3-D Visualization of Software Structure

Mathew L. Staples
Millivision, L.L.C.
29 Industrial Drive East
Northampton, MA 01060 USA

James M. Bieman
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA

August 1998

Abstract

A common and frustrating problem in software engineering is the introduction of new faults as a side-effect of software maintenance. An understanding of all of the relationships that exist between modified software and the rest of a system can limit the introduction of new faults. For large systems, these relationships can be numerous and subtle. The relationships can be especially complex in object-oriented systems that include inheritance and dynamic binding. Software visualization can potentially ease both impact analysis and general program understanding. Software visualization can facilitate program understanding by graphically displaying important software features. However, despite recent success in developing useful and intuitive graphical representations for certain aspects of software, current software visualization systems are limited by their lack of *scalability* — the ability to visualize both small and large-scale software entities. This paper demonstrates that three-dimensional (3-D) graphics and a hierarchy of overlapping views can increase the scalability of software visualization. The hierarchy provides detailed information without sacrificing the "big picture". Overlapping is used to provide context between high and low-level views. A prototype system, *Change Impact Viewer (CIV)*, tests these visualization mechanisms. CIV highlights areas of a system that can potentially be affected by a change to a selected function. The mechanisms, as implemented in CIV, show improvements in scalability over those provided by other systems, without decreasing usefulness or intuitiveness.

To Appear in *Advances in Computers*, Volume 49, Edited by M. Zelkowitz, Academic Press, London, 1999.

Contents

1 Introduction

Throughout a software product's life cycle, many different people are responsible for understanding the design details of the software code. New engineers face the difficult task of "coming up to speed" on the code before they can do any useful work. Learning the structure of new code is especially costly during the software maintenance phase, since maintenance programmers are often not involved in the original design and implementation. Design documents can help maintenance programmers understand the code, but after several maintenance cycles documents are often out of date and incomplete.

Software visualization (SV) is one tool to aid program understanding. In the broadest sense, SV is simply the creation of a graphical model of a software system (or some aspect of a software system). While SV can be done manually by drawing diagrams such as flow-charts, the term "software visualization" usually implies automatic analysis of software followed by automatic creation of a graphical model. With any type of SV, however, the common goal is to facilitate understanding of some aspect of the software.

Several SV systems have been implemented that provide useful and intuitive visualizations of certain aspects of software. A major problem that is common to these systems is a lack of support for *scalability.* Scalability, with respect to SV, is the ability to visualize both large and small-scale software systems. The usefulness of most current SV systems tends to decrease as the size of the visualized software increases. This paper presents a method for increasing the scalability of SV through the use of 3-D graphics and a hierarchy of overlapping views.

One of our hypotheses is that scalability of software visualization systems can be increased by applying the techniques that are used in cartography to display detailed information about large geographical areas. By using a hierarchy of views, a world atlas can contain views ranging in scale from the whole world down to individual cities and towns. Furthermore, each level of the hierarchy provides a different level of detail; the amount of detail shown in each view is inversely proportional to the scope of the view. Similarly, a software visualization tool that contains a hierarchy of views with multiple levels of resolution should be more scalable than a system that uses only one level of resolution. Also, as in cartography, overlap between views can be used to provide context between different views of the system.

Another hypothesis is that the use of three-dimensional (3-D) graphics to display these hierarchical views will further increase scalability. By displaying views in 3-D, a simulated volume is created that provides more virtual space than is available in a 2-D display. Therefore, a 3-D view should appear less cluttered than a 2-D view that contains the same amount of information. Likewise, a 3-D view should be able to show more information than a 2-D view without increasing the amount of clutter.

In order to test these hypotheses, we developed a prototype software visualization system that can graphically display useful software maintenance information. In particular, the system is designed to aid in visualizing *change impact. Impact analysis* (IA) (also referred to as *change impact analysis*) is the process of determining the potential effects of a software code change [17]. The prototype system, the Change Impact Viewer (CIV), can graphically display a range of impact analysis results using overlapping hierarchies of 3-D views.

Section 2 provides some background on software visualization. Criteria are given for evaluation of SV systems and several existing systems are discussed with respect to these criteria. One of these criteria, scalability, is shown to be a common problem in existing systems. Section 3 describes principles of software impact analysis which are used to design the prototype CIV tool. Section 4 discusses in detail the problem of scalability in SV systems and describes the solution we used to design CIV. Section 5 describes the features of CIV and explains how they solve the problems described in Section 4. Section 6 comparing CIV with other existing systems. Finally, Section 7 provides some conclusions and gives direction for further research.

2 Software Visualization

Software visualization (SV) is defined by Price, et al., [16] as "the use of the crafts of typography, graphics design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software". This definition includes both *algorithm visualization* and *program visualization*. In an *algorithm visualization*, a process is shown graphically, usually as an animation. This type of SV requires *dynamic* information such as can be attained from a run-time debugging program. For the purposes of this paper, SV is taken to mean *program visualization*, or *static* software visualization, unless otherwise stated. This type of SV requires only static information, such as can be attained with the front end of a compiler, and generally results in static views of the software system.

Additionally, we define the following terms with respect to SV: "*software system*" refers to the software to be visualized; "*programmer*" refers to the writer(s) of the software that is to be visualized; "*SV system*" refers to the software that performs the visualization; "*user*" refers to the person creating the visualization through the SV system[1]; and "*information overload*" describes the situation in which a visualization shows more information in one view than a user can easily understand.

2.1 Software Understanding

SV should not be confused with the study of *program understanding*, although they are closely related. In SV, visualization is used as a tool to aid in program understanding. The study of program understanding, therefore, provides important background for designing useful visualization tools.

Program understanding or comprehension involves the process by which a programmer learns about a software system at the code level. For large systems, this can be a difficult task that requires a large amount of time. Jerding and Stasko argue that the original developers have a mental picture or conceptual model of the software that they will transform into software code [10]. When other programmers read this code, they formulate their own conceptual model. However, some subtleties and semantic information will generally be overlooked. The lack of detailed understanding of the software can make it difficult to make changes to the software without changing the underlying design. Often, after many cycles of maintenance, the original design of a software system becomes badly obscured. Making changes that do not fit naturally into the original design can make further changes more difficult and will eventually make the software unmaintainable [22].

Jerding and Stasko also point out that the problem of program understanding is particularly difficult in object-oriented software. The object-oriented paradigm, they say, aids in programming and design, but also has the side effect of making the resulting systems difficult to understand. Consider, for example, the task of tracing a function call. Tracing a call in a C program requires searching for the called function. Programs written in C++, however, may apply dynamic binding. The specific method that gets executed is determined at run time, and could be any of a hierarchy

[1]In [16] this person is called the *visualizer*, and the *user* is defined as the person using the resulting visualization to understand a software system. However, in a fully automated system, such as CIV, the visualizer and the user are one in the same.

of virtual functions. Dynamic binding may have been useful in the design, since it allows objects of different types to be treated in the same way, but it makes the code reader's job more difficult. The reader must try to determine the specific abstraction that the virtual function represents in a variety of situations.

A major goal of program understanding research is to determine the learning process that programmers use, so that appropriate support tools can be developed. Several models of how a programmer actually goes about this learning process are discussed by von Mayrhauser and Vans [23]. Some believe that the program understanding process is best described by a *top-down* model [19]. Programmers first gain a high level understanding, and then gradually learn the details. This can be done by making hypotheses about the program, and then testing those hypotheses by further inspection of the code. Others believe that program understanding is best described with a *bottom-up* model [15]. Programmers build a program model by studying the control flow. *"Beacons"* are used to group low-level parts into higher-level parts. Beacons are "cues that index knowledge" [23], for example, a reference such as "`list->next()`" is a beacon for a linked-list. Programmers also build a situation model which maps parts of the code to the real-world problem domain. Code fragments can be summarized with natural language to help understand the code at a higher level. For example, the code "`tmp = *a; *a = *b; *b = tmp;`" can be described as "swap a and b".

Von Mayrhauser and Vans [23] developed a hybrid model for program comprehension. They determined experimentally that programmers tend to use multiple models, and switch between them frequently. Programmers may start out with a top-down, high-level model, formulate hypotheses about that model, and then switch to a bottom-up approach to test that hypothesis about a particular part of the system.

The use of multiple models of program understanding implies that a useful visualization tool for program understanding would present both high-level abstractions, and low-level detail of the software system. Additionally, the SV system should support quick context switches between high and low-level views of parts of the system. Optional static analysis queries would also be helpful for testing hypotheses about the software.

2.2 Software Visualization Criteria

Computer visualization has been applied in a wide range of fields. Scientists use computer visualization to help understand everything from the structure of a DNA molecule to the terrain of Mars. Only recently, however, has computer visualization been used to visualize software systems. What are some of the reasons for this delay?

A large part of the difficulty in providing intuitive visualizations is the lack of physical structure in software. The question immediately arises, "What does software *look* like?" According to Brooks [11], software has no visualizable structure: "Despite progress in restricting and simplifying software structures, they remain inherently unvisualizable, and thus do not permit the mind to use some of its most powerful conceptual tools". Clearly, active researchers in SV take a less pessimistic view, but creating visualizations that are both useful and intuitive is nevertheless a difficult problem.

Another problem common to many SV systems is *information overload.* A view may seem useful and intuitive for small, simple examples, but may quickly become too cluttered to understand for larger examples.

We identify key criteria for evaluating SV systems based on the problems causing slow adoption of existing systems:

Usefulness: Does the visualization actually make the programmer's job easier? A useful visualization should provide information that is not readily available by direct inspection of the source code.

Intuitiveness: Is the visualization easy to understand? The visualization should match the programmer's intuition about what the software "looks" like.

Scalability: Does the visualization work well for large, real-world systems? The visualization should not become less useful or intuitive as the size of the visualized system increases.

We also look at the scope of SV systems to identify which aspects of the software an SV attempts to visualize. Often, an SV system will consist of several views which each show specific aspects of the software. Another concern is whether or not empirical studies support the SV system's usefulness, intuitiveness, and scalability.

2.3 Existing Visualization Tools

Despite the pessimistic outlook of Brooks, there has been some reported success in visualizing various aspects of software systems. This section describes some of the more well known systems, and evaluates them based on the criteria given in Section 2.2.

HP SoftBench Static Analyzer

Hewlett Packard's HP SoftBench [7] is a multi-user system designed to facilitate program development by incorporating the major software development tasks (program editing, configuration management, etc.) into a single environment. HP SoftBench Static Analyzer [8], which provides static 2-D views of various software relationships, is an integrated part of HP SoftBench. Some of the visualization features of the static analyzer include automatic creation of call-graphs, class-inheritance graphs and file-inclusion graphs. An example of a small call-graph is shown in Figure 1.

The SV features of SoftBench are very intuitive. Graphs such as the call-graph are familiar to most software engineers. The class-inheritance graph is useful since it provides high-level information that is not immediately obvious from direct inspection of the code. The call-graph is also useful in that it enables the user to trace function calls more easily than by searching through the code by hand. This is especially true when the function calls span multiple files.

The most powerful feature of SoftBench is its query engine, rather than its visualization abilities. It is primarily a query engine. Queries can be performed on specific identifiers, for example, "Show me all of the uses of variable y". Results of these queries can then be shown in textual form and in a separate query graph. This type of analysis can speed up the process of testing hypotheses about the software. SoftBench's SV features would be more useful if they were better integrated into this query engine. For example, it would be useful if the user could select objects in a graph, perform queries based on the selected objects, and then see the results graphically.

SoftBench's SV capabilities are both useful and intuitive for small systems, but they fail to work well for large systems. While the underlying query engine is scalable, the visualization features become less useful as the size of the visualized system increases. This is especially true for the call-graph. For example, Figure 1 shows only 46 functions. Several other functions are off the screen, but can be seen by scrolling the view. For large numbers of functions, the view only shows a small portion of the entire graph at one time. In many cases, it is impossible to see a calling

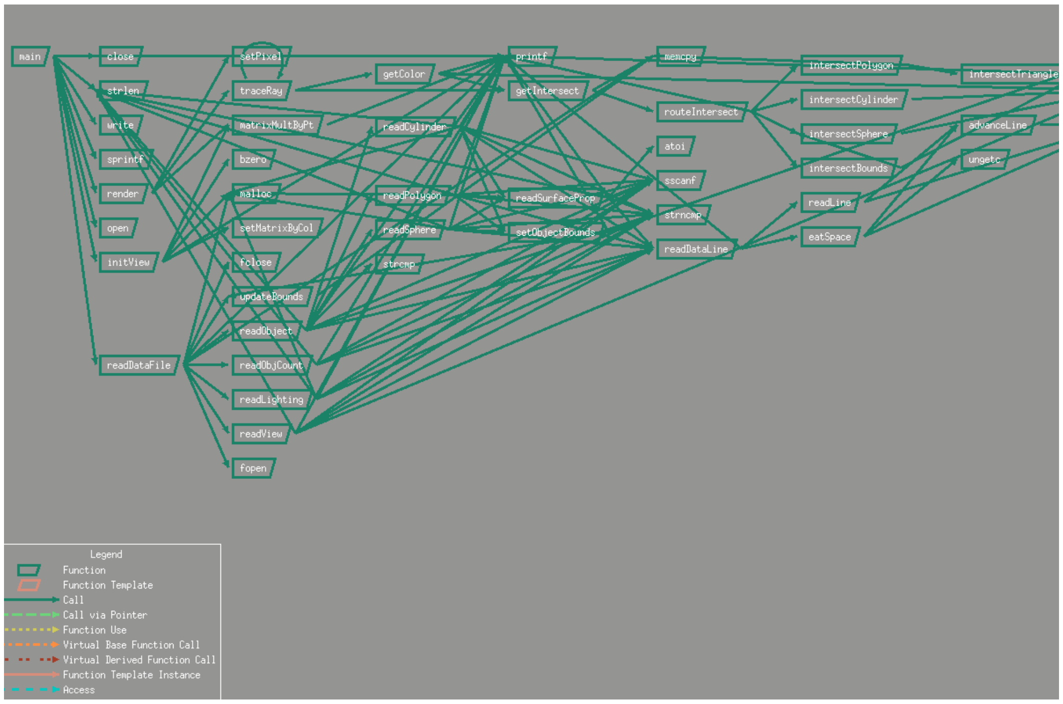


Figure 1: A call-graph generated by HP SoftBench Static Analyzer

function and the corresponding called function simultaneously. The number of functions can be reduced by selecting a sub-set of the total number of functions in the program, but the selection process can be tedious. The other SoftBench graphs are less prone to information overload, since they show only classes or files, and there are fewer classes and files than functions.

Imagix 4D

Imagix 4D, developed by the Imagix Corporation, contains a sophisticated 3-D code browser. This system is built with *Tcl/Tk* [14], which is a useful tool for developing X-Windows applications. Tcl is a script language that, together with the Tk toolkit, enables interesting graphical applications to be developed at a higher level than with C or C++. This makes it a good tool for rapid prototyping of SV systems.

Figure 2 shows an initial Imagix 4D view for a small application. This graph shows the function, main, at the top of the window. Below main are the functions that main calls (laid out in a grid). Also shown (in the same grid as the called functions) are the data items accessed by the function. In this way, a call-graph and a def/use graph are incorporated into a single view.

Selecting a function shows the local call-def/use-graph of the function, and the parent (calling function) of the called function. Selecting a data item shows the data item at the bottom, with all of the functions that access it in a grid above it. A text viewer provides a hyper-text code view with point-and-click navigation of the source code (similar to popular web-browser programs such as *Netscape*). Objects that can be shown graphically are underlined. Selecting these objects changes the main graphical view to focus on the new object, in addition to making a hyper-text "jump" to the referenced location in the source code.

