehension:** We will study how to selectively visualize slices of the massive data sets we collect to maximize comprehension and minimize information overload.

5. **Code:** We will study how to collect data about every kind of static structure of a software system, including data about its historical development.

Additionally, we will generalize the system making it easy to deploy in other domains where the design process can be enhanced by visualization of massive data-sets.

2 Overview

Much research has gone into the design of tools for *software visualization*, *program comprehension*, *reverse engineering*, *code refactoring*, *performance profiling*, and *debugging*. Although targeting different aspects of the program life-cycle, these tools all have in common that they seek to discover information about a program, in order to support a particular problem solving activity. Unfortunately, the research communities that study the design of such tools are largely disjoint, and this has prevented a unified view of tools for program understanding and manipulation. The goal of the research proposed here is to study the design of a tool that provides such a holistic view, helping a programmer to gain an understanding of the history, logic, and performance characteristics of a large software system. Such understanding is necessary in order to support a variety of creative problem solving activities. Let us begin by exploring a few scenarios in which $\mathfrak{C}^5$ is helpful.

2.1 Scenario 1 — Static Code Visualization

Wendy has just been hired as a maintenance engineer. Her first task is to familiarize herself with the code base. She fires up $\mathfrak{C}^5$, and takes a fly-through tour of the code. She starts by visualizing the code at a high level. As she flies through the code, she sees packages and classes laid out as a landscape of continents, countries and cities, with roads and highways representing the connections. The recursive structure of the code (packages within packages, etc.) is naturally reflected in the recursive structure of geography (cites within states, states within countries, etc.). Close ties between pieces of code are reflected in their geographic proximity. She zooms in on a particularly large cluster of cities, noticing the highly complex network of roads that connect them. She makes a mental note to herself that this segment of the code may be a prime target for future refactoring. She continues zooming in on one of the cities (classes), examining each city block (method) in turn.

This scenario illustrates *one* of several ways in which we can create a geographic map from the class hierarchy of the underlying program.

2.2 Scenario 2 — Development History Visualization

Terrance, a programmer on a security-sensitive project, disappears under mysterious circumstances. The FBI fears the project may have been compromised, perhaps by the insertion of a Trojan Horse. The project's CVS repository is viewed in $\mathfrak{C}^5$, highlighting all of Terrance's code changes. Did he add a piece of code (a possible Trojan) to a part of the system no longer under active development, to avoid scrutiny by other developers? Unlike in the previous example, here we are looking at a dynamic process, namely the evolution of the program over time. The code is again visualized using a geographic map but this time the map is derived from the CVS history of the program. As code grows and packages and classes split and merge over time, the corresponding countries and cities grow and merge over time. Because we know all the changes going back in time, we can prepare the map in such a way that local code changes result in local map changes, causing any unusual activity to stand out. By overlaying a heatmap, where a change in color indicates high activity, we can easily spot pieces of code directly manipulated by Terrance. Next, the program is repeatedly executed under $\mathfrak{C}^5$ with different inputs. A fly-through of the program "terrain" is viewed during execution, highlighting any code that Terrance wrote, but which is never executed on normal inputs — signs of a possible Trojan.

2.3 Scenario 3 — Performance Visualization

Philip, a performance-tuning engineer for a computer game company, has received reports of a mysterious performance problem when gamers move from level 23 to level 24. He runs the game under $\mathfrak{C}^5$, executing at near full speed. After an hour of play Philip moves to level 24, but notices nothing out of the ordinary. He uses $\mathfrak{C}^5$'s backwards execution facility to go back one minute in time, and continues forward again, this time picking a different way to kill Monster #354. This time, he does notice a remarkable delay in moving to level 24. The map visualization of the game includes several heatmap overlays indicating CPU activity, I/O activity, network traffic, etc. Philip notices that $\mathfrak{C}^5$'s I/O heatmap is glowing red exactly at the time of the performance hit. He goes back in time once more, stepping forward through the program at a more detailed level. During this process he discovers a bug that under certain conditions causes the level 24 scenery to be read from disk multiple times, rather than once, as expected.

2.4 Research Goals

Any specific program analysis task in the scenarios above may require several traditional tools, such as debuggers and profilers, to pinpoint a particular problem. The goal of the proposed research is to design an intuitive and powerful environment that allows an engineer to seamlessly move from task to task without having to move from tool to tool. However, there are many technical challenges that need to be addressed:

- Modern programs (games, simulators, browsers, instant messaging clients) are highly interactive. To do accurate performance measurements, the programmer must be allowed to interact with the program at near real-time speeds. At the same time, performance data needs to be captured at every level, from the CPU and disk to the operating system to virtual machines, and analyzed and visualized for the programmer. This requires high performance of $\mathfrak{C}^5$ itself.

- Fully understanding the dynamic behavior of a program not only means understanding its behavior as time goes forward, but also in reverse. Reverse execution is complicated by the need to accurately reproduce all internal and external events, including operating system scheduling decisions, network traffic, and disk I/O.

- Tremendous amounts of execution and performance data will be gathered during long program runs and presenting this to the programmer in an easily digestible format will be challenging.

- When the system under study is highly interactive (such as immersive simulations or computer games) it will not be possible for one person to simultaneously interact with it and $\mathfrak{C}^5$. It will be necessary to study the interaction between two or more users driving the system under study and driving and monitoring $\mathfrak{C}^5$ itself.

Figure 1 gives a high-level overview of $\mathfrak{C}^5$. Major components of the system are a high-end multi-core machine running the system under study, a large, multi-touch display to visualize and manipulate static and dynamic properties of the system, and tools for steering the computation and visualization.

$\mathfrak{C}^5$ will be able to process x86 executable files as well as Java jar files ⓕ. These can be provided directly by the user or, when historical analysis is desired, automatically extracted and built from a version control system such as CVS, using tools previously built by the PIs [28]. Static analysis tools also previously built by the PIs [36] or publicly available (for example Debray's PLTO [41] system for static analysis of x86 executables) will analyze and extract static information about the user program. Performance data will be extracted from every level of the system under study. The data will be transformed into several multi-level maps, representing the underlying program's control-flow graph, method call graph, inheritance class graph, etc. In addition, map overlays representing different activity levels (e.g., CPU, I/O, network, etc.)

will be available to provide a comprehensive view of the program. These overlays can be presented as *small multiples*, that is, as a series of small similar pictures, making a point through repetition. The pictures can be heatmaps or 3D terrains, with color or spikes indicating high activity levels.

3 Proposed Research

In this section we describe the idea of using geographic maps to visualize large and complex data arising in the context of real world software systems. We then consider three specific examples highlighting our approach to dealing with static, historical, and dynamic data. We finally consider some of the research problems and technical challenges associated with accomplishing this work.

3.1 Geographic Maps for Comprehensive Software Visualization

Software visualization presents performance profiles and static structures of programs, visualized as tables of data, histograms, hair-ball graph drawings, etc. So far, there have been no unified model for visualizing static, dynamic, and historical data about a program. Such a unified visualization metaphor should naturally support nesting of concepts since programs are inherently hierarchically structured. The metaphor must also be "extensible" and support many different types of visualization, since visualization requirements change over time and may be particular to certain applications. Our overall goal is to create a representation which makes the underlying data understandable and visually appealing. *Geographic maps* are familiar and intuitive and seem to fit the bill perfectly. In addition, maps are inherently hierarchical (cities within states within countries within continents, etc.), and there are many tools for manipulating geographic data. For these reasons, we will investigate the use of the geographic map metaphor for visualizing, browsing, and interacting with a program.