A user can select between files, classes, or calls for a given view. For example, choosing classes

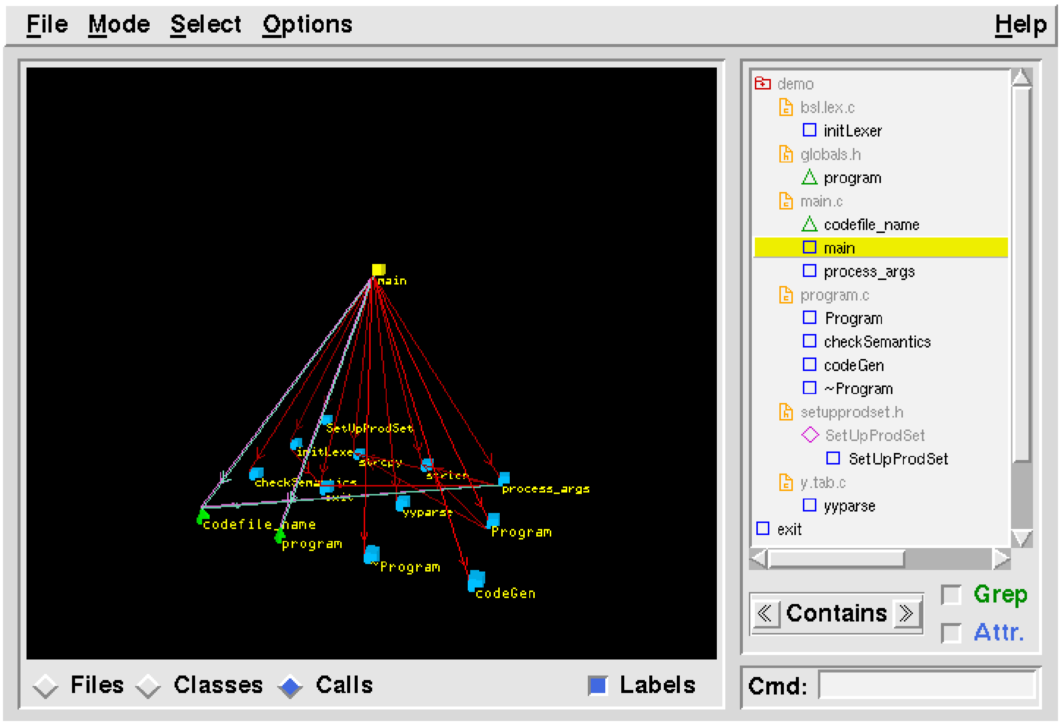


Figure 2: Imagix 4D view of function main.

switches the main view to an inter-class relational view. Like the HP SoftBench Static Analyzer, Imagix 4D has some static analysis query features. After selecting a class, a set of queries about that class can be made, such as, "show all super-classes of this class", or "show all sub-classes of this class". The queryeatures are better integrated into the visualization system than those of HP's Static Analyzer: objects for queries are selected through the display, rather than by typing the name of an identifier in a separate window.

Imagix 4D is a useful tool for code understanding, since it allows quick access between structures and their relations. However, it lacks a consistent view hierarchy. Many different views of a particular object are possible depending on how the object was accessed. For example, if two different functions, A and B, each make a call to a third function, C, and the graph of C is selected through the graph of A, then the parent in the graph of C will be A. Accessed another way, the parent of C will be B. Thus, a user can easily get "lost" while navigating through the system. Imagix 4D is therefore most appropriate for a bottom-up approach to program understanding.

Imagix 4D attempts to solve the scalability problem by decomposing the visualized system into small, easily visualizable parts. Each view consists of detailed information about a small part of the system, such as an individual function. In this way, Imagix 4D is capable of visualizing very large systems. Unfortunately, the "big picture" information is lost. Scalability has been increased, but at the cost of usefulness. It is unclear that the visualization of a single function provides more information than the accompanying text display of that function. The user may choose to ignore the graphical display and use only the text-based code browser.

Nevertheless, Imagix 4D provides an intuitive graphical means of browsing source code. The tight coupling between the graphical display and the hyper-text code browser increases the intuitiveness of the visualization.

POLKA and Goofy

Several prototype dynamic visualization systems have been developed using POLKA [21]. POLKA, or Polka-3D, developed at Georgia Institute of Technology, is an object-oriented system which allows programmers to develop 2 and 3-D animations without needing to be well-versed in 3-D graphics techniques. It provides classes for, "1) an entire animation 2) individual views or windows onto the animation 3) the entities which help define a view such as graphical objects and actions or motions." As such, it is a general-purpose tool that can be used to build many types of visualizations.

Goofy, designed at the University of Exeter, is a customization of POLKA which provides an environment for animating object-oriented and procedural features of C++ [6]. Goofy consists of a language with which an animation designer can define objects, define their movements, change their attributes, and choreograph events in multiple windows.

Goofy is capable of displaying many types of animations for the same algorithm. For example, Ford conducted an experiment in which programming students used Goofy to create animations of their designs [6]. The students developed a wide variety of representations for the same constructs and algorithms. The results of this experiment were then used to formulate a model for the process of learning object-oriented programming.

Although Goofy and POLKA are very flexible, they do not provide facilities for static or dynamic analysis of source code. Thus they are not full-fledge SV systems, but rather tools that can be used to build SV systems. The user must either create the animations by hand, or develop an automated system on top of them. For example, another system, *cppinfo* [5], also developed at the University of Exeter, uses Goofy to automatically create static and dynamic views of C or C++ programs. Source code analysis was performed using *newyacc*.

Other Systems

Twelve software visualization systems are described in [16]. The following provides a brief description of six selected systems:

1. *BALSA* shows multiple simultaneous animations of each running program. This was the first program to show multiple algorithms racing on a single display. A later Macintosh version (BALSA-II) included rudimentary sounds associated with algorithm events.

2. *Zeus* is the latest version of BALSA. Its primary feature is that it can animate parallel-programs. Also, the most recent extension uses 3-D views to encode additional information.

3. *ANIM*, from AT&T Bell Labs, has a simple application programming language for creating animations. The user must insert statements into "interesting points" in the code.

4. The *Incense* prototype SV system, created by Myers at Xerox PARC in 1980, was the first SV system to automatically produce graphical displays of program data structures. In 1988 Myers developed a related production system called *Amethyst* at Carnegie-Mellon. to display both static and dynamic representations of data structures for Pascal programs. Amethyst was later integrated with MacGnome (a Pascal programming environment for Macintosh computers), which is now known as Pascal Genie. Pascal Genie has been used to teach data structures and algorithms.

5. *The University of Washington Program Illustrator (UWPI)* automatically provides visualizations for high-level *abstract* data structures (as opposed to concrete data structures which

appear in the implemented code). It contains an “inferencer”, which suggests a number of possible abstractions for each data type found in the code. Each possibility is assigned a weight based on closeness of fit between the operations performed on that data type (in the code) and the operations allowed on that abstract data type. The abstraction with the highest fitness value is then chosen to be displayed. A separate“layout strategist” displays each abstraction in a manner appropriate to that abstract data type. The rule-base used by the inferencer is a good example of the potential use of artificial intelligence in software visualization.

6. *LogoMedia* allows programmers to associate non-speech audio with program events while the code is being developed. The system uses a Macintosh linked to a MIDI synthesizer. Sound commands can turn on MIDI instruments, play back samples or adjust the sound’s pitch and volume. Empirical studies suggest that this audio feedback may actually be intuitive and useful in debugging. Use of audio feedback is described as *auralization*; creating a mental picture through audio rather than visual cues.

Many SV prototypes have been developed, but few have made their way to industrial use [16], perhaps because of a lack of empirical studies demonstrating real benefits from these systems.

Dynamic visualizations such as Zeus and ANIM, can help users to understand algorithms, however, dynamic visualizations tend to be less automated and less interactive than static visualizations. They can be effective in producing presentations or animated documentation of software, but they require the creator of the animation to understand the software. This lack of automation makes many of the current SV systems unsuitable as stand-alone program understanding tools.

Static visualizations lack run-time information. Thus, data structures such as linked-lists are difficult to recognize statically. Some reasoning about run-time behavior is necessary for the programmer to get a full understanding of the software.

These visualization systems exhibit one or two common problems. The visualized information may be obvious from looking directly at the code, so that programmers will not bother using the tool. Otherwise, useful information is shown, but the presentation is not easy to interpret. For example, function views in Imagix 4D show the called functions and the referenced data. For a single function, the code view contains the same information in addition to control-flow information adding context to the calls and references. There is no evidence that the 3-D view provides a better visualization than the hyper-text code browser. Conversely, HP SoftBench can show a call-graph for the entire program, but the graph is so large that it is difficult to interpret. Because of scalability problems, most existing static SV systems work well for small, toy examples, but fail to work well for larger scale examples [16, 10, 5].

All of the above systems have attempted to visualize software abstractly. The views created by these systems look nothing like the source code that they represent. Ball and Eick [2] have taken a different approach: they have developed techniques for visualizing the code itself. Their approach begins with a “pretty printed” version of the source code. Color and indentation are used to highlight various source code features such as function definitions and loop constructs. Then, by scaling this view, the text becomes smaller and smaller until the text itself is no longer readable. Although the text cannot be read in this view, the highlighted properties can still be seen. A separate browser window can then be passed over the compressed code view in order to read the code (this works much like a micro-fiche reader).

This type of visualization seems to be both intuitive and very scalable. They have been able to visualize systems with over a million lines of code. It is also useful in that metrics, such as the age (time since last modification) of each line of code, can be easily visualized. However, abstract relationships between high-level entities, such as classes, cannot be easily represented in this way. Abstract graphical representations of software entities have better potential to represent design-level information. Also there is no obvious reason why these two, fundamentally different, approaches to SV could not be used together to provide complementary views of large software systems.

2.4 The Need for Scalability

Software visualization has the potential to reduce software engineering costs by easing program understanding. In order to realize this potential, SV systems must be designed that are not only *useful* and *intuitive*, but also *scalable.* The usefulness and intuitiveness of an SV system should not decrease as the size of the visualized software system increases.

To date, many SV systems have been built, but most have not been widely used in industry. A major reason for this is their lack of scalability. Techniques for visualizing the code itself, such as those developed by Ball and Eick, have shown the ability to visualize large systems, but they also lack the ability to visualize relationships between high-level software entities. Abstract, relational views of software are desirable, but current systems either fail to work for large systems, or decompose large systems into small pieces and thereby lose the "big picture" information.

3 Impact Analysis

Our prototype CIV system uses SV techniques to visualize the results of impact analysis.

Impact Analysis (IA) is defined by Arnold and Bohner [1] as "the activity of identifying what to modify to accomplish a change, or of identifying the potential consequences of a change". Thus, one component of IA is the determination of the syntactic changes necessary to effect a semantic change. The second component of IA is the determination of the semantic effects resulting from a syntactic change. Since semantic information is difficult to express automatically, neither component of IA appears to be easily automatable. However, the second component can be approximated by determining the area of code in which the semantic effect *could* appear. That is, if an automated system can identify the parts of the system which are definitely *not* affected, then the programmer's job will have been significantly reduced. For the purposes of this paper, we assume that a desired syntactic change has been specified before any analysis takes place. A related, but separate issue is the problem of actually specifying the change in an interactive system.

IA makes use of compiler theory. Many systems use techniques such as dependency analysis and program slicing to obtain information about change impact. Therefore, before discussing any specific IA techniques, the following common definitions from compiler theory, are presented.

Definition 3.1 *A* digraph $G \stackrel{\text{def}}{=} \langle N_G, E_G \rangle$ *where* N_G *is a nonempty set of nodes and* E_G *is a (possibly empty) set of directed edges such that* $E_G \subseteq \{N_G \times N_G\}$*. For all* $(u, v) \in E_G$ u *is the* source *and* v *is the* target*.*

Definition 3.2 *A* walk *in a graph* G *is a finite, non-null sequence of nodes* $W \stackrel{\text{def}}{=} n_1, n_2, \ldots, n_k$ *such that* $n_i \in N_G$ *for each* i *in* $1 \ldots k$*, and* $(n_i, n_{i+1}) \in E_G$ *for each* i *in* $1 \ldots k-1$*.*

Definition 3.3 *A* flowgraph $G \stackrel{\text{def}}{=} \langle N_G, E_G, n_0 \rangle$ *where* $\langle N_G, E_G \rangle$ *is a digraph and* $n_0 \in N_G$ *such that there exists a walk from* n_0 *to all other nodes in* N_G*.* n_0 *is often called the* initial node*. If* m *and* n *are two nodes in* N_G*, then* m dominates n *if* m *is on every walk from* n_0 *to* n*.*

Definition 3.4 *A* hammock graph $G \stackrel{\text{def}}{=} \langle N_G, E_G, n_0, n_e \rangle$ *where both* $\langle N_G, E_G, n_0 \rangle$ *and* $\langle N_G, E_G^{-1}, n_e \rangle$ *are both flowgraphs.* $E_G^{-1} \stackrel{\text{def}}{=} \{(a,b) \mid (a,b) \in E_G\}$. *If* m *and* n *are two nodes in* N_G*, then* m inverse dominates n *if* m *is on every walk from* n *to* n_e.

Definition 3.5 *A* control-flow graph *(CFG) for a procedure in a program is a hammock graph in which* n_0 *represents the procedure's entry point,* n_e *represents the procedure's exit point, and every node* $m \in N_G$ *inverse dominates* n_0*. Nodes in* N_G *represent program statements. Some procedures have multiple exit points, however, by creating an additional node, all such procedures can be represented by a CFG with a single entry and a single exit node.*

Definition 3.6 *A* call-graph *for a program is a hammock graph in which each node in* N_G *represents a procedure,* n_0 *is the entry procedure (main), and* n_e *is the exit procedure (_exit). Edges in* E_G *represent procedure calls.*

Definition 3.7 *A* def/use graph $DUG \stackrel{\text{def}}{=} \langle G, \Sigma, D, U \rangle$*, where* G *is a* CFG*,* Σ *is a finite set of identifiers which name the variables referenced in the nodes of* G*, and* D *and* U *are functions* $N_G \rightarrow \mathcal{P}(\Sigma)$ *where* $D(n)$ *returns the set of identifiers* modified *in node* n *and* $U(n)$ *returns the set of identifiers* used *in node* n*.* $\mathcal{P}$ *represents the power set.*

3.1 Tool Support for Impact Analysis

Given a change to a particular function, one possible method of IA is as follows. The programmer begins by determining the local effects of the change. The programmer will then try to determine if the change could affect other areas of the code. In general, the programmer will search through the code for any indications of a possible undesired side effect of the change before committing to the change. The programmer inspects functions that call the given function or use the same global or class-member data for potential impact. The inspection also includes functions that are called by the affected function and might execute a different control-flow path because of the change. Any functions found to be affected by the change must, in turn, initiate another phase of analysis. In the worst case, the effect can "ripple" throughout the entire software system.

Manual IA is a time-consuming process. Thus, tool support can be very helpful. The simplest type of IA tool support is a keyword search using a text editor, or system utilities such as *grep*. After finding all initially affected objects, the programmer may need to expand the search one key-word at a time. For example, in order to investigate a def/use chain from a modified variable, the programmer must search for the first defined variable in the chain, then look at at all uses of that variable. This process must be repeated until the programmer is convinced that there are no unwanted side-effects. Not only is this process time-consuming, but it can also be error-prone. Errors can be introduced as side effects of a modification. More sophisticated IA tools are needed to narrow the programmer's search, and increase the accuracy of the resulting analysis.