In the past, software visualization has used drawings of *graphs* to present many aspects of programs. Graphs capture relationships between objects and graph drawing allows us to visualize such relationships. Typically vertices are represented by points in two or three dimensional space, and edges are represented by lines between the corresponding vertices. The layout optimizes some aesthetic criteria, such as, showing underlying symmetries, or minimizing the number of edge crossings. While such point-and-line representation are most commonly used, other representations have also been considered. For example, treemaps [127] use a recursive space filling approach to represent trees. There is also a large body of work on representing planar graphs as contact graphs [40,82,94], where vertices are embodied by geometrical objects and edges are shown by two objects touching in some specified fashion. Graph representations of side-to-side touching regions tend to be visually appealing and have the added advantage that they suggest the familiar metaphor of a geographic map.

In the context of visualizing programs, we would need to represent general graphs as maps. Clearly, there are theoretical limitations to what graphs can be represented exactly by touching regions, namely, subclasses of planar graphs. However, our aim here is practical rather than graph theoretical. We do not insist that the created map be an exact representation of the graph but that it captures the underlying relationships well. With this in mind, we do not insist that all vertices are represented by individual polygons either. In fact, we group closely connected vertices into regions, with the help of clustering and colors differentiating different clusters.

Given a graph representing a program structure (e.g., a control-flow graph) we will produce a geographic map using a 3-step process: *embedding, clustering, mapping*. The embedding step determines the placement of the vertices in the plane; see the top of Figure 2. The clustering step determines groups of related vertices, and the mapping step creates regions around the clusters, and colors them with map-like colors; see the bottom left of Figure 2. Our maps will have a "natural" map-like look, outer boundaries that follow the

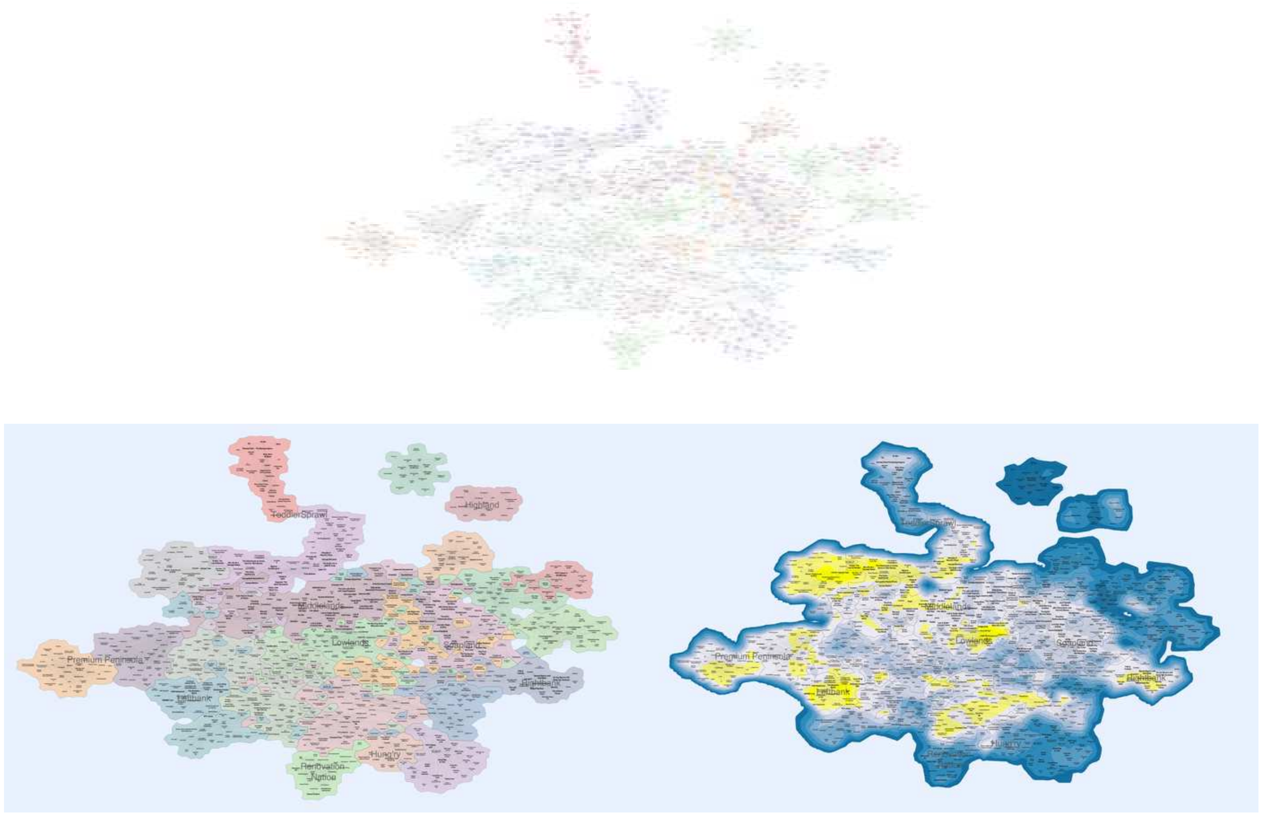

Figure 2: A graph, a map of the same graph, and a heatmap overlay.

outline of the vertex set, and inner boundaries having the twists and turns found in real maps. Our maps will also have lakes, islands, and peninsulas, similar to those found in real geographic maps. Once the continents and islands have been determined, different overlays such as heatmaps or 3D terrains, we be used to display various properties associated with the objects on the map (e.g., CPU activity, I/O activity, network traffic); see the bottom right of Figure 2. We give further details about generating maps in Section 3.3.1.

We believe that maps offer a familiar and intuitive way to present data. Informal studies of the way users interact with traditional graph drawings and maps of the same data seem to indicate different interaction patterns. Specifically, users are more likely to stop by and examine a poster of a map that than the exact same data drawn as a graph. Moreover, users tend to spend significantly longer time studying the maps, and they find non-trivial structural information without any prompting. We plan to perform formal user studies of the effectiveness of the geographic map metaphor in the context of software visualization.

3.2 Visualization examples

In this project we will examine the effectiveness of the map metaphor to visualize all kinds of static and dynamic software data. Below we give some examples of such visualizations and how they can be used to aid in the scenarios we showed in Section 2.

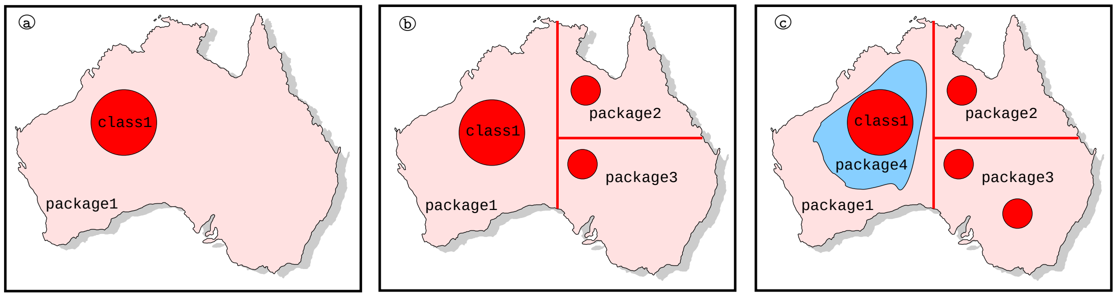

Figure 3: Visualizing the historical development of a program.

3.2.1 Visualizing Historical Development

In Section 2, Scenario 2, Terrance' code is visualized from a historical perspective using $\mathfrak{C}^5$. The natural way of doing this is to employ animation of static code layouts. In Figure 3 we show how the historical development of a Java program can be visualized geographically. In ⓐ, the program starts out as one class `class1` (a city) within a package `package1` (a country). A continent represents the entire program. In ⓑ, the program has developed into three classes, each within its own country. In ⓒ, `class1` now resides within the package `package1.package4`, and by slight abuse of the geographic metaphor we visualize this as a country within a country[1].

Our previous work on visualizing the historical development of a program [28] extracted a Java program (SANDMARK, currently 120,000 lines of Java code) from its CVS database. One version was extracted per day, the program was built (compiled to Java bytecode), and control-flow graphs, inheritance graphs, and static call-graphs were constructed. These graphs were then visualized using a temporal graph-drawing system [68]. Tree-structures (such as inheritance and package graphs) will be easy to visualize as a geographic map, as illustrated in Figure 3. Recall that for our purposes a map is an embedded graph + clustering + mapping. The clustering and mapping determine what vertices are grouped in what countries and how these countries look (shape, color, etc.) but the embedding of the graph determines the positions of the vertices in the plane. Thus one of the main problems in creating a good map is that of finding a suitable embedding of the graph. For example, the structure of the graph and the clustering of the graph can be used to guide the graph embedding. In the case of static call graphs the clustering is defined by the hierarchy: classes (cities) contain methods, methods (city blocks) are connected by call edges (roads), and cities are located to minimize distance between tightly coupled classes.

3.2.2 Visualizing Performance Profiles

In Section 2, Scenario 3, Philip used $\mathfrak{C}^5$ to visualize performance. Our goal is to be able to overlay this information on the static visualization of the code as described above. We plan to use the idea of *small multiples* for visualizing several performance parameters based on the same underlying geographic map. Once a particular geographic map has been created for a program, the user quickly gets used to that layout. This allows us to show several parameters of the program with several heatmap overlays. These overlays will be smaller in size and on the same scale, allowing for an easy overview of what activity is taking place and where it is taking place. Current page fault rate, amount of freed memory after garbage collections, number of active threads, and current CPU utilization are examples of such information.

[1]While unusual, countries within countries do exist. Examples include Lesotho within South Africa, and the Hopi reservation within the Navajo reservation within Arizona within the United States.

In this case, we are concerned with the visualization of truly dynamic graphs that change in real-time. A graphs that is notoriously difficult to deal with, yet can aid greatly in the discovery of dynamic allocation patterns, is the heap-graph. All programmers are familiar with "pointer-bugs," "memory leaks," and "heap fragmentation." In long-running programs the heap can contain millions of objects which makes visualization a formidable task. Furthermore, the heap changes frequently which means that there may be much activity between time-slices that needs to be presented to the user.

Visualizing the heap graph can also be combined with visualization of the call-graph and the inheritance graph, to dampen the effect of the large number of changes that occur in the heap graph. There are several possible ways to visualize such data using our geographic map metaphor. Using several heatmap overlays provides a simple 2D visualization that allows for a quick overview of high-level trends. Alternatively, 3D terrain overlays can be used, with spikes drawing the attention to the areas with high activity. Activity levels can also be expressed as the amount of light given off by a city, or as the traffic flow on a highway (method call interconnects).

3.2.3 Visualizing Static Structures

Much research has been done within the software visualization community to visualize different aspects of source code. This includes static call graphs, inheritance graphs, control-flow graphs, package graphs, identifier cross-references, program slices, data flow, etc. PI Collberg's SandMark system for code obfuscation and software watermarking, for example, uses standard graph-drawing techniques to view control-flow graphs and inheritance graphs, in order to allow simulated manual attacks against software watermarks. Standard visualization techniques become impractical when the structures to be visualized grow large. In particular, drawings of graphs with more than a few hundred vertices fail to convey much information, except when the underlying structures are very simple (trees) or regular (meshes). Building on earlier work on visualization of large graphs [70,71] we will explore the use of compound-fisheye views [5] and multi-dimensional, multi-scale techniques [47,48].

In Section 2, Scenario 1, Wendy uses $\mathfrak{C}^5$ for a tour of a legacy system with which she needs to familiarize herself. There is a multitude of relationships between code structures for Wendy to explore, and a multitude of ways to visualize these relationships using our geographic metaphor. The goal of this part of the project is to explore different uses of the metaphor to find out which layouts give the best visualization experience, and to develop algorithms for realizing these layouts. As an explicit example, consider the problem of simultaneously visualizing a Java package graph, inheritance graph, and static class call graph. All three graphs share the same set of vertices (classes). Edges in the call graph correspond to classes whose methods call each other. Edges in the inheritance graph correspond to the inheritance relation. The package graph is different in that it is a containment hierarchy. We would like to embed the vertices (classes) in the terrain so as to take into consideration all three relationships. To capture the two relationships defined by the inheritance and call graphs we can adopt our simultaneous graph embedding techniques [18,59,61] to the geographic metaphor. Generic graph visualization algorithms attempt to create layouts where distances between vertices in the layout correspond to the graph theoretic distances in the abstract graph. Algorithms for simultaneous graph embedding extend this notion to multiple graphs on the same set of vertices. Thus, we can obtain layouts where vertex locations are determined by more than one abstract graph.

$\mathfrak{C}^5$ will process Java jar files with SandMark [36] to extract any information about the code that could be visualized. PLTO [41] will be used to extract similar data from x86 executables. A visualization engine decides which data should be presented based on user guidance. The final step is to render appropriate geographic scenes and animation, and supporting user interactivity such as zooming, fly-throughs, etc.

3.3 Technical Challenges

There are several technical challenges in designing, developing, and deploying $\mathfrak{C}^{\mathbf{5}}$. First is the overall challenge of incorporating the vast amounts of data collected by $\mathfrak{C}^{\mathbf{5}}$ into geographic maps that are comprehensible and useful to the user. Second is the challenge of collecting the necessary data efficiently and unobtrusively, while affecting the monitored program as little as possible. Third is the challenge of analyzing and correlating the collected data and presenting the results via maps so as to improve the user's comprehension of the program being studied. We cover each of these in the following subsections.

3.3.1 Generating Maps

The input to the map-generating algorithm is a relational data set from which we extract a graph $G = (V, E)$. The set of vertices V corresponds to the objects in the data, e.g., methods in a program, and the set of edges E corresponds to the relationship between pairs of objects, e.g., one method calls the other. In its full generality, the graph is vertex-weighted and edge-weighted, with vertex weights corresponding to some notion of the importance of a vertex and edge weights corresponding to some notion of the distance between a pair of vertices.

The first step in our algorithm is to embed the graph in the plane. Possible embedding algorithms include principal component analysis [86], multidimensional scaling (MDS) [93], a force-directed algorithm [69], or non-linear dimensionality reduction such as LLE [125] and Isomap [132].

The second step is a cluster analysis of the underlying graph or the embedded pointset from step one. In this step, it is important to match the clustering algorithm to the embedding algorithm. For example, a geometric clustering algorithm such as k-means [95] may be suitable for an embedding derived from MDS, as the latter tends to place similar vertices in the same geometric region with good separation between clusters. On the other hand, with an embedding derived from a force-directed algorithm [69], a modularity based clustering [107] could be a better fit. The two algorithms are strongly related [108] the recent findings therefore we can expect vertices that are in the same cluster to also be physically close to each other in the embedding.

In the third step the two-dimensional embedding together with the clustering are used to create the map. In the final step, the countries are colored using a coloring algorithm to maximize color differences between neighboring countries.

To make maps more useful for interactive exploration of large underlying data sets we plan to incorporate topological clustering which would allow us to show the map in varying level of detail. We can leverage previous work on large graph visualization such as topological fisheye views [72] and the related compound fisheye views [5].

3.3.2 Comprehensive and Efficient Gathering of Performance Data

Monitoring a program as it runs must be both complete and non-intrusive. The monitoring must collect data from multiple sources, such as the CPU (performance monitors), an instrumented operating system, an instrumented virtual machine (such as the Java virtual machine, JVM), an instrumented program (Java bytecode, TCL scripts, binary executables), network packet sniffers, disk performance monitors, the Java profiling interface (JVMPI), etc.; see Figure 1 ⓑ. Monitoring must also be low-overhead so as to not disturb the program being monitored. Traditionally, this means refraining from collecting some data (potentially reducing the accuracy of the analysis), or resorting to a post-mortem analysis which does not work well for highly interactive programs.