Clearly, programmers will benefit from tools that automatically determine the impact of a particular change on a software system. Unfortunately, such a tool requires program comprehension, which is a semantic problem similar to natural language comprehension. Complete IA involves answering fundamental semantic questions about a software system. It is difficult to determine if a change to a function changes its *defined* behavior in any way, but it is impossible to determine if the defined behavior is the *desired* behavior without semantic information. For example, a bug fix in a function will change the behavior of that function. If a calling function relied on the previous bug, then it would be negatively impacted by correcting the bug. If, however, the changed function's *desired* behavior has not been changed, then the calling function may not be negatively affected. Without natural language interpreters that can read and understand design documents, many important IA questions can only be answered by a human.

While no automated system will be able to fully solve the IA problem, there are subproblems which can be automated to greatly ease the maintenance programmer's task. The results of automated IA systems are therefore approximate. Generally, the result of analysis is simply a subset of the system in which impact could *potentially* be felt. Conversely, the job of an automated IA program is to "prune" the parts of the system which are definitely not affected by a given change.

3.2 Program Slicing

A program *slice* is the portion of a program that might affect the value of a particular identifier at a specified location in a program [24]. The location and identifier of interest define a *slicing criterion*. *Program slicing* is the process of generating a reduced program that only includes the source code that affects or is affected by the slicing criterion.

A tool that can determine the parts of a system that can potentially be affected by a change will help to reduce the complexity of the impact analysis problem. *Program Slicing* can determine a subset of a program's text that implements a specified subset of the program's behavior. For example, the subset of a program's behavior that sets a particular variable's value at a particular point in the control-flow graph will generate a program slice consisting of only the code necessary to produce that behavior. Any parts of the program that the given variable does not depend on can be deleted. The union of all slices of a program is the original program. Given a specific change, a program slice can be constructed based on the changed lines of code. The manual part of the impact analysis can then be restricted to this slice.

Program slicing is based on dependence analysis. Dependence analysis is also a primary tool for impact analysis.

3.3 Dependence Analysis

Most techniques for impact analysis use at least some aspects of dependence analysis, such as data-flow, or control-flow analysis. For example, a change to a given function is likely to affect those functions which are dependent on the given function. What does it mean to say that one function is dependent on another? Loyall and Mathisen [12] suggest that a procedure A is dependent on another procedure B if and only if one or more statements in A are dependent on one or more statements in B.

There are several types of statement-level dependencies, and hence there are several types of procedure-level dependency. The three main types are *data dependence*, *control dependence* and *syntactic dependence*. A statement A is *data dependent* on a statement B if the value of any variable in A is affected by the value of any variable in B. More precisely, A is data dependent on B if, given a def/use graph $DUG = \langle G, \Sigma, D, U \rangle$, there exists a walk from node B to node A in G and at least one identifier is in both $D(B)$ and $U(A)$.

A statement A is *control dependent* on another statement B if the execution of A is dependent on B. There are two main types of control dependence. *Strong control dependence* occurs when the execution of B can determine if A gets executed at all. For example, the statements in the else branch of an if-then-else statement are strongly control dependent on the conditional if statement. *Weak control dependence* occurs when A is strongly control dependent on B and/or the execution of statement B could indefinitely suspend the execution of A. For example, statements which occur after the body of a while loop are weakly control dependent on the conditional of the loop.

A statement A is *syntactically dependent* on another statement B if there is a chain of control and data dependence from B to A. For example, if statement A is control dependent on statement B, and B is data dependent on statement C, then A is syntactically dependent on C.

The above definitions assume that the corresponding graphs contain both statements A and B. In the standard definition of a CFG, however, a single procedure is assumed. An inter-procedural CFG is needed to determine statement-level control dependency between two procedures. Similarly, an inter-procedural def/use graph is needed to determine data dependence between two statements in different procedures. Loyall and Mathisen introduce the concepts of inter-procedural control-flow and inter-procedural def/use graphs [12].

An inter-procedural CFG is a set of procedural CFG's with added arcs for procedure calls and returns. It is produced by combining procedural CFG's via the call-graph for the program. Each procedure has two special nodes in the graph: $proc_i$ and $proc_f$. The call is made to $proc_i$, and the return arc is from $proc_f$. An important aspect of this is that the *calling context* is saved in order to ensure that the calls return to where they were called from. For example, if two different procedures make a call to a third procedure, the call and return arcs are ordered such that the first return arc followed corresponds to the last call-arc followed. This correspondence, which mimics the call frames that would be placed on the stack at run-time, makes it possible to include recursive procedures in the inter-procedural CFG.

An inter-procedural def/use graph is essentially the same as a procedural def/use graph, except that the procedural CFG which it contains is replaced by an inter-procedural CFG. Unlike the dependence analysis done in [24], variable names and variable instances are distinguished by calling context. This distinction is necessary in order to avoid conflicts between variables that have the same name, but different scope.

3.4 Approximation in Impact Analysis

The impact of a change can be either *automatic* or *potential* [17]. An automatic impact is an impact that will unconditionally be caused by a given change. Example changes with automatic impact are changing a procedure's interface, modifying code which changes a procedure's return value(s), etc. A potential impact is an impact that could *possibly* be caused by a given change. For example, a decision point in the control flow can make it impossible to determine statically if a particular branch is actually affected by a given change.

Since impact analysis attempts to identify the potential, as well as the automatic consequences of a change, *conservative*, but *safe* approximations are often made ([13], [24], [12], [1]). Dependence analysis techniques based on compiler theory lead to the view that an IA system should err on the side of completeness. That is, an IA system should not exclude any components of the given software system that can potentially be affected by the given change.

Unfortunately, conservative assumptions can lead to an over-estimation of the affected area. The usefulness of an IA system depends on its ability to prune the unaffected areas of the code. One way to do this is to allow the user to manually prune the impact as the analysis proceeds. Automated pruning of IA results remains a difficult problem.

Jackson and Ladd [9] hypothesize that human users are more forgiving than compilers and can make use of incomplete results. In their words, "...we have taken the radical step of trying to maximize the accuracy of our tool's output by compromising its soundness." Their tool, *Semantic Diff*, compares two versions of a program and produces an impact report. By trying to capture the semantics of a change rather than making conservative assumptions about the syntax of the change, the system is able to recognize meaning-preserving changes, for example, renaming a local variable. However, in order to avoid inter-procedural analysis, which they claim is overly time-consuming, the system does make worst-case assumption with respect to procedure calls.

3.5 The “Ripple” Effect

A common difficulty in making software changes is the *“ripple” effect* [1, 17, 4]. This can occur in several ways, but the basic idea is that a change in one procedure can require a change in another procedure which in turn requires changes in other procedures. For example, adding a parameter to a function will necessarily require a change to all calling functions. If those functions don’t have the information required to pass this extra parameter, they will need to add a parameter to their interfaces. This type of change can cause a “ripple” of changes throughout the system.

By changing a function’s interface, the ripple is initially caused by a syntactic change to the function’s interface. This only causes a required change to the immediate callers of the changed procedure. Determining if those procedures now need to change their interface is a semantic question. This means that dependency analysis alone will not be able to determine the extent of the ripple effect. If conservative assumptions are made, then the result of impact analysis will show the entire call-graph prior to the changed function.

Original Code:

```
void swap(void *a, void *b)
{
  void *tmp;

  tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Modified Code:

```
ErrCode swap(void *a, void *b)
{
  void *tmp;

  if (a == NULL || b == NULL)
    return InvalidParam;

  tmp = *a;
  *a = *b;
  *b = tmp;