Efficient Simultaneous Simulation. We propose to solve this problem by running the program inside a virtual machine that allows $\mathfrak{C}^{\mathbf{5}}$ to monitor the program without adversely affecting it. By using multiple virtual machines, each running on a different core, $\mathfrak{C}^{\mathbf{5}}$can not only run the program at different points in its execution, can also vary the amount of detail in the collected information. Simulating and tracing system behavior at various levels of detail is not a new idea. SimOS [124], for example, is a well-known system that provides this functionality. However, existing systems present a tradeoff between accuracy and speed that must be decided by the user. Resimulating portions of the execution at a greater level of detail is useful, but it forces the user to decide which portions of the execution to resimulate and at what level of detail, and then to wait for the results.

We propose to solve these problems by running the simulations concurrently on multiple cores. The program itself runs on a subset of the cores and interacts with the user. Periodic checkpoints of the program's state are used to direct more detailed simulations of the program that run on additional cores. These checkpoints, along with necessary information collected from the monitored program, allow the simulations to faithfully replicate the program's behavior.

Creating Determinism. Simultaneously simulating different portions of the program's execution requires that the program be deterministic, so that every simulation behaves exactly like the program. We will leverage off the work done by Time-traveling virtual machines (TTVM) [89] to collect the necessary information so that that simulations used by $\mathfrak{C}^5$ are deterministic. We will augment the TTVM technology with *continuous data protection* (also called *continuous backup*), a storage industry term for fine-grain time-travel within a storage system.

Vertical Profiling. We refer to the analysis of data from multiple levels in the system as *vertical profiling* [81,131]. There are two aspects of the problem. First, the system must collect enough information to allow correlation between events at different levels. For example, poor browser performance may be caused by poor resource utilization in the lower-level Javascript interpreter. Second, the system must record enough context information for an event to permit analysis. This context information might be found at a different level from the one in which the event occurs. For example, the excessive disk traffic might be caused by excessive garbage collection in the JVM, which in turn might be caused by a particular web page.

SimOS solves this problem through *annotations*, snippets of code that are run when an event occurs to collect the proper information from the system. We will adopt a similar strategy in $\mathfrak{C}^5$. These snippets will necessarily be specific to the system being monitored; we plan to investigate generating them automatically through static analysis of the code.

3.3.3 Dynamic Program Comprehension

In $\mathfrak{C}^5$, *low-level comprehension* (information gained from dynamic analysis) *statistical comprehension* (information gained from from profiling), *global comprehension* (information gained from static analysis), and *historical comprehension* (information gained from CVS repositories), all contribute to a complete understanding of the behavior of a program.

Dynamic analysis (debugging) and statistical analysis (profiling) are often seen as distinct, but in reality they both require executing the program and analyzing the artifacts of the execution. Debugging a program requires executing it at high-levels of detail, perhaps one instruction at a time, while examining code, events, and data. This allows the programmer to gradually build up an understanding of the low-level control-flow of the program. Profiling provides a statistical comprehension of the program's behavior, such as the number of times a routine is called and the amount of time spent in the routine.

In contrast, static analysis is performed before the program runs. Using call-graphs as an example, a static analyzer can produce a conservative estimate of a program's call graph, whereas dynamic analysis produces the exact graph for a particular input. $\mathfrak{C}^5$ will integrate the results of static and dynamic analysis. For example, this will allow a programmer to ask $\mathfrak{C}^5$ to: "stop the program at the entry of every function that could *possibly* be called from function `foo`." Much of the required functionality (such as computing static slices) is available in the PI's SANDMARK [36] tool and Debray's PLTO [41] tool.

$\mathfrak{C}^5$ will also use historical information to facilitate program comprehension, by extracting information from source code control systems. A useful feature in $\mathfrak{C}^5$ will be the ability to execute two or more closely related versions of a program in lock step, on the same user input, highlighting any differences in control-flow or data values. This *relative debugging* feature [6] will allow a programmer to run yesterday's version of a program ("where feature X was known to work") simultaneously with the current version (where X is broken), focusing on the differences in behavior between the two. These differences might be represented as heat values on a common map representing the two versions of the program. Again, since $\mathfrak{C}^5$ performs extensive static analysis, static information can also be integrated in the process, visualizing simultaneously the differences between static structures and behavior.

To support interactivity and long-running programs, $\mathfrak{C}^5$ will be able to run programs reverse. $\mathfrak{C}^5$ monitors a running program and collects enough information to fully analyze its behavior, including running it in reverse. The general technique is to checkpoint the state of the virtual machine periodically (either on disk or in memory), then use these checkpoints to initialize the program state to just before the target instruction. The debugger then rolls the program forward until it reaches the target instruction, providing the illusion of running the program in reverse.

4 Preliminary Work

The research proposed here builds on, extends, and utilizes infrastructure from previous work by the PIs. The three co-PIs have extensive background in data visualization, software visualization, performance analysis, operating systems and virtualization, manipulation of code at the native and virtual machine code levels, and in the design of large software systems:

code manipulation: PI Collberg has extensive experience in the manipulation of code at the virtual machine and binary code levels. In [24,29–32,36] he describes a tool (SANDMARK) for watermarking, obfuscating, measuring, reverse engineering, and visualizing Java bytecode. SANDMARK contains a rich library for manipulating Java bytecode (control-flow analysis, data-flow analysis, slicing, etc.) which will form the basis for the Java portion of $\mathfrak{C}^5$. In [24] the PLTO [41] x86 binary editing system is used to watermark binary executables. In [37,38] a system for automatic extraction of semantics of machine code instructions is described.

software visualization: In [39], PI Collberg describes a system for high-quality text-and-graphic renderings of source code. We will extend these results when designing the part of $\mathfrak{C}^5$ that views and browses low-level code. In [28], PIs Kobourov and Collberg show how historical data about a program can be extracted from a CVS revision control repository, and visualized using graph-drawing techniques. The system allows easy identification of the parts of a system that have been in long-term flux and may need to be redesigned. The system also makes it possible to identify which authors have been involved with different phases of the implementation. In [52] we study the behavior of dynamically modifiable code. We will build on these experiences to visualize programs using $\mathfrak{C}^5$'s more natural geographic metaphor.

multi-user interaction: In [35], PIs Kobourov and Collberg describe the first non-trivial application, a multi-player version of Tetris, for the DiamondTouch hardware. DiamondTouch [43] is a touch sensitive multi-user input device developed by Mitsubishi. Follow-up work includes a study of the performance of collaborative spatial/visual tasks under different input configurations [23] and an interactive multi-user system for simultaneous graph drawing that takes advantage of direct physical interaction of several users working collaboratively [90]. We will use these experiences in designing the multi-user interaction facilities of $\mathfrak{C}^5$.

graph drawing: PI Kobourov has experience in graph drawing [50,53,70,71] visualization of large graphs [5, 47–49], evolving graphs [28,52,58,62], graph theory [7,13,25,51], and computational geometry [45, 46,54]. Graph-drawing techniques and computational geometry will be important in the design of algorithms for $\mathfrak{C}^5$'s geographic visualization of code.

performance analysis: PI Hartman has published several papers on performance analysis, including SMP lock contention [78], the effect of mobile code on server performance [128], and optimizing TCP forwarder performance [129].

operating systems: PI Hartman has helped design and implement several complete operating systems including Sprite [112], Scout [99], and Joust [75,80]. He has also worked on operating system support for SMP [78], active routers [116], and PlanetLab [100]. He has done extensive research in file and storage systems, all of which involve network protocol and operating system development, including RAID-II [44], Zebra [79], Swarm [76,77,101], Gecko [8,10], and Mirage [9].

5 Related Work

5.1 Software Visualization

Software visualization aim to improve human understanding of computer programs by portraying them in a form that is more readable than mere source code [103,104,118,123].

BALSA [19] and Zeus [20] annotate a program with hooks so that "interesting events" such as changes to data structures and subroutine calls and returns can be relayed to the visualization system. SHriMP [130] shows inheritance hierarchies and aggregation using a variety of graphical views. Young [138] visualizes dense call-graphs as stacks of cubes, viewing the result in a virtual reality environment.