  return NoError;
}
```

Figure 3: A modification causing a potential ripple effect.

Consider the modification shown in Figure 3. Error checking is added to the swap function to make it more robust. This change could affect all calling processes, since swap now returns a value that was not previously returned. Since it is now possible for swap to fail, it is also possible for the callers of swap to fail. The callers of swap may now need to add return values to their interfaces. However, this error condition may never occur since the calling functions always pass in proper values, and it might be decided to ignore the returned value. This is a case in which the potential impact of the change could ripple throughout the entire system, but might not have any real affect that a programmer would be interested in.

Now we show how the impact of a change can be effectively visualized through scalability improvements.

4 Scalability of Software Visualization

Scalability in software visualization can be improved by using multiple levels of resolution and 3-D graphics. Section 4.1 explains the problem of information overload and how it relates to scalability. Section 4.2 discusses user interface problems that can limit scalability. Section 4.3 introduces the concept of a multi-resolution visualization. This concept forms the basis of the hierarchical viewing scheme used by CIV. Finally, Section 4.4 discusses how 3-D viewing can be used to reduce clutter and increase scalability.

4.1 Information Overload

Virtually all SV systems suffer from *information overload*. A view becoming too complicated or cluttered to be easily understood is generally the result of poor *scalability*.

As programs get larger, more data needs to be displayed on computer screens that provide a fixed and relatively small amount of space. Two classic solutions are (1) scaling the objects to fit more of them on the screen, and (2) allowing the view to take up more virtual area than the screen. Both are partial solutions. Scaling causes objects to become too small to be easily read. Using the second solution, the user must scroll a small view port over a larger virtual view. As the virtual view becomes large, the user cannot observe the larger picture. An extreme example is a digitally scanned painting observed using a scrolling window that shows only one pixel at a time (scaled large enough to see). A user could not recognize the painting simply by scrolling this window around the image.

A call-graph is a more realistic example. A call-graph of the entire system can be shown for a system with only a few functions. However, except for very small applications, the number of functions is prohibitively large to display a complete call-graph of an entire application. Compressing a large graph into a single view results in a cluttered, difficult to read diagram. Alternatively, the graph can be spread out over a large virtual space, which can be viewed by scrolling. Then call arcs may extend over large parts of the graph, making it difficult for users to determine where the arcs begin and end. Although such visualization schemes may work well on a small scale, they cease to be useful on a large scale. On a small scale, the single view could provide a “big picture” of an entire system. Unfortunately, as the number of functions increases, the view shows less and less of the complete system.

Scaling and scrolling are not the only ways to deal with large amounts of information. Consider scale in a road atlas. A traveler planning a cross-country trip begins with a map of the entire country. A road map of the entire country does not show every road, but rather it shows only the major highways. More detail is provided by individual maps of the states, and even more by looking at county maps, city maps, etc. Similarly, a call-graph that shows all of the functions in a large system is analogous to an online road map of the entire country showing every neighborhood street. The information overload in such a view renders the view essentially useless. We propose a solution analogous to that used in maps. A hierarchy of views is created to allow detail to be shown without sacrificing the global, “big picture” information.

4.2 User Interface Issues

To display a subset of a system as a graph, the user must determine the portion of the graph to include. For a call-graph that displays each function as a node, there are two ways to determine this set interactively: the user selects each function individually, or begins with all functions and deletes any functions that are not desired in the graph. Both approaches are supported by HP

SoftBench. Either approach can be tedious. Consider again the example of the driver planning a cross country trip. If the driver has only county maps, he/she must first list all of the county maps needed for the journey. The alternative is for the driver to list all of the county maps that would *not* be needed. Both solutions are tedious.

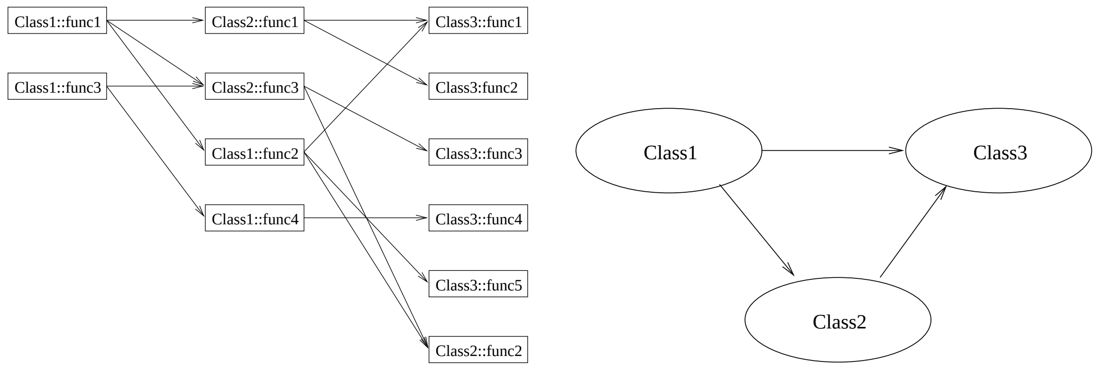
The process can be improved with an SV system that allows the user to select a single function and then the system generates a call-graph containing all functions which call or are called by the selected reference function (this is the approach used by Imagix 4D). Still, the user may want to know more than one level of call information at a time. This method of visualizing one function at a time provides an isolated view which does not explain how the reference function fits into the system as a whole.

Another major problem is that a user who is not familiar with the software may not know what areas of the software are interesting with respect to the problem at hand. Just as a driver cannot be expected to know every county on a route from New York to Los Angeles before determining a higher level route (e.g. a list of states and inter-state highways), a programmer cannot be expected to list all of the functions of interest before understanding the system at a high level.

4.3 Multi-resolution Visualization

A hierarchical viewing scheme should help solve the information overload problem. The detail in views can be limited by packaging the system into logical pieces. A hierarchy of views of the system should minimize the user's effort in creating the view and increase the user's understanding of the system as a whole.

How can a call-graph be broken into logical pieces? Since software is generally designed at a higher level than that of individual functions, perhaps it should be visualized at a higher level also. Consider software written in C++. The object-oriented nature of the code provides a natural hierarchy which should be reflected in the visualization. Figure 4 shows such a simple hierarchical viewing scheme for a call-graph. Part (a) shows a call-graph containing all functions in a small system. Part (b) then shows the same call-graph containing only class-level calling information.



a. Call-graph showing all functions.

b. Higher-level call-graph showing class relationship.

Figure 4: Using modularity to simplify a call-graph.

For a set of hierarchical views to be useful, each view must make intuitive sense. Does a view using classes as nodes (as in Figure 4(b)) make more intuitive sense than a view using functions as nodes (as in Figure 4(a))? The basic idea behind object-oriented design is to view classes as independent entities which communicate through message passing. In practice, message passing is implemented via function calls. A call arc from a class, A, to another class, B, would literally mean that some method in A makes a function call to some method in B. The higher-level object-oriented view is that class A sends a message to class B. So, the class-level view (Figure 4(b)) better matches the client/server object-oriented model than the call-graph that displayed each function as a node (Figure 4(a)).

Using the nodes to represent classes rather than functions makes the user's job of selecting an initial set of nodes easier. In the previous example the user had to select a set of functions. With a class-level call-graph, the user still needs to select a set of nodes (which now represent classes), but the number of choices has decreased significantly.

While the class model aids the understanding the system at a high level, the user still needs to understand implementation details in order to perform actual maintenance on the system. The class-level call-graph does not show the function-level detail. The user needs a mechanism to move from a high-level initial view to a lower-level view without displaying all of the functions in the system.

Consider again the road atlas analogy. A map of the entire United States will not show city-level detail, but rather, separate views of the details of the cities are available. Because it is often difficult to determine how a separate, detailed view fits into a large scale view, some maps show in-place intermediate blow-up views. These views help determine how the roads entering and leaving the city match up with separate detailed views. Intermediate views for software call-graphs can be created by expanding classes of interest to show their internal functions (see Figure 5(a)). Function-level detail can be shown by creating a separate view of the internal methods of a class, or as is shown in Figure 5(a), by expanding the class in place. The first type of class-expansion is useful for showing the intra-class calling relationships, while the second is useful for showing inter-class calling relationships at the individual function level. By allowing multiple levels of resolution to be shown simultaneously, detail can be ignored throughout large parts of the system, and can be included where detail is desired.

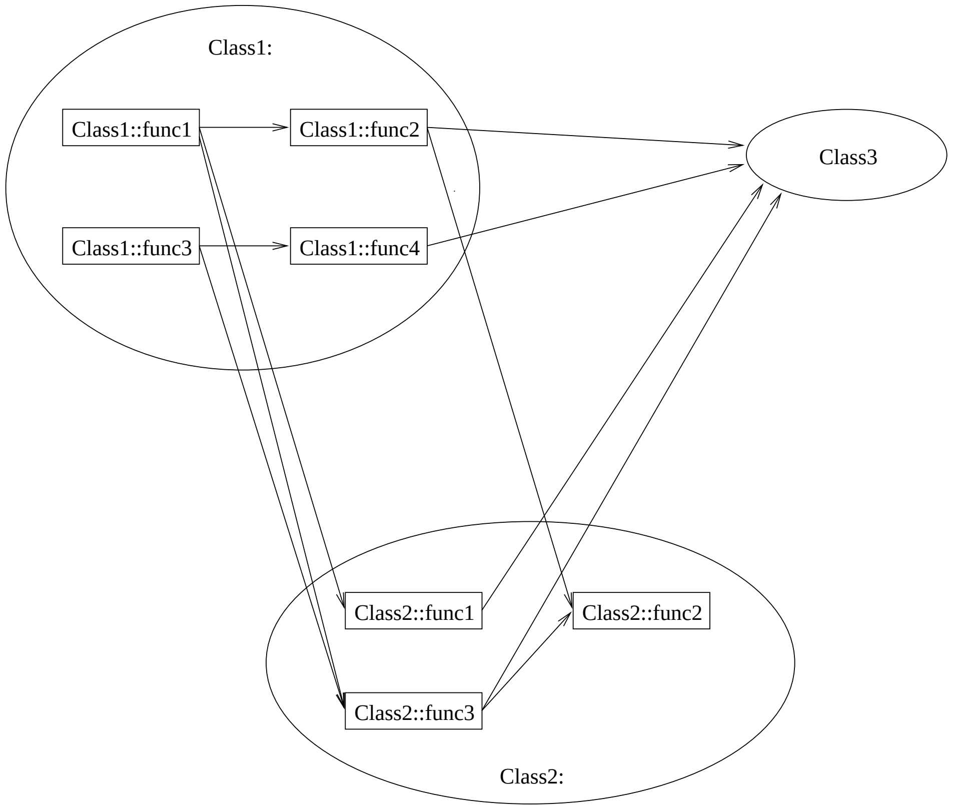
4.4 3-D Viewing

Expanding classes in place can cause information overload when several classes are expanded in the same view, The viewing area ("real estate") in a single two dimensional display is limited. Any given pair of classes may contain enough methods to fill up the entire view.

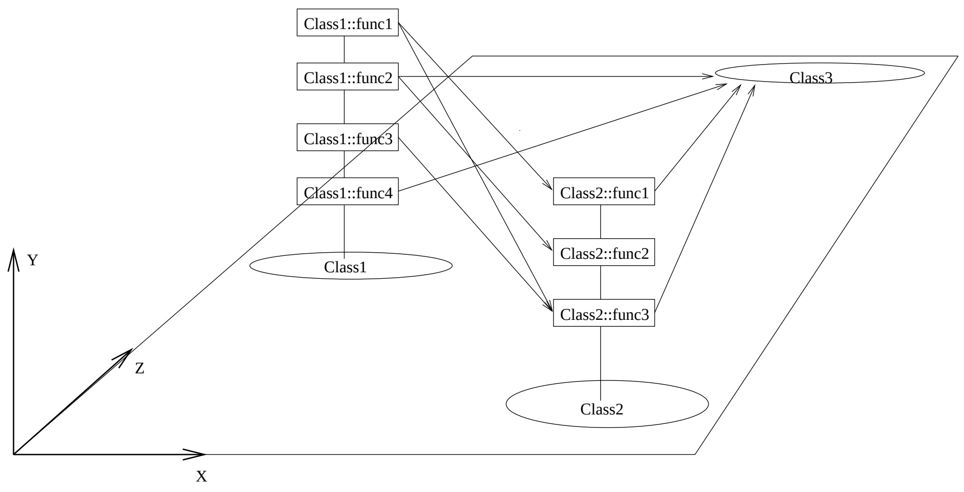
Three-dimensional graphics techniques help solve this problem. In the physical world, when real estate is limited, high-rise buildings are built. Similarly, the third dimension can be used to show internal details of classes, as shown in Figure 5(b), without using up additional space in the 2-D plane. Furthermore, the original, high-level view's layout does not have to be redone to account for each detailed view which be shown.

Clearly, since the 3-D view must be shown on a 2-D screen, the amount of clutter in the view has not really decreased. However, a 3-D view can be rotated interactively, so that the user can see the same view from many different angles. Since the view is contained in a simulated volume, rather than a flat plane, there is *conceptually* more room in which to show the same amount of information.

The expanded classes in Figure 5(b) no longer show the intra-class call relations; they show the inter-class call relations. Information has been lost in the translation from part (a) to part (b) of Figure 5, but, a separate intra-class view can be provided. Furthermore, as long as intra-class information is available, it should not be displayed in the high-level inter-class view. The low-level intra-class information would distract from the displayed high-level relationships and further clutter the image. Similarly, in a road atlas, multiple blow-ups of major cities are usually



a. In-plane class expansion.



b. Class expansion along a third axis.

Figure 5: High level call-graph with two classes expanded to show internal detail.

available. A blow-up which shows the major routes into and out of the city may be shown in the corner of the state map, while a highly detailed map of the inner city would probably be on a separate page.

Multiple levels of resolution and 3-D graphics are implemented in our prototype CIV system.

5 Change Impact Viewer (CIV)

The prototype CIV implements solutions to the scalability problems described in Section 4. We provide an overview here; more detailed information can be found in [20]. CIV builds hierarchies of 3-D views in which objects (classes, files, functions, or variables) are related through calls and data-flow. These relationships are shown as arcs between objects on the screen. The user can make change impact queries, such as, "What is the impact of changing the implementation of some specific function?" The system then shows the resulting impact by highlighting the arcs and nodes that might be affected by the change.

CIV is an X-Windows/Motif application for analyzing C++ and C programs; it will run on any HP700 workstation. Graphics rendering for CIV is performed by SPHIGS (Simple Programmer's Hierarchical Graphics Standard [18]).

5.1 Static Analysis

Both the impact analysis and the general visualization features of CIV make use of a static analysis sub-system, using a static analysis class library provided by Hewlett Packard. Static analysis is performed on a static database file. This file is produced with the SoftBench C++ compiler when given the proper command-line flag. Queries, such as finding all calls to a given function, are performed as read-only transactions on the database through the static analysis classes.

5.2 Visualization Features

CIV consists of two primary views which show inter-class and inter-file relations, respectively. From either of these views, a sub-view can be shown which shows the internal details of a specific class or file. Each of the four views can show a call-graph and/or a def/use-graph. The inter-class view also contains a graph of the inheritance relationships between classes. Color is used to distinguish between graphs — arcs in the call-graph are *green* and arcs in the def/use graph are *orange*.

All views are displayed in 3-D, and can be viewed from arbitrary angles by moving the view point and orientation via the mouse. In both primary views, classes or files are laid out on the X-Z plane. Classes or files can then be expanded along the Y-axis. (This looks like the example shown in Figure 5(b).) In this way, different parts of a project can be displayed with different levels of resolution.

The basic layout of the primary 3-D views was inspired by previous work by Litau Wu [25]. A prototype system that he developed for a class project used 3-D graphics in a similar way. The class hierarchy was laid out on the X-Y plane, and member functions were shown along the Z-axis. This was used to visualize function reuse through inheritance. A major difference between his project and CIV, is the use of user interaction to dynamically expand or contract parts of a class along the third axis.

Call and def/use graphs in CIV are updated whenever the resolution is changed. For example, a call arc exists between two classes if some method in one class calls some method in the other class. One or both of these classes can then be expanded to determine which methods are actually involved in the call.

The next two sections describe each of the primary views and their sub-views in more detail. They also provide several screen "snap-shots" of various views. Each view is a visualization of the same test system (an early prototype version of CIV).

Class Views

Inter-Class Relational View The inter-class view displayed in Figure 6 shows various relationships which exist between classes. Inheritance is shown as *blue* arcs (when displayed in color) between class nodes. The source of an inheritance arc is the derived class and the destination is the base class (this corresponds to the "is-a" relation, which exists between a sub-class and its base-class(s)).

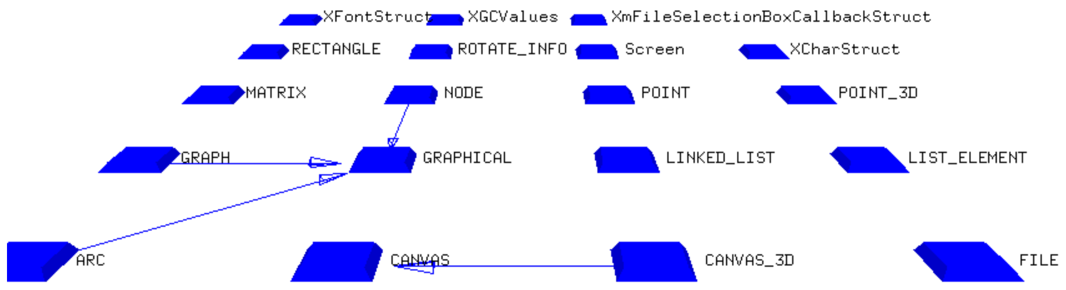


Figure 6: Inter-class relation default view. Blue arcs (when displayed in color) represent class inheritance (the "is-a" relationship). The source class is derived from the destination class.

A call or def/use arc entering or leaving a class indicates that the source or destination of that arc is a member of the class. For example, a call-arc between two classes indicates that some method in one class calls some method in the other class. Examples of a call and a def/use graph are shown in Figures 7 and Figure 8. Both the call and def/use graphs can also be shown simultaneously, but such a simultaneous display can create a somewhat cluttered view.

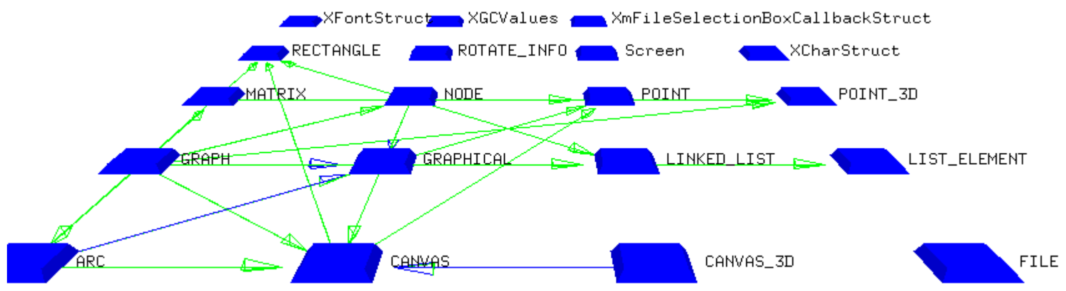


Figure 7: Inter-class view with call-graph shown.

As mentioned above, the internal parts of a class can be expanded along the Y-axis. Any or all of the following parts of a class can be shown: public data; public methods; protected data; protected methods; private data; and private methods. The given order of these options reflects the precedence in which they are displayed. Those listed first will be shown on top of the ones listed later (i.e. higher on the Y-axis). This order is maintained so that if, for example, the public data of a class is shown first, and then later the protected data is shown, then the public data would be moved up, and the protected data would be inserted underneath. The order is preserved in order to maintain a consistent interface.

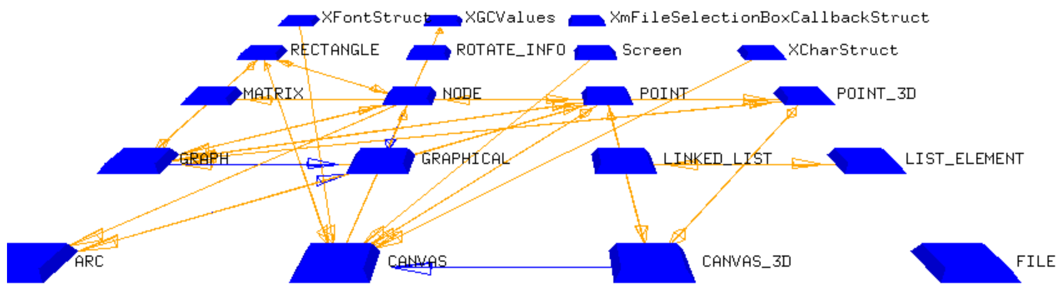


Figure 8: Inter-class view with def/use graph shown.

Figures 9, 10 and 11 show how various parts of classes can be expanded to determine how classes interact at the function level. Call-arcs in these examples are updated as classes are expanded. For example, Figure 7 shows a call-arc from the `GRAPH` class node to the `GRAPHICAL` class node. This means that one or more methods in `GRAPH` call one or more methods in `GRAPHICAL`. After expanding the public methods of `GRAPHICAL`, as shown in Figure 9, it becomes clear that `GRAPH` actually makes calls to four different constructors of the `GRAPHICAL` class. If any call-arc still terminates at the class node after the public methods have been expanded, then there are class methods (protected or private) that are still not shown.

Intra-Class Relational View The intra-class relational views displayed in Figures 12 and 13 show internal relationships between the methods and data of a class. Method and data nodes are laid out in circles parallel to the X-Z axis. Each circle represents a specific part of the class, such as private data or public methods. The circles are separated along the Y-axis such that public methods are on top, followed by public data, protected methods, protected data, private methods and finally private data on the bottom.

File Views

Inter-File Relational View The inter-file relational view, displayed in Figure 14, shows various relationships between files. The file view is similar to the class view; it groups functions together to form higher-level objects. While the class view more closely matches the hierarchy intended in the original object-oriented design, the file view provides additional information about class are implementation. By using *both* views, users get a better picture of how design and implementation interact.

The file view allows the system to work for both C++ and standard C source code. Information about the source language is contained in the static database file, which enables CIV to distinguish between C and C++ source code. Therefore, CIV can provide the same file views for C source code as for C++ source code, and then simply disable class views for any systems not implemented in C++.