Many tools visualize historical information stored by change management systems such as CVS [11,12, 22,55,85,97,134]. SoftChange [97] produces textual reports of the complexity, size, purpose and author of program changes. Ball [11,12] describes tools that visualizes many different aspects of software using line, pixel and hierarchical representations. Eick [55] visualizes software changes using bar-graphs, pie-charts, matrix views, and cityscape views.

Integrated systems for program understanding [56,87,109,110,137] include RevEngE [137] (support for pattern matching, redundancy analysis, and reverse engineering), Gammatella [87,110] (storing and visualizing program-execution data using a treemap view), and the "War Room Command Console" [109] (eight linked consoles to present system development information). Eng [56] proposes a visualization framework to provide static and dynamic program visualizations.

Tools for visualizing dynamic behavior or profiling information [16,42,73,114,120–122,135,139] include EVolve [135] (visualization of the runtime behavior of Java programs using barcharts and dotplots), OverView [42] (an event-based Eclipse plug-in for runtime visualization of distributed systems), Desert [120] (customizable visualizations of dynamic call counts, dynamic call traces, load and store traces), and FileVis [139] (high-level overviews of a software system using 3D graphics and VR technology).

Aesthetic computing [1,67,117] addresses artistic issues in visualization. The RUBE system [67] allows for model visualization with different metaphors (e.g., event graphs visualized using a cityscape metaphor). The RelVis system [117] uses Kiviat diagrams using a tree-ring visualization to provide integrated graphical views of source code and release history.

5.2 Graphs as Maps

There is little prior work on representing general (non-planar) graphs as geographic maps. In geography there are many papers about accurately and appealingly representing a given geographic region, or on re-drawing an existing map subject to additional constraints. Examples of the first kind of problem are found in traditional cartography, e.g., the 1569 Mercator projection of the sphere onto 2D Euclidean space. Examples of the second kind of problem are found in cartograms, where the goal is to redraw a map so that the country areas are proportional to some metric, an idea which dates back to 1934 [119] and is still popular today (e.g., the New York Times red-blue maps of the US, showing the presidential election results in 2000 and 2004 with states drawn proportional to population).

The map of science [17] uses vertex coloring in a graph drawing to provide an overview of the scientific landscape, based on citations of journal articles. Treemaps [127], squarified treemaps [21] and the more recent newsmaps [136] represent hierarchical information by means of space-filling tilings, allocating area proportional to some metric.

Representing imagined places on a map as if they were real countries also has a long history, e.g., the 1930's Map of Middle Earth by Tolkien [133] and the Bücherlandes map by Woelfle from the same period [2]. More recent popular maps include xkcd's Map of Online Communities [3]. While most such maps are generated in an *ad hoc* manner and are not strictly based on underlying data, they are often visually appealing.

In self-organizing maps (SOM) [92] an unsupervised learning algorithm places objects on a two-dimensional grid such that similar objects are close to each other. SOM also uses a map metaphor to visualize the resulting embedding. but in SOM the "map" is created by coloring cells of the grid based on a feature value, therefore operating on a discrete grid space, without a clear inner boundary between "countries".

Generating synthetic geography has a large literature, connected to its use in computer games and movies. Most of the work (e.g., [102]) relies on variations of a fractal model. These techniques could provide additional photorealism, and may be used in future extension of our work.

5.3 Profiling

There are two principal methods of collecting profiling information. Code instrumentation approaches such as `gprof` [74] insert instructions at compile-time or post-link-time, counting procedure invocations, basic block executions, branches taken, etc. Sampling-based systems such as the PCT system [14] periodically suspend the program and examine the program state. PCT's basic philosophy is similar to this proposal, in that profiling is viewed as a form of debugging.

The Java virtual machine profiling interface (JVMPI [4]) is used by the Pajè [111] and JaViz [88] systems to collect traces of distributed systems that are then visualized (for example, as call-graphs) off-line.

To correctly capture the performance of a system execution information has to be collected at multiple levels (Figure 1 ⓑ). Sweeney, Hauswirth, et al. [81,131] integrate information from an instrumented JVM with data from hardware performance counters, allowing performance to be visualized off-line.

5.4 Debugging

Early work on debugging is surveyed by Paxon [115] and McDowell *et al.* [96,115]. There are four basic implementation techniques to enable reverse debugging: *logging*, *checkpointing*, *forking*, and *instrumentation*. *Logging* was introduced by Zelkowitz [140], in the first reverse execution system. Moher's PROVIDE [98] similarly logs variable updates but was limited to short-running programs. Netzer [106] compresses the log at the cost of less flexible backward motions. Feldman and Brown's IGOR [66] system uses *checkpointing* to write program state (modified pages and open file descriptors) to disk at fixed intervals. Pan and Linton's Recap [113] system periodically *forks* off a child process that holds the state of the program at that point. Boothe [15] uses a similar idea but also periodically thins out the forked processes to reduce memory usage. Backwards execution also requires deterministic re-execution. Boothe [15] stores the return values of system calls and replays them during re-execution. Recap [113] stores the times of system calls, signals, and shared memory accesses, enforcing deterministic re-execution.

Abramson and Sosič [6] present a system for *relative debugging*. The idea is to run a version of a program known to work in parallel with a new, buggy, version and display the differences in the data structures.

6 Results from Prior NSF Support

6.1 Collberg

Collberg is PI on NSF grant CCR-0073483, entitled *Code Obfuscation, Software Watermarking, and Tamper-Proofing — Tools for Software Protection*, $265,000, September 1, 2000–August 31, 2004. This project resulted in papers describing techniques for constructing error-correcting graphs [33], an overview of software protection techniques [29], a description of the SANDMARK software protection infrastructure [36], novel software watermarking algorithms [24], SANDMARK's obfuscation executive [83], and evaluation of published software protection techniques [34,105,126]. Three bachelor's and one master's thesis have been completed, and one PhD thesis is expected to complete this year.

6.2 Kobourov

Kobourov is PI on NSF grant ACR-022920, entitled *Visualization of Giga-Graphs and Graph Processes*, for the period September 2002 through August 2005, $240,358. This project has resulted in a number of publications on visualization of large graphs and graphs that evolve through time [18,28,52,58–60,65, 68] and several software systems for graph visualization, including GraphAEL [57] (for visualization of computing literature) and GMorph [62] (for intersection-free morphing of planar graphs). Many students have been involved in this research, both of the graduate and undergraduate levels. Cesim Erten co-authored ten papers [18,53,57–59,61–63,65,84] and completed a PhD on this topic in 2004 [64]. Four MS students were co-authors on papers related to the project [5,23,28,57,58,62,63,68,90,91]. Ten of our publications have at least one undergraduate co-author [23,25–28,35,52,57,58,61,68].

6.3 Hartman

Hartman is PI on NSF grant CNS-0834179, entitled *Virtual Watts: Energy Management in Virtual Environments*, for the period August 2008 through August 2011. This project focuses on the intersection of energy management and virtual machine environments. It investigates the proper distribution of functionality between the guest operating system, virtual machine, and underlying virtual machine monitor so as to improve overall system energy efficiency. All guest operating systems must collaborate with the virtual machine monitor to exchange information about current context and the device state. System-wide energy efficiency requires global information, as is available in non-virtualized systems. The challenge is to provide the desired information flow while preserving the abstraction provided by the virtualization. Solving this challenge is critical since future systems will rely heavily both on energy efficiency to prolong battery life and reduce energy costs, and on virtualization to provide isolation, portability, and many other benefits.

7 Development Plan, Evaluation, and Broader Impacts

7.1 Development Plan

The proposed research will be carried out over a three-year period. It involves a significant amount of design, development, and evaluation. We request support for two PhD-level graduate students and one post-doc to assist the three PIs in carrying out this research. We anticipate the following development plan:

Year 1: During the first year we will focus on static analysis (Scenario 1 in Section 2). We will develop a prototype system that uses maps to enable static analysis on a large-scale display. We will also begin research on historic and dynamic analysis, specifically developing the infrastructure for running multiple simultaneous simulations on multiple virtual machines. This will primarily focus on checkpoint and continuous data protection technologies.