Files can be expanded in the same manner as in the inter-class view. Global data, exported functions, and local functions can be displayed. Global data consists of any variables declared in a file's scope. Exported functions are those functions whose prototypes are listed in a header file, or are otherwise referenced in other files. "Exported" is used here to mean that the function *could* be called from another file, but its use is not guaranteed. Public member functions are an example of functions which are exported from the file where they are defined. Local functions are

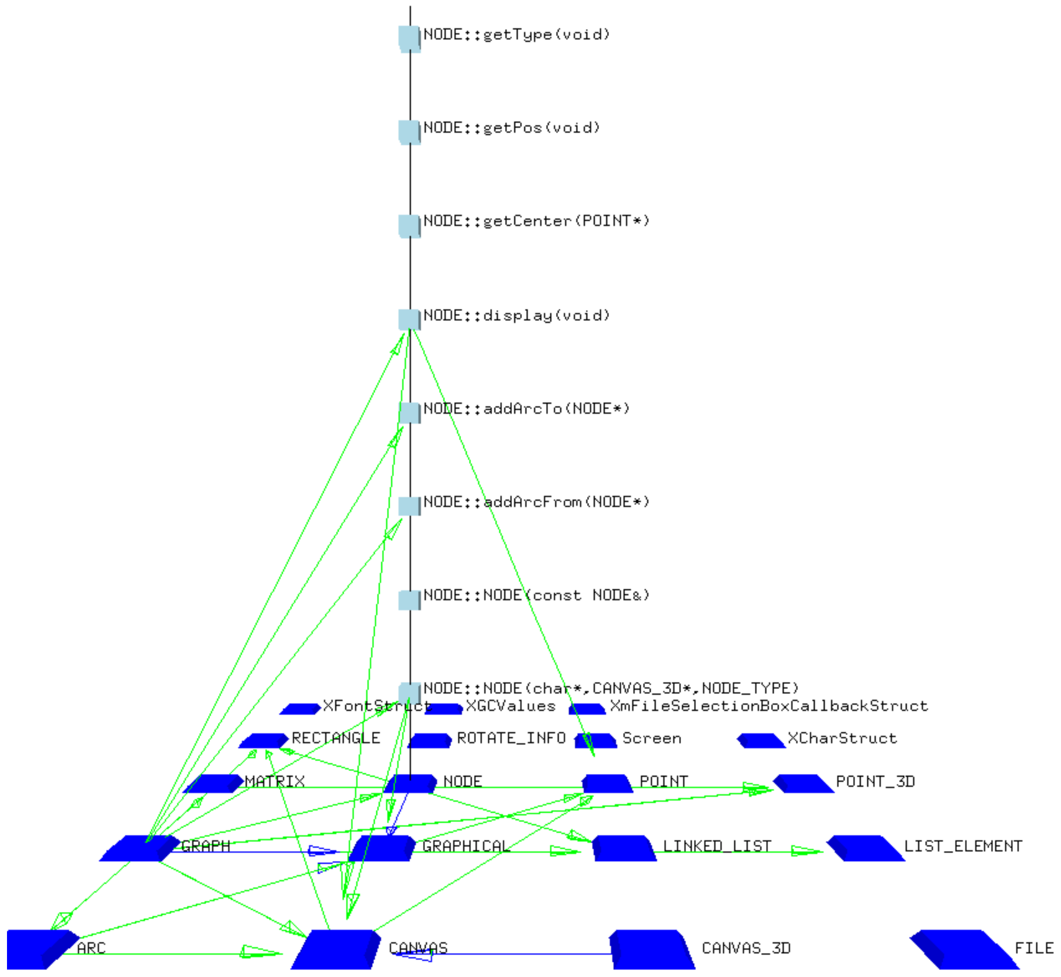


Figure 9: Inter-class view with call-graph and one class expanded.

functions that are not made public to other files. These functions cannot be referenced outside of the file in which they are defined. As in the inter-class view, the spatial ordering of expanded parts is constant regardless of the order of expansion. Global data is always on the top, followed by exported functions and local functions.

Call and def/use graphs can be shown as in the inter-class view. These graphs are also sensitive to expansion of files in the same way that the call and def/use-graphs are sensitive to expansion of classes in the inter-class view.

Intra-File Relational View The intra-file view shows relationships between the internal functions and global data defined in a specific file. Similar to the intra-class view, function and data nodes are laid out in two circles parallel to the X-Z axis. The top-most circle contains the functions, and the bottom-most circle contains the global data. All other features of the intra-class view also apply to the intra-file view.

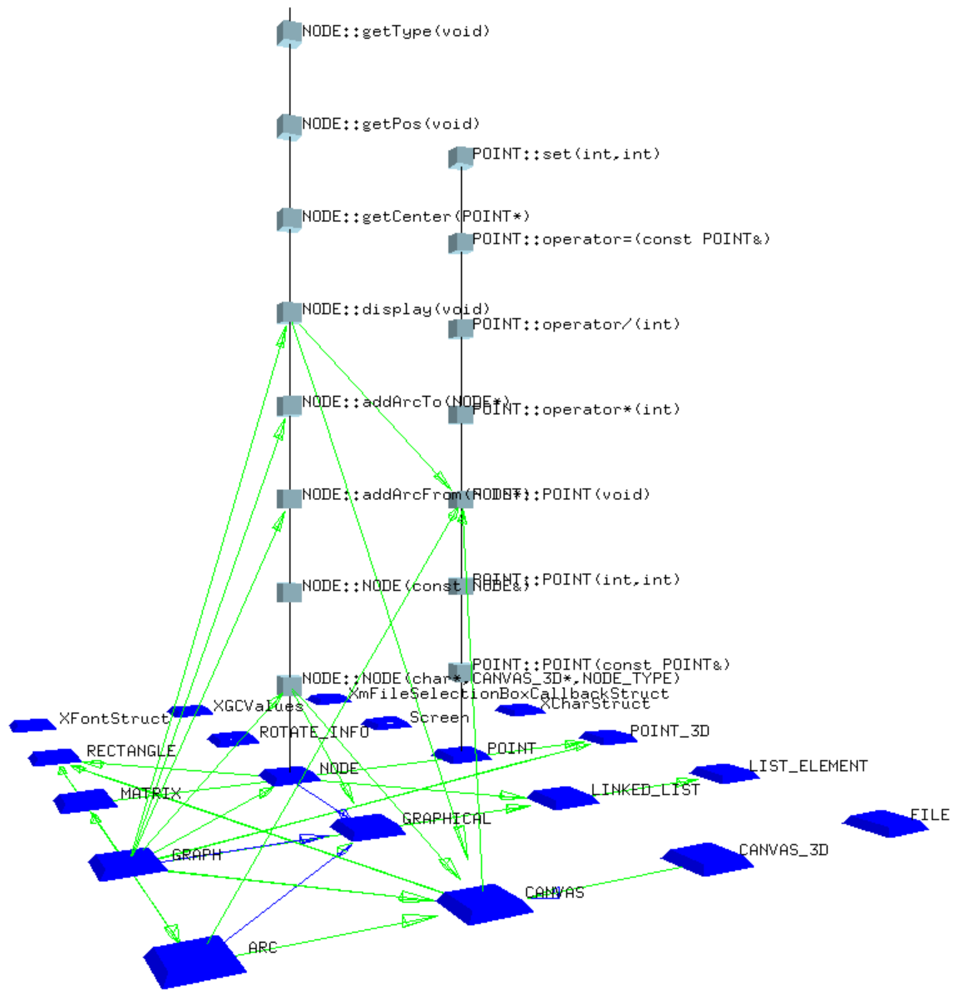


Figure 10: Inter-class view with call-graph shown and two classes expanded.

5.3 Impact Analysis Features

CIV can show the results of impact analysis queries. Queries are made by selecting a function, then selecting a change type from a list of possible high-level changes to that function. Change options include adding a parameter to the function's interface, deleting a parameter, change a parameter, modify the function's return value. A very conservative approach is now used to calculate change impact: the impact is always the transitive closure produced by following the call arcs backwards towards main. Limitations in impact analysis calculations result primarily from the limitations of the static analysis database.

Although the impact analysis calculations are now very simplistic, and err on the conservative side, the visualization features of CIV make it *capable* of showing detailed IA information. Affected portions of the view (the current view in which the IA query was made) are highlighted to show impact. In addition, arcs which contribute to the impact are highlighted, so that the user not only knows *what* has been impacted, but also has information concerning *how* that impact occurred. For example, in a more sophisticated IA system, the impact might be felt through a def/use relation. In this case, the def/use arcs involved in the impact would be highlighted. Also, several

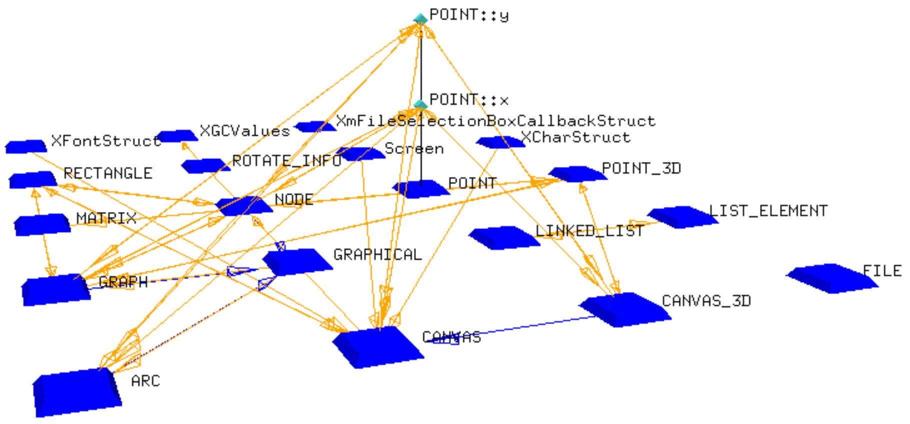


Figure 11: Inter-class view with def/use-graph shown and one class expanded to show its public data.

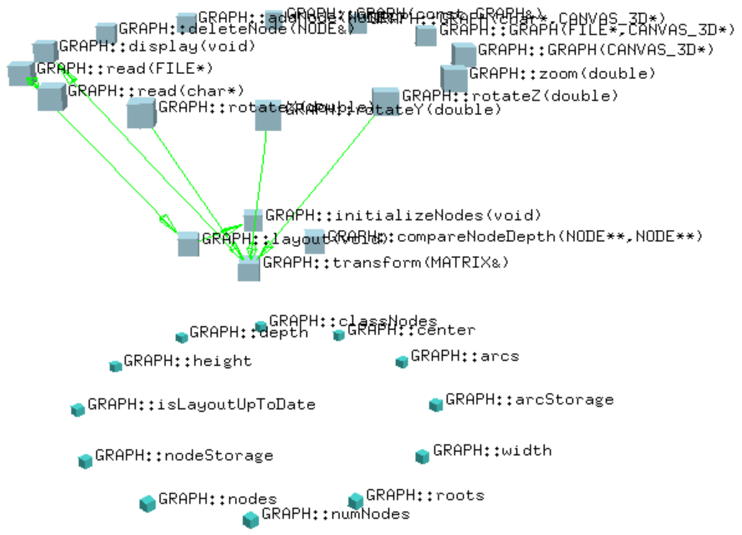


Figure 12: Intra-class relation with call-graph shown for class GRAPH. This class contains only public methods, private methods, and private data.

types of impact could be shown by using multiple highlight colors or line styles. Currently, highlighting of objects (nodes) is performed by changing their color to red. Highlighting of arcs is accomplished by changing the thickness of the lines. In this way, the color of the arc — which is used to specify what type of relation the arc represents — is not changed.

Figure 15 shows the change impact for a single class method. By expanding the affected classes in this view, the impact will be updated to show how each class was impacted internally as displayed in Figure 16. Still more detail about the impact can be seen by performing the IA analysis in the internal class view as is shown in Figure 17.

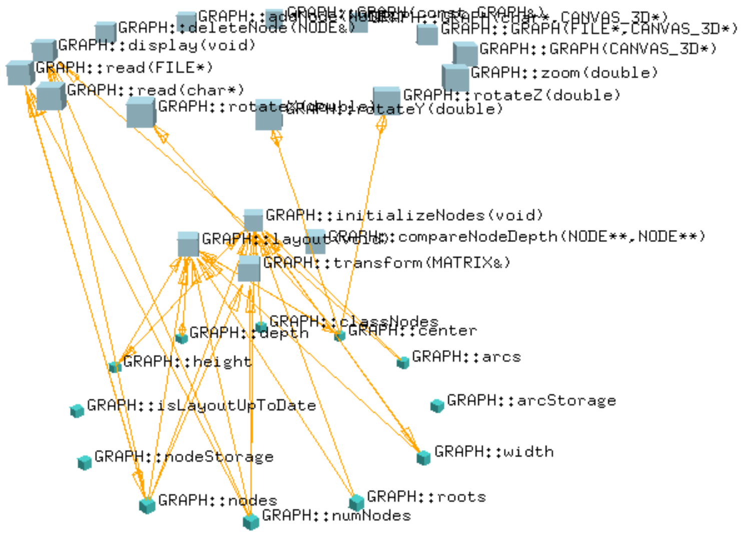


Figure 13: Intra-class relation with def/use-graph shown.

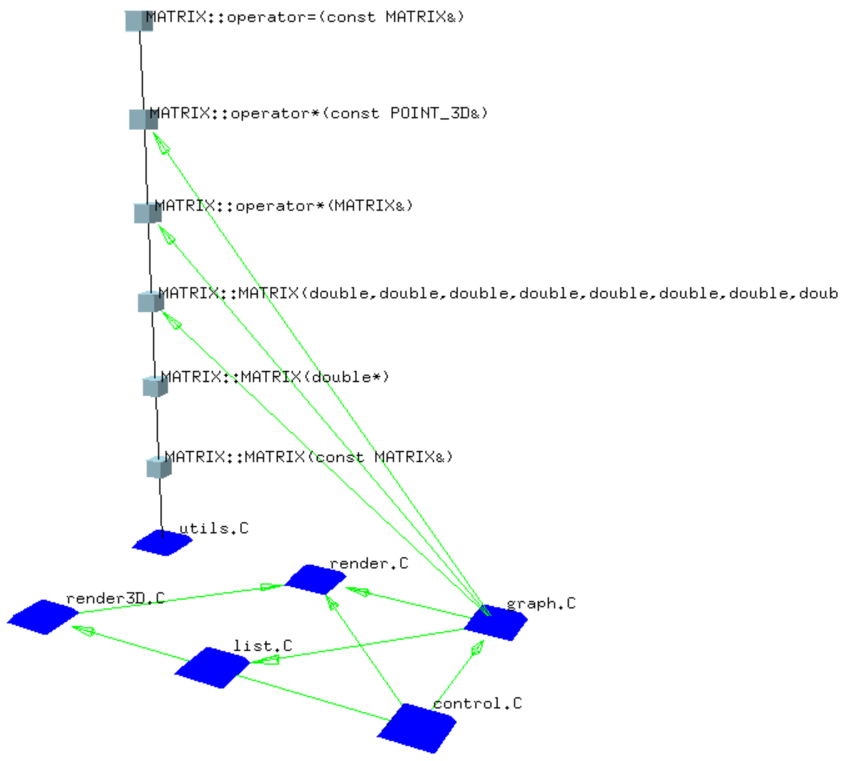

MATRIX::operator=(const MATRIX&)
MATRIX::operator*(const POINT_3D&)
MATRIX::operator*(MATRIX&)
MATRIX::MATRIX(double,double,double,double,double,double,double,doub
MATRIX::MATRIX(double*)
MATRIX::MATRIX(const MATRIX&)
utils.C
render.C
render3D.C
graph.C
list.C
control.C

Figure 14: Inter-file view with call-graph shown and one file expanded.

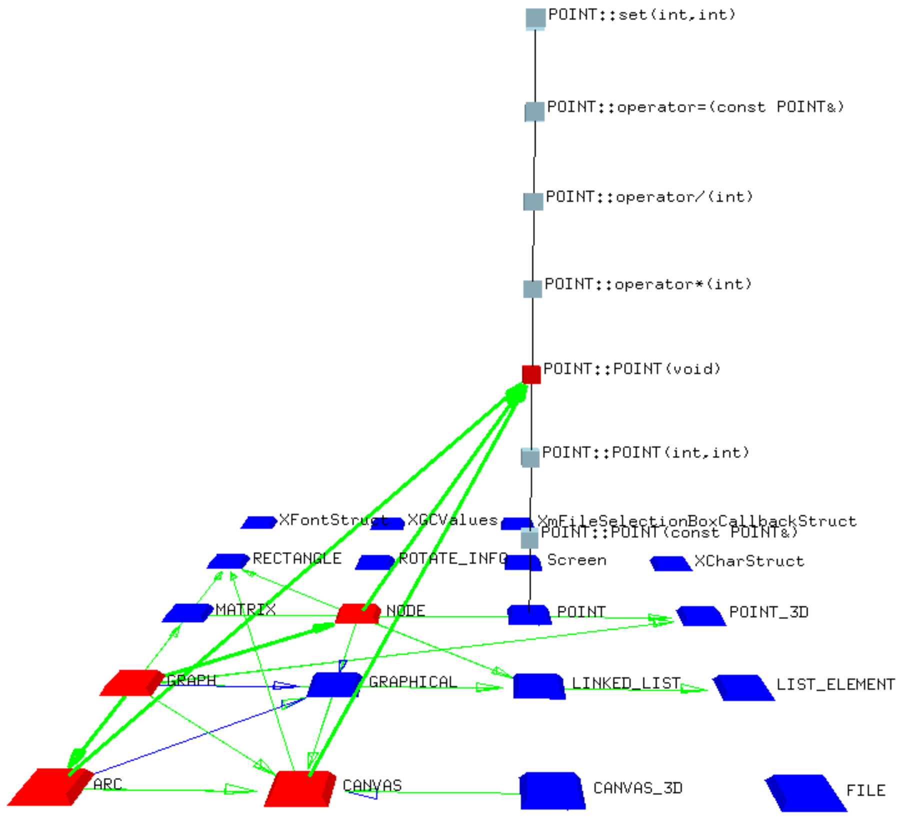


Figure 15: Inter-class relation with call-graph and impact shown for a change to the function $POINT :: POINT()$.

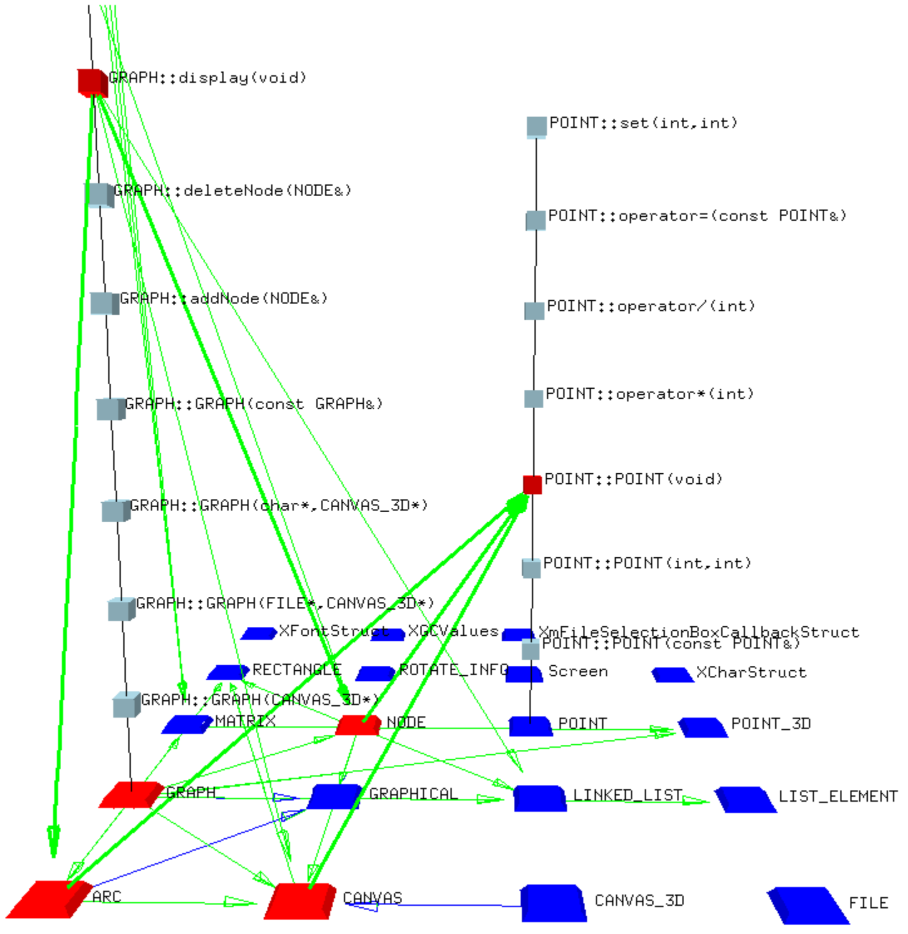


Figure 16: Inter-class relation with call-graph and impact shown for a change to the function $POINT :: POINT()$. An affected class is expanded to show how it is affected internally.

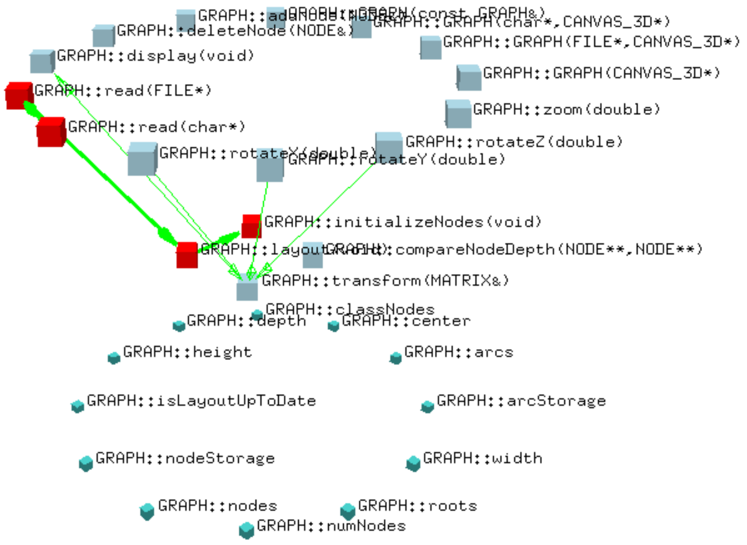


Figure 17: Intra-class relation with call-graph and impact shown for a change to the function $GRAPH :: initializeNodes()$.

6 Evaluation of Results

We evaluate our solution to the scaling problem as embodied in the prototype CIV system against a set of criteria, and we compare its performance with that of other existing systems.

6.1 Overall Evaluation

CIV can currently display classes, files, functions, and data as abstract objects. Four types of views are used to show inter-class, intra-class, inter-file, and intra-file relationships. In each view, call and def/use graphs are used to show how objects interact through data and control-flow. The inter-class view also contains a class-inheritance graph which shows the "is-a" relationships between classes. Results of impact analysis are shown by highlighting affected areas of the various graphs. Affected nodes are highlighted in addition to any arcs which contributed to the impact.

The three evaluation criteria defined in Section 2.2 are usefulness, intuitiveness, and scalability.

Usefulness

CIV is useful as a high-level code browser. By starting with the inter-class view, a user can quickly view the entire class hierarchy of the system. Then, by adding the call-graph, the user can view the entire call-graph at a high-level. At this high level, the user can quickly determine which classes interact directly and which do not. By expanding class nodes, the user can then see how the classes actually interact at the method level.

This same type of top-down analysis is not be possible via direct inspection of the source code. The user would have to search header files for class specifications, and then piece together the class-hierarchy from the bottom up.

Bottom-up code browsing is also possible using CIV. As will be discussed in Section 7, a logical extension of CIV is to allow direct access to the source code through method nodes. In this manner, the lowest level of detail can be accessed through the high-level, inter-class view. Also, by opening a separate view for a particular class, the intra-class interaction can be seen. Then, multiple levels of detail can be shown simultaneously, giving the user a better understanding of how the low-level parts of the system interact at a higher level.

Section 3.1 describes the usefulness of impact analysis. While the impact analysis features of CIV are not fully functional, the basic design of the system will support the following process. A user selects a node, and suggests one of several types of changes. The system will then show the area of the view which is potentially affected by the change. By highlighting relational arcs which contribute to the effect, the system will also show *how* the affected objects were affected.

Intuitiveness

A qualitative argument demonstrates the intuitiveness of CIV. CIV is intuitive in that it contains call-graphs and def/use graphs which, aside from being displayed in 3-D, are similar to those drawn by hand in many design documents. Nodes are labeled, colored, and distinctly shaped so that the user can easily distinguish between classes, methods, and data. The concept of encapsulation is maintained by showing class methods as being internal parts of the class. Furthermore, classes can be viewed as either closed "black boxes" or expanded sets of functions, matching the basic premise of object-oriented programming — an object should have a public interface and a private implementation.

The IA features of CIV are also intuitive. Visual feedback about change impact is more easily understood than the textual report that would be generated by a program slice. The purpose of a program slice is to determine a subset of the code that contributes to a specified subset of the program's behavior. A visual representation of that subset should be more intuitive than a new version of the source code. In the best case, a visual display would be shown first, and then the text report could be studied. Then there would be some immediate recognition of the subset that the code represents.

Scalability

By making use of the natural encapsulation which exists in object-oriented software, CIV is capable of showing full-system views of medium-sized programs. The largest system that has been visualized by CIV is CIV itself, which contains approximately 10,000 lines of code and 60 classes. Figure 18 shows the initial inter-class view for CIV. Note that CIV is currently not capable of distinguishing classes defined in third-party libraries from classes defined in the application code. Therefore, the system being in Figure 18 actually contains about 150 classes.

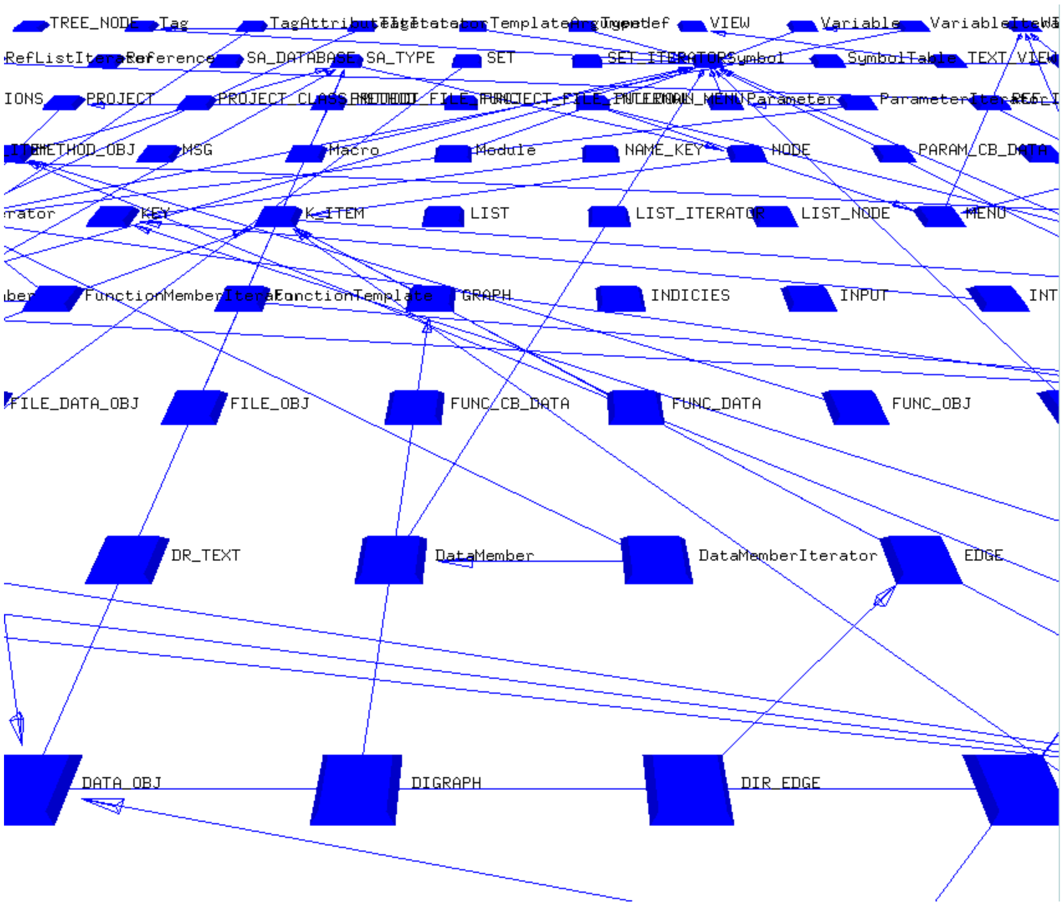


Figure 18: Inter-class view of CIV.

While the view in Figure 18 exhibits information overload, a user can use the view to gain useful insight into the system. For example, the view shows that a few classes (e.g. Symbol — second to last row, and fourth from the right) serve as base classes for a wide range of other classes.

As the number of classes increases, the inter-class view becomes more complex. However, the intra-class views maintain fairly constant complexity. That is, as systems grow large, the number of classes tends to increase much faster than the size of each class. This growth relationship is especially true when classes are reused. Rather than add new functionality to an existing class, designers may created a new, derived class which adds the new functionality. Figure 19 shows the internal view of a typical class in CIV.

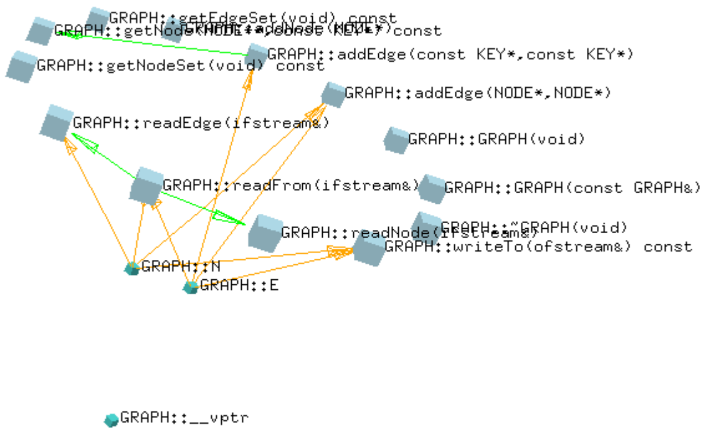

GRAPH::getEdgeSet(void) const
GRAPH::addEdge(const KEY*,const KEY*)
GRAPH::getNodeSet(void) const
GRAPH::addEdge(NODE*,NODE*)
GRAPH::readEdge(ifstream&)
GRAPH::GRAPH(void)
GRAPH::readFrom(ifstream&)
GRAPH::GRAPH(const GRAPH&)
GRAPH::readNode(ifstream&)
GRAPH::~GRAPH(void)
GRAPH::writeTo(ofstream&) const
GRAPH::N
GRAPH::E
GRAPH::__vptr

Figure 19: Intra-class view of class GRAPH in CIV.

The use of the class hierarchy alone has not solved the problem, it has only postponed it. Higher-level grouping is needed to reduce the total number of objects that are shown in a single view. For example, since CIV contains many small classes which are contained in only a few source files, the file view provides a less cluttered picture of the system. Figures 20 and 21 show a call-graph in the inter-file view for CIV. The use of higher-level grouping is discussed further in Section 7.

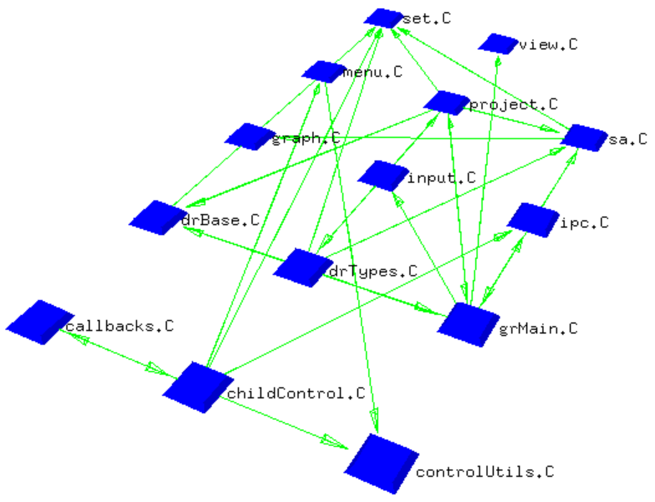


Figure 20: File view of CIV showing call-graph.

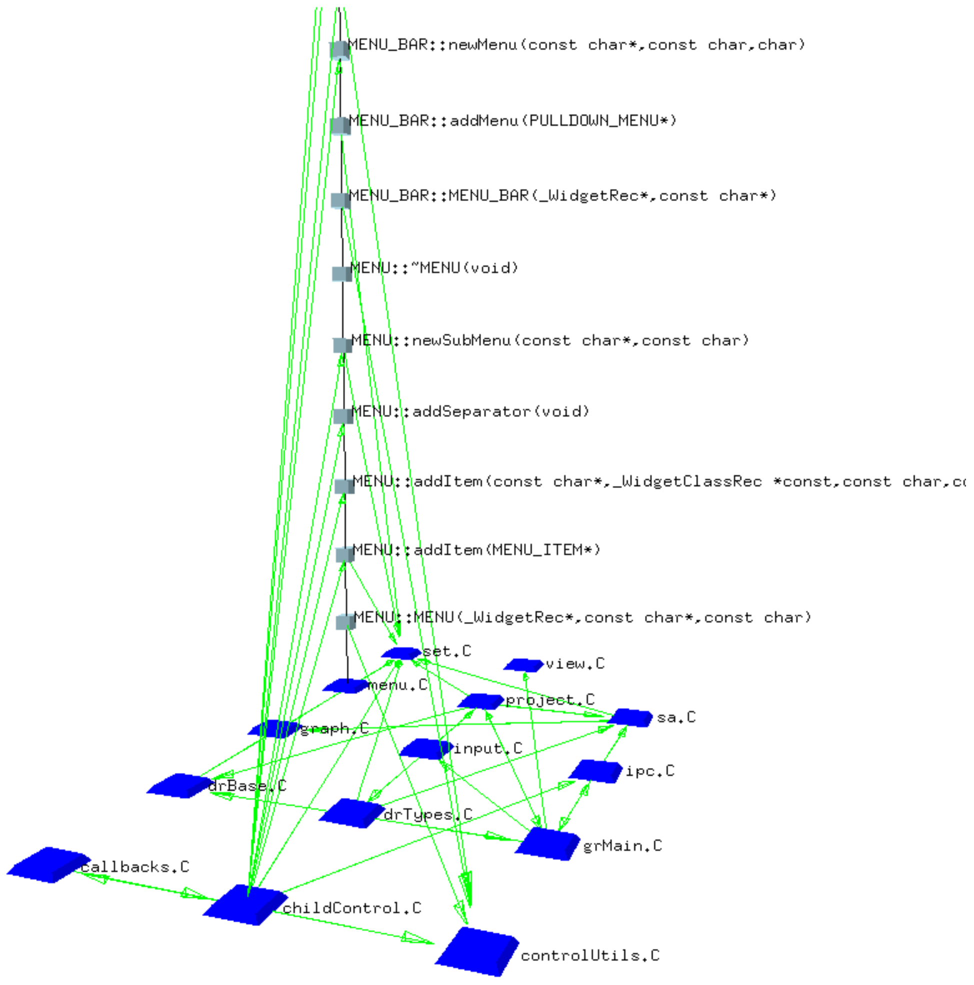

MENU_BAR::newMenu(const char*,const char,char)
MENU_BAR::addMenu(PULLDOWN_MENU*)
MENU_BAR::MENU_BAR(_WidgetRec*,const char*)
MENU::~MENU(void)
MENU::newSubMenu(const char*,const char)
MENU::addSeparator(void)
MENU::addItem(const char*,_WidgetClassRec *const,const char,c
MENU::addItem(MENU_ITEM*)
MENU::MENU(_WidgetRec*,const char*,const char)
set.C
view.C
menu.C
project.C
sa.C
graph.C
input.C
ipc.C
drBase.C
drTypes.C
grMain.C
callbacks.C
childControl.C
controlUtils.C

Figure 21: File view of CIV showing call-graph with file menu.C expanded.

6.2 Comparison with Other Systems

The analysis done in the previous section was largely subjective. A more objective analysis can be performed through comparison with other existing systems. In this section, visualization features of CIV are compared directly with those of HP SoftBench for the same input data. A less direct comparison with Imagix 4D is also given, but due to the lack of a full, working copy, this system could not be tested with the same data. Other systems described in Section 2 are not directly comparable with CIV due to differences in system capabilities.

HP SoftBench

The HP SoftBench Static Analyzer was compared with CIV for two input systems: an early prototype of CIV containing about 2000 lines of code and 15 classes; and the full implementation of CIV.

For comparison purposes, CIV generated all of the snapshots in the last section from the small system. We compare these snapshots to those generated by HP Softbench. CIV showed clear advantages in scalability. The figures from CIV show improved scalability and lower information overload than the snapshots generated by HP SoftBench.

Figure 22 shows the call-graph and class-inheritance graph created by SoftBench for the small system. The call-graph takes up slightly more than three virtual screens which can be viewed in SoftBench by scrolling. The inheritance relationships between the classes are shown as a separate graph. In Figure 7, CIV displays the entire class hierarchy in conjunction with a high-level view of the entire call-graph. More detail can then be shown by expanding classes of interest as is shown in Figures 9 and 10. Some scrolling of the image is required to see all of the methods in large classes, but the image can also be "zoomed out" to show the entire graph. In general, more information is shown in less space by CIV than by SoftBench.

CIV can also visualize larger systems with less clutter than SoftBench. Compare Figures 18 and 19 generated by CIV with Figures 23 and 24 generated by SoftBench for the same system. Figure 23 shows approximately $\frac{1}{30^{th}}$ of the entire call-graph, and Figure 24 shows approximately $\frac{1}{5^{th}}$ of the entire class-inheritance graph. Conversely, Figure 18 shows about 90% of the entire class-inheritance and high-level call-graph view, and Figure 19 shows 100% of the detailed internal view of a single class. So, while CIV exhibits information overload for the larger system, SoftBench has greater information overload problems.

In addition to scalability improvements, CIV has the advantage that it shows the relationships between the call and def/use graphs and the class-inheritance graph. With SoftBench it is not as clear how functions are grouped into classes. Also it is not clear how classes relate to each other at a high level, other than through inheritance.

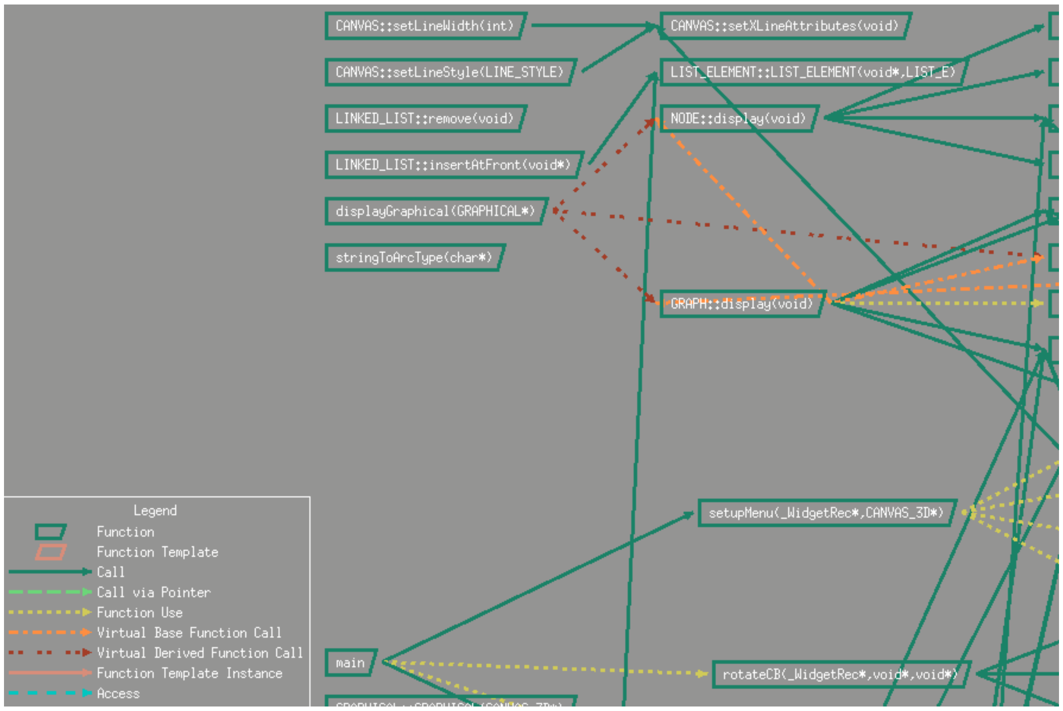
Imagix 4D

The visualization features of Imagix 4D provide primarily low-level views of functions. Unlike SoftBench, which shows a call-graph for the entire system, Imagix 4D shows the call-graph for only one function at a time. While these function-level views are supplemented by the hyper-text code browser, the actual visualization does not provide any additional information. The functions called by a single specific function can be determined by simply reading that function's code. In contrast, the high-level call information shown by CIV would require scanning the entire system to attain by hand.

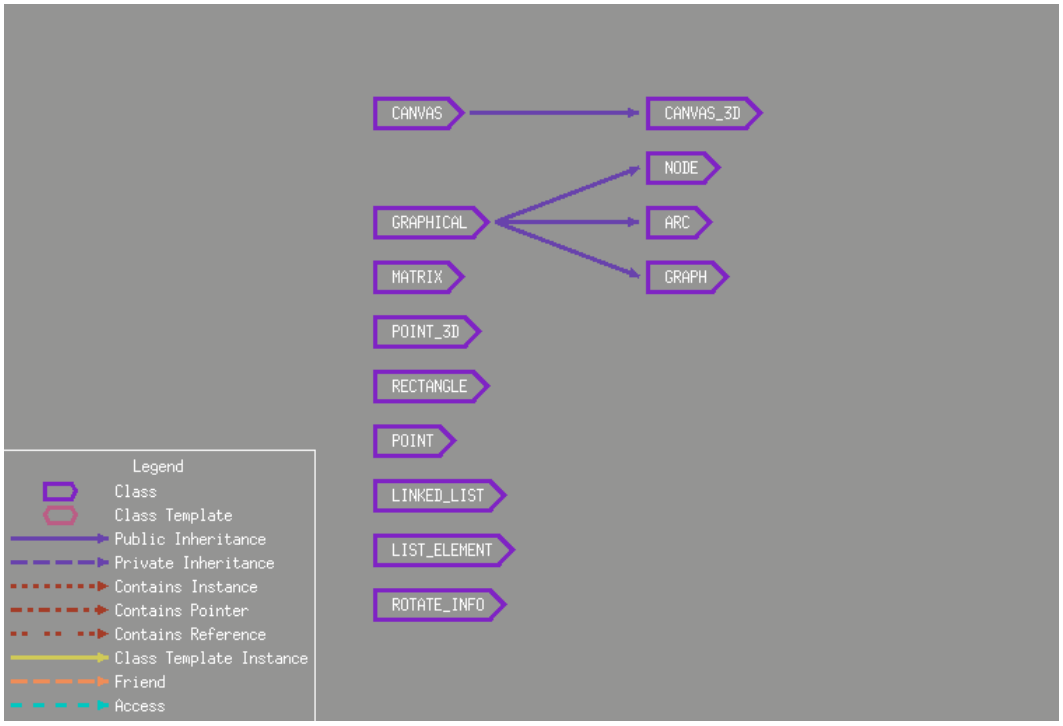
Class and file relational views are also provided by Imagix 4D. Figure 25 shows a file-inclusion graph produced by Imagix 4D (their class-inheritance graph is similar). Unlike CIV, files or classes cannot be expanded to see their internal methods and data. So, while these graphs provide useful information, it is unclear how they are related to the function-level views. Much like SoftBench, a user cannot easily determine how a particular method fits into the class view. By incorporating multiple relationships into single views, CIV clearly shows the interrelation between the relationships.

6.3 Impact Analysis in CIV

The impact analysis features of CIV are not fully functional, because the HP static analysis database does not contain the necessary control-flow information. High-level control-flow information such as function calling and def/use location is available, however, function level control-flow information, such as order of operation, cannot be determined. Without this level of control-flow information, standard dependency analysis techniques cannot be applied. While some amount of conservative estimation can be made through def/use information within the function, the lack of control-flow information makes pruning of the impact results difficult.


CANVAS::setLineWidth(int)
CANVAS::setLineStyle(LINE_STYLE)
LINKED_LIST::remove(void)
LINKED_LIST::insertAtFront(void*)
displayGraphical(GRAPHICAL*)
stringToArcType(char*)
CANVAS::setXLineAttributes(void)
LIST_ELEMENT::LIST_ELEMENT(void*,LIST_E)
NODE::display(void)
GRAPH::display(void)
setupMenu(_WidgetRec*,CANVAS_3D*)
main
rotateCB(_WidgetRec*,void*,void*)
Legend
Function
Function Template
Call
Call via Pointer
Function Use
Virtual Base Function Call
Virtual Derived Function Call
Function Template Instance
Access

a. Upper left-hand corner of call-graph.



b. Class hierarchy view.

Figure 22: HP SoftBench call-graph and class-inheritance graph for small test system.

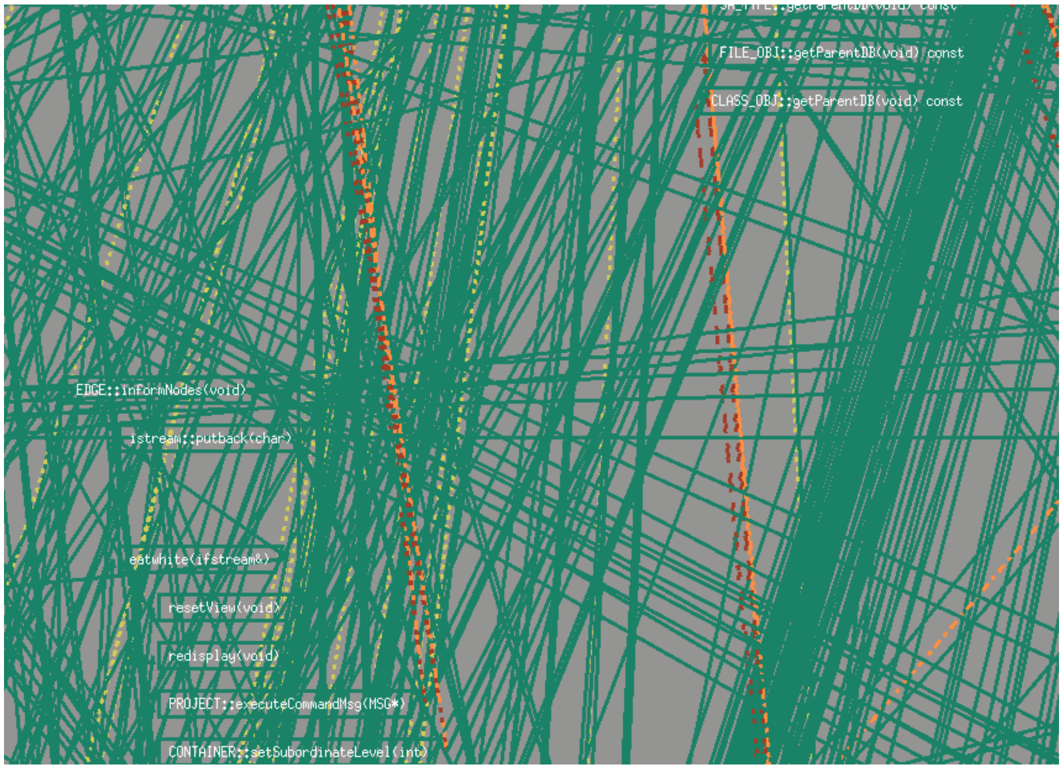

FILE_OBJ::getParentDB(void) const
CLASS_OBJ::getParentDB(void) const
EDGE::informNodes(void)
istream::putback(char)
eatwhite(ifstream&)
resetView(void)
redisplay(void)
PROJECT::executeCommandMsg(MSG*)
CONTAINER::setSubordinateLevel(int)

Figure 23: Part of the call-graph produced by SoftBench for CIV.

CIV provides a solid foundation on which more sophisticated impact analysis systems can be built. Given more full-fledged IA features, CIV could be used both to help understand a system and to make changes with more confidence. For example, suppose a maintenance engineer wants to make a change to a particular return value in a function. Through CIV, he/she could quickly determine the parts of the system that would likely be affected by the change. Some of the affected functions may be expected by the user, but others may be a surprise. Often, affects are propagated in indirect ways that would not be obvious without a visualization tool. For example, suppose a method in class A is modified, and the affect is felt by a method in class B. This method affects another method in class B that in turn affects a method in class C. In this case, CIV would show at a high level that classes A, B, and C are potentially affected. By expanding these classes, it would be obvious how class C was affected through class B. The ability to quickly recognize these unexpected relationships makes CIV a powerful tool for software maintenance. Understanding all aspects of a change can help the user make changes with a minimum of unwanted side effects. In many cases, there are several possible source code changes that will produce the desired effect. A system like CIV can greatly reduce the effort of choosing between these changes.

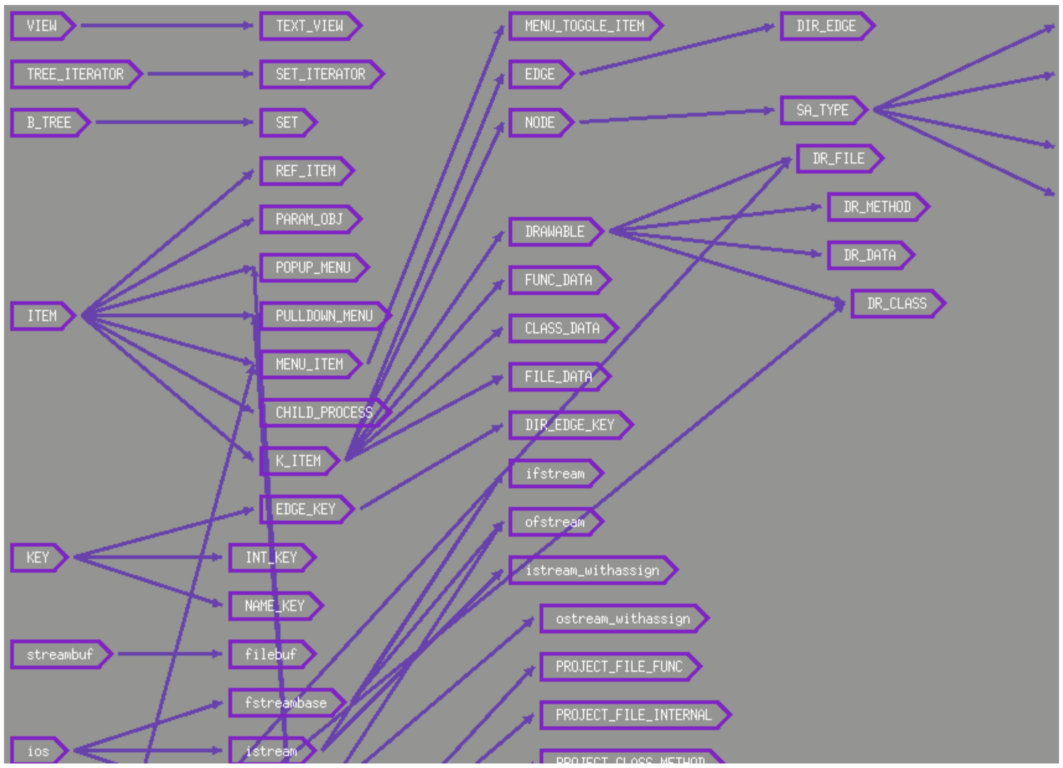


Figure 24: Part of the class-graph produced by SoftBench for CIV.

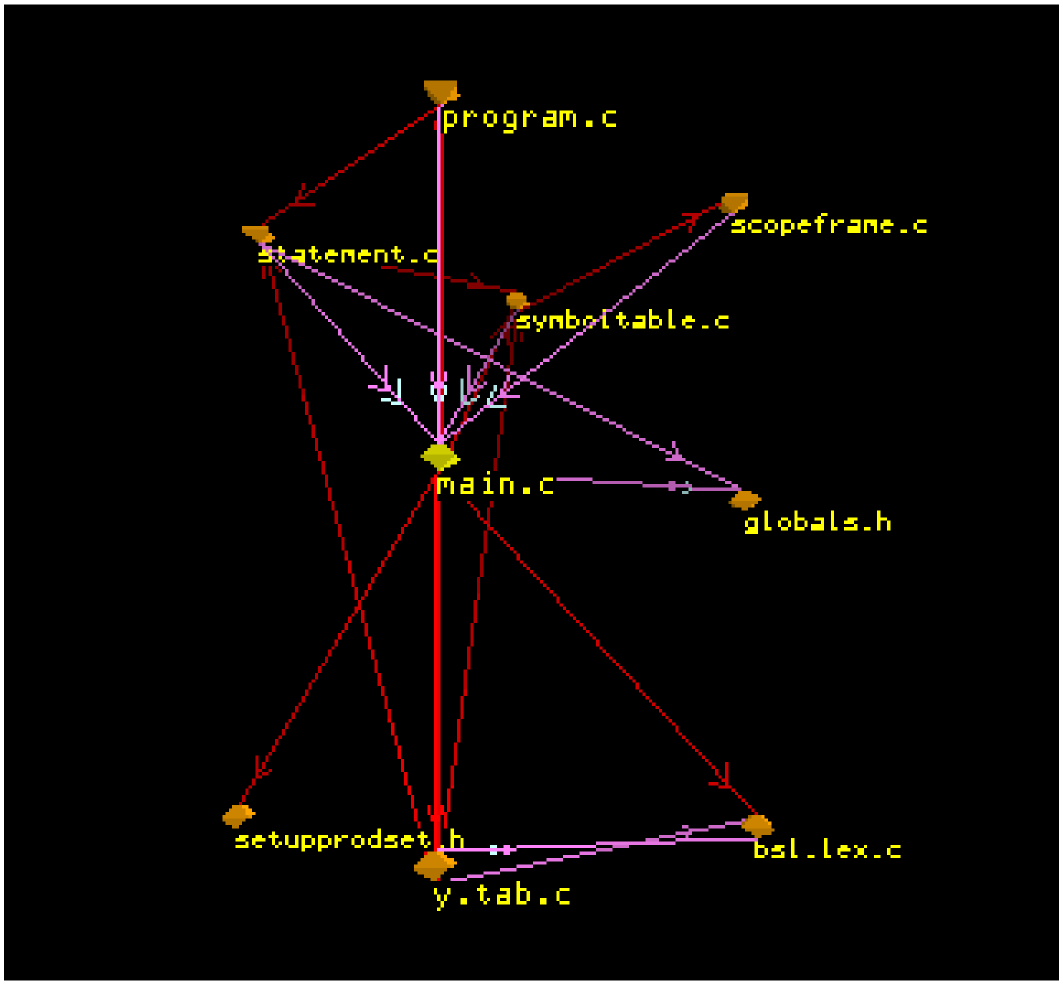


Figure 25: File inclusion graph generated by Imagix 4D.

7 Conclusions and Future Work