Year 2: During the second year we will extend our work on static analysis to include historic analysis (Scenario 2 in Section 2). We will develop better visualization technologies based on our results from the first year, as well as develop a prototype dynamic visualization system. We will also extend our virtual machine infrastructure to include vertical profiling support and improve the overall infrastructure based on our results from the first year.

Year 3: During the third year we will integrate static, historic, and dynamic analysis into a single integrated tool. We will complete our work on simultaneous multi-level simulation and visualization. We will investigate extending our techniques to other domains, such as visualization of client/server systems and peer-to-peer systems.

7.2 Evaluation

The goal of the proposed work is to answer two questions: (1) *Is program comprehension improved using maps to represent various program characteristics and behaviors, if massive high-resolution touch-screen real-estate is available?* (2) *Can software be more thoroughly analyzed statically, dynamically, and historically using the multi-core machines that are commonplace?* These questions can be reformulated as “will the integrated environment provided by $\mathfrak{C}^5$ allow programmers to find logic and performance bugs more effectively than currently available tools?” and “how much hardware resources are necessary to collect, store, analyze, and visualize data about a program while maintaining an adequate interactive experience for the programmer?” We will collaborate with psychologists in designing and performing effective user studies to answer these questions.

The first question could be addressed by a study in which we vary the visualization methods and measure their effectiveness by keeping track of the time users need to solve a particular problem. The second question could be addressed by varying the amount of resources available to $\mathfrak{C}^5$, while running test scenarios similar to those in Section 2.

7.3 Broader Impacts

The work will be carried out by the investigators, graduate students, and undergraduate students. Undergraduates have played key roles in our recent projects, as discussed in Section 6, and we plan to continue employing them.

This project will foster the development and education of the participating students and will influence the curriculum development of the PIs. We will also make all software developed as part of the project available to the broader research community.

8 References

References

[1] Aesthetic computing. Dagstuhl Seminar #2291, 2003.

[2] Karte des Bücherlandes. `strangemaps.wordpress.com/2009/04/08`.

[3] Map of online communities. `xkcd.com/256`.

[4] Java virtual machine profiler interface. *IBM Systems Journal*, 39(1), 2000.

[5] J. Abello, S. G. Kobourov, and R. Yusufov. Visualizing large graphs with compound-fisheye views and treemaps. In *12th Symposium on Graph Drawing (GD)*, pages 431–442, 2004.

[6] D. Abramson and R. Sosic. Relative debugging using multiple program versions. In *8th Int. Symp. on Languages for Intensional Programming*, Sydney, May.

[7] Baruch Awerbuch and Stephen G. Kobourov. Polylogarithmic-overhead piecemeal graph exploration. In *Proceedings of the 11th Annual ACM Conference on Computational Learning Theory (COLT)*, pages 280–286, 1998.

[8] Scott Baker and John H. Hartman. The design and implementation of the Gecko NFS Web proxy. *Software–Practice and Experience*, 31(7):637–665, 2001.

[9] Scott Baker and John H. Hartman. The Mirage NFS router. In *Proceedings of the 29th IEEE Conference on Local Area Networks*, Tampa, FL, November 2004.

[10] Scott M. Baker and John H. Hartman. The Gecko NFS Web proxy. *Computer Networks*, 31(11–16):1724–1736, 1999.

[11] 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.

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

[13] T. Biedl, E. Demaine, C. A. Duncan, R. Fleischer, and S. G. Kobourov. Tight bounds on maximal and maximum matching. *Journal of Discrete Mathematics*, 285(1):7–15, 2004.

[14] C. BLAKE and S. BAUER. Simple and general statistical profiling with pct. In *Usenix Technical Conference*, 2002.

[15] Bob Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 299–310. ACM Press, 2000.

[16] Robert Bosch, Chris Stolte, Diane Tang, John Gerth, Mendel Rosenblum, and Pat Hanrahan. Rivet: a flexible environment for computer systems visualization. *SIGGRAPH Comput. Graph.*, 34(1):68–73, 2000.

[17] Kevin Boyack, Richard Klavans, and Katy Borner. Mapping the backbone of science. *Scientometrics*, 64(3):351–374, 2005.

[18] P. Brass, E. Cenek, C. A. Duncan, A. Efrat, C. Erten, D. Ismailescu, S. G. Kobourov, A. Lubiw, and J. S. B. Mitchell. On simultaneous graph embedding. *Computational Geometry: Theory and Applications*. To appear in 2005. (Preliminary version in *8th Workshop on Algorithms and Data Structures (WADS)*, p. 243-255, 2003).

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

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

[21] Mark Bruls, Kees Huizing, and Jarke van Wijk. Squarified treemaps. In *Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42. Press, 1999.

[22] Michael Burch, Stephan Diehl, and Peter Weißgerber. Visual data mininng in software archives. In *Proceedings of 2005 ACM Symposium on Software Visualization (Softviz 2005)*, pages 37–46, St. Louis, Missouri, USA, May 2005.

[23] Justin Cappos, Stephen G. Kobourov, Mark Miles, Martin Stepp, Kyriakos Pavlou, and Amanda Wixted. Collaboration with diamondtouch. In *10th International Conference on Human Computer Interaction (INTERACT)*. To appear in 2005.

[24] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, June 2004. (To appear).

[25] C. Collberg, S. G. Kobourov, E. Carter, and C. Thomborson. Error-correcting graphs for software watermarking. In *29th Workshop on Graph Theoretic Concepts in Computer Science (WG)*, pages 156–167, 2003.

[26] C. Collberg, S. G. Kobourov, J. Louie, and T. Slattery. SPLAT: A system for self-plagiarism detection. In *Proceedings of the IADIS Conference WWW/Internet*, pages 508–514, 2003.

[27] C. Collberg, S. G. Kobourov, J. Miller, and S. Westbrook. Algovista: A tool to enhance algorithm design and understanding. In *7th Annual Symposium on Innovation and Technology in Computer Science Education (ITICSE)*, pages 228–228, 2002.

[28] C. Collberg, S. G. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *ACM Symposium on Software Visualization (SoftVis)*, pages 77–86, 2003.

[29] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. Technical Report TR00-03, The Department of Computer Science, University of Arizona, February 2000.

[30] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proc. 1998 IEEE International Conference on Computer Languages*, pages 28–38.

[31] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Sciences, The University of Auckland, July 1997.

[32] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. 25th. ACM Symposium on Principles of Programming Languages (POPL 1998)*, pages 184–196, January 1998.

[33] Christian Collberg, Edward Carter, Stephen Kobourov, and Clark Thomborson. Error-correcting graphs. In *Workshop on Graphs in Computer Science (WG'2003)*, June 2003.

[34] Christian Collberg, Andrew Huntwork, Edward Carter, and Gregg Townsend. Graph theoretic software watermarks: Implementation, analysis, and attacks. Technical Report TR04-06, Department of Computer Science, University of Arizona, 2004.

[35] Christian Collberg, Stephen G. Kobourov, Steven Kobes, Ben Smith, Stephen Trush, and Gary Yee. Tetratetris: An application of multi-user touch-based human-computer interaction. In *9th International Conference on Human-Computer Interaction (INTERACT)*, pages 81–88, 2003.

[36] Christian Collberg, Ginger Myles, and Andrew Huntwork. SandMark — a tool for software protection research. 1(4):40–49, July-August 2003.

[37] Christian S. Collberg. Reverse interpretation + mutation analysis = automatic retargeting. In *Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997. SIGPLAN, ACM. www.cs.auckland.ac.nz/ collberg/Research/Publications/Collberg97b/index.html.

[38] Christian S. Collberg. Automatic derivation of compiler machine descriptions. *ACM Trans. Program. Lang. Syst.*, 24(4):369–408, 2002.