Most prior SV systems are not scalable, and thus cannot adequately visualize large-sized, real-world systems. Without scalability, acceptance of these systems by industry will be limited. The use of 3-D graphics together with a hierarchical viewing scheme can provide visualizations that are limited in clutter without sacrificing the amount of information shown. Evaluations using our prototype CIV system support the hypothesis that a hierarchy of views with multiple levels of resolution is useful for increasing scalability. Examples suggest that abstract information, such as the results of impact analysis, can be visualized at a high level for large systems. Quick access to low-level views then provides the missing details. Furthermore, by allowing expansion of objects within a view, in addition to providing separate low-level views, an intermediate level of detail is provided. This overlap of high-level and low-level information adds high-level context to the low-level view.

The use of 3-D graphics provides a means of showing more information on the screen at one time. While individual 2-D images from the visualization may appear somewhat cluttered, the user can interactively rotate the view so that the information of interest can be seen clearly. The primary advantage of 3-D viewing is through the ability to interactively manipulate the view.

Impact analysis remains a difficult problem. Without a high degree of semantic understanding, IA systems can only produce approximate results. Since automated IA systems do not produce exact results, the IA process should be interactive in nature [17]. Feedback with the user is necessary to prune the potential impact set as the analysis propagates through the system. By highlighting potentially affected areas of the system, our prototype CIV provides quick, intuitive feedback about the potential impact of a change. The user can quickly assess both *where* and *how* the impact of a change is felt. This makes it useful as an interactive tool for IA. Additionally, the IA results are intuitive at all levels of resolutions, since they show a subset of the currently displayed graph. This is easily interpreted as the "physical area" affected by the given change.

Extensions to CIV and Open Problems

To visualize very large systems, the hierarchy of views needs to be extended. Systems need to be grouped at a higher level than is currently supported by CIV. For example, classes can be grouped into logical sub-systems. Unfortunately, determining these groups is a difficult problem. Classes that are related through inheritance might not be part of the same logical sub-system. For example, a class might be derived from both "GRAPHICAL" and "LIST_ITEM" base classes — logically, the derived class is meant to be part of the graphics sub-system, and not part of the dynamic storage sub-system of which LIST_ITEM is a part. The use of packages as in Java provids a language defined mechanism for programmers to specify hierarchies of subsystems, which can be used in SV.

Another problem with the addition of more hierarchical levels is the lack of additional spatial dimensions for expansion. One solution is to use in-plane expansion similar to that shown in Figure 5(a). Another possibility is to have the option of either expanding the classes in the groups along the third axis, or opening a separate inter-class view for the group.

A number of simpler extensions can be added to CIV to make it more robust. First, the layout of classes in the inter-class view does not group related classes together — a simple grid is used. A more sophisticated layout scheme can replace the grid with an organized 2-D layout of the class hierarchy. This layout would increase cohesion in the inter-class view, and increase its intuitiveness. Techniques for this type of layout are described by Bertolazzi, Battista, and Liotta [3].

Second, function-level information could be added. The text of a function could be viewed in a separate text-window by clicking on a function object. The graphical display of CIV could be used to access a hyper-text code browser similar to that used in Imagix 4D. This would provide lower-level information than is currently available, as well as a better logical connection between the source code and the visualization. The advantage of this over Imagix 4D, is that the visualization would show higher-level information than the code view. Immediate access to the code of individual functions would provide better understanding of the relationships that exist in the higher-level call and def/use graphs. In this way, the code view and abstract visualization would complement each other.

Dynamic analysis could also be incorporated into CIV by integration with a run-time debugger. Nodes and arcs could be highlighted as the execution progressed. This would provide high-level information that would not be immediately obvious in a text-based debugger. For example, when tracing into a function with a debugger, it may not be immediately obvious that the code being executed is in a different module or file. It may also not be obvious whether a modified variable is a class member or a global variable. Also, dynamic binding could be shown by highlighting a path through the class hierarchy from the called function to the inherited virtual function that is actually executed.

Other simple improvements have been suggested. Curved arcs could be used in situations where multiple relationships exist between two nodes. For example, if two class nodes have both a call arc and an inheritance arc between them, then the inheritance arc could be drawn straight, and a call-arc could curve below the plane. This would make it clear that two separate relationships exist between the two nodes.

Another suggestion is to encode additional information in the size of the nodes, and thickness of arcs. For example, by making the size of a class node proportional to the amount of code in the implementation of the class (or the number of functions in the definition of the class), the user could quickly see which classes make up most of the system. Also, the thickness of a call-arc between two classes could indicate the number of actual calls represented by the arc.

Finally, empirical experiments need to be performed to more rigorously test our hypotheses. Experiments need to directly compare the usefulness, intuitiveness, and scalability of CIV with other existing systems. However, since these are highly subjective properties, they can only be measured indirectly. For example, the amount of time needed for an average programmer to become familiar with a new software system using each of a set of tools, can be used to indicate the relative usefulness of those tools. Repeating this experiment with software systems of increasing size would provide an indication the relative scalability of the tools in terms of usefulness.

References

[1] R. Arnold and S. Bohner. Impact analysis — towards a framework for comparison. *Proceedings IEEE Int. Conference on Software Maintenance*, pp. 292–301, 1993.

[2] T. Ball and S. Eick. Software visualization in the large. *Computer*, pp. 33–43, April 1996.

[3] P. Bertolazzi and G.L.G. Di Battista. Parametric graph drawing. *IEEE Transactions on Software Engineering*, 21(8):662–673, August 1995.

[4] G. Duggan. Visualizing c++ programs, April 1995. Hewlett Packard Application Development White Paper.

[5] L. Ford. Automatic software visualization using visual arts techniques. Research Report 279, University of Exeter, Exeter, U.K., September 1993.

[6] L. Ford. How programmers visualize programs. Research Report 271, University of Exeter, Exeter, U.K., March 1993.

[7] Hewlett Packard. *Exploring HP SoftBench: A Beginner's Guide*, 1989.

[8] Hewlett Packard. *HP SoftBench Static Analyzer: Analyzing Program Structure*, 1989.

[9] D. Jackson and D. Ladd. Semantic diff: A tool for summarizing the effects of modifications. *Proc. IEEE Int. Conference on Software Maintenance*, pp. 243–252, 1994.

[10] D. Jerding and J. Stasko. Using visualization to foster object-oriented program understanding. Technical Report GIT-GVU-94-33, Georgia Institute of Technology, Atlanta, GA, July 1994.

[11] F. Brooks Jr. No silver bullet. *Computer*, pp. 10–19, April 1987.

[12] J. Loyall and S. Mathisen. Using dependency analysis to support the software maintenance process. *Proceedings IEEE Int. Conference on Software Maintenance*, pp. 282–291, 1993.

[13] M. Moriconi and T. C. Winkler. Approximate reasoning about the semantic effects of program changes. *IEEE Transactions on Software Engineering*, 16(9), September 1990.

[14] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Company, 1994.

[15] N. Pennington. Comprehension strategies in programming. In *Proceedings of the Second Workshop on Empirical Studies of Programmers*, pp. 100–112, 1987.

[16] B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1994.

[17] J-P. Queille and J-F. Voidrot. The impact analysis task in software maintenance: A model and a case study. *Proceedings IEEE Int. Conference on Software Maintenance*, pp. 234–242, 1994.

[18] D. Frederick Sklar and C. Brown. *SPHIGS for ANSI-C*, v1.0 edition, March 1993.

[19] E. Soloway, B. Adelson, and K. Ehrlich. Knowledge and processes in the comprehension of computer programs. In M. Chi, R. Glaser, and M. Farr, editors, *The Nature of Expertise*, pp. 129–152. A. Lawrence Erlbaum Associates, 1988.

[20] M. Staples. Scalability and software visualization. Master's thesis, Colorado State University, 1996.

[21] J. Stasko. Three-dimensional computation visualization. Technical Report GIT-GVU-92-20, Georgia Institute of Technology, Atlanta, GA, 1992.

[22] A. von Mayrhauser. *Software Engineering: Methods and Management.* Academic Press Inc., 1990.

[23] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, August 1995.

[24] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, pp. 352–357, 1984.

[25] L. Wu. A tool for the visualization of c++ classes, methods and their relations. Class Project, December 1993.