[39] Christian S. Collberg, Sean Davey, and Todd A. Proebsting. Language-agnostic program rendering for presentation, debugging and visualization. In *VL '00: Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*, page 183, Washington, DC, USA, 2000. IEEE Computer Society.

[40] H. de Fraysseix, P. O. de Mendez, and P. Rosenstiehl. On triangle contact graphs. *Combinatorics, Probability and Computing*, 3:233–246, 1994.

[41] Saumya Debray, Benjamin Schwarz, Gregory Andrews, and Matthew Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Rewriting (WBT-2001)*, September 2001.

[42] Travis Desell, Harihar Narasimha Iyer, Carlos A. Varela, and Abe Stephens. Overview: A framework for generic online visualization of distributed systems. *Electr. Notes Theor. Comput. Sci.*, 107:87–101, 2004.

[43] Paul H. Dietz and Darren Leigh. Diamondtouch: A multi-user touch technology. In *Proceedings of 14th Annual ACM Symposium on User Interface and Software Technology (UIST'01)*, pages 219 – 226, November 2001.

[44] Ann L. Drapeau, Ken W. Shirrif, John H. Hartman, Ethan L. Miller, Srinivasan Seshan, Randy H. Katz, Ken Lutz, David A. Patterson, Edward K. Lee, Peter H. Chen, and Garth A. Gibson. RAID-II: A high-bandwidth network file server. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 234–244, 1994.

[45] C. Duncan, D. Eppstein, and S. G. Kobourov. The geometric thickness of low degree graphs. In *20th ACM Symposium on Computational Geometry*, pages 340–346, 2004.

[46] C. A. Duncan, A. Efrat, S. G. Kobourov, and C. Wenk. Drawing with fat edges. In *9th Symposium on Graph Drawing (GD)*, pages 162–177, 2001.

[47] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced aspect ratio trees and their use for drawing large graphs. *Journal of Graph Algorithms and Applications*, 4:19–46, 2000.

[48] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. *Journal of Algorithms*, 38:303–333, 2001.

[49] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Planarity-preserving clustering and embedding for large planar graphs. *Computational Geometry: Theory and Applications*, 24(3):203–224, 2002.

[50] C. A. Duncan and S. G. Kobourov. Polar coordinate drawing of planar graphs with good angular resolution. *Journal of Graph Algorithms and Applications*, 7(4):311–333, 2003.

[51] Christian A. Duncan, Stephen G. Kobourov, and V. S. Anil Kumar. Optimal constrained graph exploration. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 807–814, 2001.

[52] Brad Dux, Anand Iyer, Saumya Debray, David Forrester, and Stephen G. Kobourov. Visualizing the behaviour of dynamically modifiable code. In *13th IEEE Workshop on Porgram Comprehension*, pages 337–340, 2005.

[53] A. Efrat, C. Erten, and S. G. Kobourov. Fixed-location circular-arc drawing of planar graphs. In *11th Symposium on Graph Drawing (GD)*, pages 147–158, 2003.

[54] Alon Efrat, Stephen Kobourov, and Anna Lubiw. Computing homotopic shortest paths efficiently. *Computational Geometry: Theory and Applications*. To appear in 2005. (Preliminary version in *10th Annual Symposium on Algorithms (ESA)*, p.411-423, 2002).

[55] Stephen G. Eick, Todd L. Graves, Alan F. Karr, Audris Mockus, and Paul Schuster. Visualizing software changes. *Software Engineering*, 28(4):396–412, 2002.

[56] David Eng. Combining static and dynamic data in code visualization. In *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 43–50, New York, NY, USA, 2002. ACM Press.

[57] C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. Yee. GraphAEL: Graph animations with evolving layouts. In *11th Symposium on Graph Drawing (GD)*, pages 98–110, 2003.

[58] C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. Yee. Exploring the computing literature using temporal graph visualization. In *Proceedings of Visualization and Data Analysis (VDA)*, pages 45–56, 2004.

[59] C. Erten and S. G. Kobourov. Simultaneous embedding of planar graphs with few bends. In *12th Symposium on Graph Drawing (GD)*, pages 195–206, 2004.

[60] C. Erten, S. G. Kobourov, A. Navabia, and V. Le. Simultaneous graph drawing: Layout algorithms and visualization schemes. *Journal of Graph Algorithms and Applications*. To appear in 2005. (Preliminary version in *11th Symposium on Graph Drawing (GD)*, p. 437-449, 2003).

[61] C. Erten, S. G. Kobourov, A. Navabia, and V. Le. Simultaneous graph drawing: Layout algorithms and visualization schemes. In *11th Symposium on Graph Drawing (GD)*, pages 437–449, 2003.

[62] C. Erten, S. G. Kobourov, and C. Pitta. Intersection-free morphing of planar graphs. In *11th Symposium on Graph Drawing (GD)*, pages 320–331, 2003.

[63] C. Erten, S. G. Kobourov, and C. Pitta. Morphing planar graphs. In *20th ACM Symposium on Computational Geometry (SCG)*, pages 320–331, 2004.

[64] Cesim Erten. *Simultaneous Embedding and Visualization of Graphs*. PhD thesis, University of Arizona, 2004.

[65] Cesim Erten and Stephen G. Kobourov. Simultaneous embedding of a planar graph and its dual on the grid. *Theory of Computing Systems*, 38(3):313–327, 2005.

[66] Stuart I. Feldman and Channing B. Brown. Igor: a system for program debugging via reversible execution. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 112–123, New York, NY, USA, 1988. ACM Press.

[67] Paul A. Fishwick, Jinho Lee, Minho Park, and Hyunju Shim. Next generation modeling i: Rube: a customized 2d and 3d modeling framework for simulation. In *Winter Simulation Conference*, pages 755–762, 2003.

[68] D. Forrester, S. G. Kobourov, A. Navabi, K. Wampler, and G. Yee. graphael: A system for generalized force-directed layouts. In *12th Symposium on Graph Drawing (GD)*, pages 454–466, 2004.

[69] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force directed placement. *Software - Practice and Experience*, 21:1129–1164, 1991.

[70] P. Gajer, M. T. Goodrich, and S. G. Kobourov. A fast multi-dimensional algorithm for drawing large graphs. *Computational Geometry: Theory and Applications*, 29(1):3–18, 2004.

[71] Pawel Gajer and Stephen G. Kobourov. GRIP: Graph dRawing with Intelligent Placement. *Journal of Graph Algorithms and Applications*, 6(3):203–224, 2002.

[72] E. R. Gansner, Y. Koren, and S. North. Topological fisheye views for visualizing large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 11:457–468, 2005.

[73] Paul V. Gestwicki and Bharat Jayaraman. Methodology and architecture of jive. In *SOFTVIS*, pages 95–104, 2005.

[74] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[75] John Hartman, Larry Peterson, Andy Bavier, Peter Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd Proebsting, and Oliver Spatscheck. Experiences building a communication-oriented JavaOS. *Software: Practice & Experience*, 30(10):1107–1126, 2000.

[76] John H. Hartman, Scott Baker, and Ian Murdock. Customizing the Swarm storage system using agents. *Software–Practice and Experience*. To appear.

[77] John H. Hartman, Ian Murdock, and Tammo Spalink. The Swarm scalable storage system. In *International Conference on Distributed Computing Systems*, pages 74–81, 1999.

[78] John H. Hartman and John K. Ousterhout. Performance measurements of a multiprocessor Sprite kernel. In *USENIX Summer*, pages 279–288, 1990.

[79] John H. Hartman and John K. Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, 1995.

[80] John H. Hartman, Larry Peterson, Andy Bavier, Peter Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd Proebsting, and Oliver Spatscheck. Joust: A platform for communications-oriented liquid software. *IEEE Computer*, 32(4):50–56, April 1999.

[81] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: understanding the behavior of object-priented applications. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 251–269, New York, NY, USA, 2004. ACM Press.

[82] Xin He. On floor-plan of plane graphs. *SIAM Journal of Computing*, 28(6):2150–2167, 1999.

[83] Kelly Heffner and Christian Collberg. The obfuscation executive. Technical Report TR04-03, Department of Computer Science, University of Arizona, 2004.

[84] Anand Iyer, Alon Efrat, Cesim Erten, David Forrester, and Stephen G. Kobourov. A force-directed approach to sensor localization. In *13th Symposium on Graph Drawing (GD)*. To appear in 2005.

[85] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *ICSE*, pages 360–370, 1997.

[86] I. T. Jolliffe. *Principal Component Analysis*. Springer, second edition, October 2002.

[87] James A. Jones, Alessandro Orso, and Mary Jean Harrold. Gammatella: visualizing program-execution data for deployed software. *Information Visualization*, 3(3):173–188, 2004.

[88] I. H. Kazi, D. P. Jose, B. Ben-Hamida, C. J. Hescott, C. Kwok, J. A. Konstan, D. J. Lilja, and P.-C Yew. JaViz: A client/ server Java profiling tool. *IBM Systems Journal*, 39(1):96–117, 2000.

[89] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[90] S. G. Kobourov and C. Pitta. An interactive multi-user system for simultaneous graph drawing. In *12th Symposium on Graph Drawing (GD)*, pages 492–503, 2004.

[91] S. G. Kobourov and K. Wampler. Non-Euclidean spring embedders. *IEEE Transactions on Visualization and Computer Graphics*. To appear in 2005. (Preliminary version in *10th Annual IEEE Symposium on Information Visualization (INFOVIS)*, p. 207-214, 2004).

[92] T. Kohonen. *Self-Organizing Maps*. Springer, 2000.

[93] Joseph B. Kruskal and Myron Wish. *Multidimensional Scaling*. Sage Press, 1978.

[94] C.-C. Liao, H.-I. Lu, and H.-C. Yen. Compact floor-planning via orderly spanning trees. *Journal of Algorithms*, 48:441–451, 2003.

[95] S. Lloyd. Last square quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[96] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, 1989.

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

[98] T.G. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6), June 1988.

[99] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the OSDI*, pages 153–168, October 1996.

[100] Steve Muir, Larry Peterson, Marc Fiuczynski, Justin Cappos, and John Hartman. Proper: Privileged operations in a virtualised system environment. In *Proceedings of the 2005 Usenix Technical Conference*, 2005.

[101] Ian Murdock and John H. Hartman. Swarm: A log-structured storage system for Linux. In *Proceedings of the FREENIX Track: 2000 USENIX Annual Technical Conference*, June 2000.

[102] F. Kenton Musgrave. *Methods for Realistic Landscape Imaging*. PhD thesis, Yale Univeristy, 1993.

[103] Brad 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, April 1986.

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

[105] Ginger Myles and Christian Collberg. Software watermarking through register allocation: Implementation, analysis, and attacks. In *International Conference on Information Security and Cryptology*, 2003.

[106] Robert H. B. Netzer and Mark H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 313–325, New York, NY, USA, 1994. ACM Press.

[107] M. E. J. Newman. Modularity and community structure in networks. *Proc. Natl. Acad. Sci. USA*, 103:8577–8582, 2006.

[108] A. Noack. Modularity clustering is force-directed layout. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 79, 2009.

[109] Ciaran O'Reilly, David W. Bustard, and Philip J. Morrow. The war room command console: shared visualizations for inclusive team coordination. In *SOFTVIS*, pages 57–65, 2005.

[110] Alessandro Orso, James A. Jones, and Mary Jean Harrold. Visualization of program-execution data for deployed software. In *SOFTVIS*, pages 67–76, 211, 2003.

[111] F.-G. Ottogalli, C. Labbé, V. Olive, B. de Oliveira Stein, J. Chassin de Kergommeaux, and J.-M. Vincent. Visualisation of distributed applications for performance debugging. In V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, and C.J. Kenneth Tan, editors, *ICCS'01: International Conference in Computational Science*, Lecture Notes in Computer Science 2074, pages 831–840, Berlin, Heidelberg, 2001. Springer.

[112] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The sprite network operating system. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society, ; ACM CR 8905-0314*, 21(2), 1988.

[113] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution for parallel programs. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 124–129, New York, NY, USA, 1988. ACM Press.

[114] Wim De Pauw, David H. Lorenz, John M. Vlissides, and Mark N. Wegman. Execution patterns in object-oriented visualization. In *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 219–234, 1998.

[115] Vern Paxson. A survey of support for implementing debuggers. Fall Semester 1990.

[116] Larry Peterson, Yitzchak Gottlieb, Mike Hibler, Patrick Tullmann, Jay Lepreau, Steve Schwab, Hrishikesh Dandelkar, Andrew Purtell, and John Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications*, 19(3):473–487, March 2001.

[117] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *SOFTVIS*, pages 67–75, 2005.

[118] Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A taxonomy of software visualization. In *Proc. 25th Hawaii Int. Conf. System Sciences*, 1992.

[119] Erwin Raisz. The rectangular statistical cartogram. *Geographical Review*, 24(2):292–296, 1934.

[120] Steven P. Reiss. Software visualization in the desert environment. In *PASTE*, pages 59–66, 1998.

[121] Steven P. Reiss and Manos Renieris. The bloom software visualization system. In *Software Visualization – From Theory to Practice*. MIT Press, 2003.

[122] Steven P. Reiss and Manos Renieris. Jove: java as it happens. In *SOFTVIS*, pages 115–124, 2005.

[123] Gruia-Catalin Roman and Kenneth C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, 26(12):11–24, 1993.

[124] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen Alan Herrod. Using the simos machine simulator to study complex computer systems. *ACM TRansactions on Modeling and Computer Simulation*, 7(1):78–103, 1997.

[125] S. Roweis and L. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290:2323–2326, 2000.

[126] Tapas Ranjan Sahoo and Christian Collberg. Software watermarking in the frequency domain: Implementation, analysis, and attacks. Technical Report TR04-07, Department of Computer Science, University of Arizona, 2004.

[127] Ben Shneiderman. Tree visualization with tree-maps: A 2-D space-filling approach. *ACM Transactions on Graphics*, 11(1):92–99, 1992.

[128] Tammo Spalink, John H. Hartman, and Garth Gibson. A mobile agent's effect on file service. *IEEE Concurrency*, 8(2):62–69, 2000.

[129] Oliver Spatscheck, Jørgen S. Hansen, John H. Hartman, and Larry L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Transactions on Networking*, 8(2):146–157, 2000.

[130] 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.

[131] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of java applications. In *3rd Virtual Machine Research and Technology Symposium (VM'04)*, May 2004.

[132] J. B. Tenenbaum, V. V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290:2319–2323, 2000.

[133] J. R. R. Tolkien. *The Shaping of Middle-Earth*. Houghton Mifflin Harcourt, 1986.

[134] Lucian Voinea, Alexandru Telea, and Jarke J. van Wijk. Cvsscan: visualization of code evolution. In *SOFTVIS*, pages 47–56, 2005.

[135] Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour, Laurie J. Hendren, and Clark Verbrugge. Evolve: An open extensible software visualisation framework. In *SOFTVIS*, pages 37–46, 2003.

[136] Marcos Weskamp. Newsmap. `marumushi.com/apps/newsmap`, 2004.

[137] Michael Whitney, Morris Bernstein, Renato De Mori, Kostas Kontogiannis, Brain Corrie, Hausi M&#252;ller, Scott Tilley, Ettore Merlo, John Mylopoulos, Kenny Wong, J. Howard Johnson, James McDaniel, and Martin Stanley. Using an integrated toolset for program understanding. In *CASCON '95: Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*, page 59. IBM Press, 1995.

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

[139] Peter Young and Malcolm Munro. Visualizing software in virtual reality. In *IWPC*, pages 19–27, 1998.

[140] M.V. Zelkowitz. Reverse execution. *Communications of the ACM*, 16(9), September 1973.