Visualizing Program Slices

Thomas Ball (tball@research.bell-labs.com)
Stephen G. Eick (eick@research.bell-labs.com)

October 1994

Appears in Proceedings of the 1994 IEEE Symposium on Visual Languages, pp. 288-295, October 1994.

Copyright © 1994 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Visualizing Program Slices

Thomas Ball & Stephen G. Eick
AT&T Bell Laboratories
1000 E. Warrenville Rd./Naperville, IL 60566
tball,eick@research.att.com

October 14, 1994

Abstract

Program slicing is an automatic technique for determining which code in a program is relevant to a particular computation. Slicing has been applied in many areas, including program understanding, debugging, and maintenance. However, little attention has been paid to suitable interfaces for exploring program slices. We present an interface for program slicing that allows slicing at the statement, procedure, or file level, and provides fast visual feedback on slice structure. Integral to the interface is a global visualization of the program that shows the extent of a slice as it crosses procedure and file boundaries, and facilitates quick browsing of numerous slices.

This paper appears in the proceedings of the 1994 IEEE Symposium on Visual Languages.

1 Introduction

Understanding the behavior of a large software system is a complex and daunting task. Program slicing, an automated program decomposition proposed by Mark Weiser[13, 14], aids in this endeavor by reducing the amount of code to be examined at any point in the process. Informally stated, a slice contains those components of a program that can potentially affect some component of interest. A slice aids program understanding by focusing attention on a smaller and relevant subprogram. Figure 1 shows a slice of a simple function, *link*. The slice point is the statement '$return(y)$;', which has been highlighted. Statements in the slice are colored, while statements outside the slice ('$x = y;$, $p[x] = y;$') are not.

Although a slice may reduce the amount of code a programmer must examine, slices may be quite large and complicated, especially for large systems. Current slicing interfaces, which are based on text browsers (such as shown in Figure 1) or syntax-directed editors [1, 6, 12], are inadequate to the task of exploring slices. In these interfaces, a slice is formed by selecting a statement or expression (the slice point) and invoking a slice command. As a result, statements in the slice are highlighted or colored. If the slice crosses procedure or file boundaries, additional browsers may be opened or browser commands may be invoked to view other entities. Such interfaces

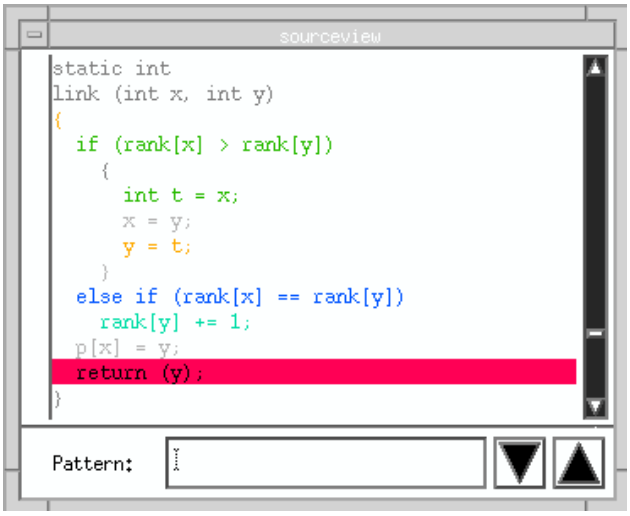

Figure 1: A simple slice

burden the programmer with the task of determining the extent of a slice and finding interesting slice features. Determining code that lies outside of the slice is also difficult. Navigation through a slice is cumbersome, especially when the slice crosses procedure and file boundaries, which will be the rule for large programs. Simply put, browser-based interfaces get in the way of program slicing rather than expedite it: it is difficult to examine the many different slices of a program because so much effort is required to see just one slice. The slice is, for the most part, invisible in such interfaces.

We have developed SeeSlice, an interactive slicing interface to address the above problems. Our motivation for developing SeeSlice was the need to better visualize the data generated by slicing tools (in particular, for a slicing tool we have built). We are investigating how to apply slicing to large legacy systems to help programmers better understand and comprehend their behavior.

SeeSlice uses a reduced visual representation of programs pioneered by Eick et. al[4] that has been extended in several ways to support slicing. Figure 2 shows a snapshot of SeeSlice's main display. Files are displayed as columns that contain representations of procedures. Procedures are displayed either in a open form in which each line of code is displayed as a thin row (see the first procedure in the first file, *qpt.c*), or in a closed form that hides their underlying code (see the first procedure in the second file). A slice is formed with respect to a code entity simply by pointing at it with the mouse. The statements, procedures and files in the slice are immediately colored. The slice is dynamically updated to track the mouse, making patterns and shared slice structure apparent. A source browser may be opened to view the actual code.

The SeeSlice interface facilitates slicing by making slices fully visible to the user, even as they extend across many procedures and files. The global hierarchical overview of a program, combined with a highly interactive interface, allows the user to quickly examine many slices, find interesting slice features, and browse through a slice.

Section 2 introduces program slicing and describes the tool we have built to find slices. Section 3 describes the slicing interface and visualization. Section 4 steps

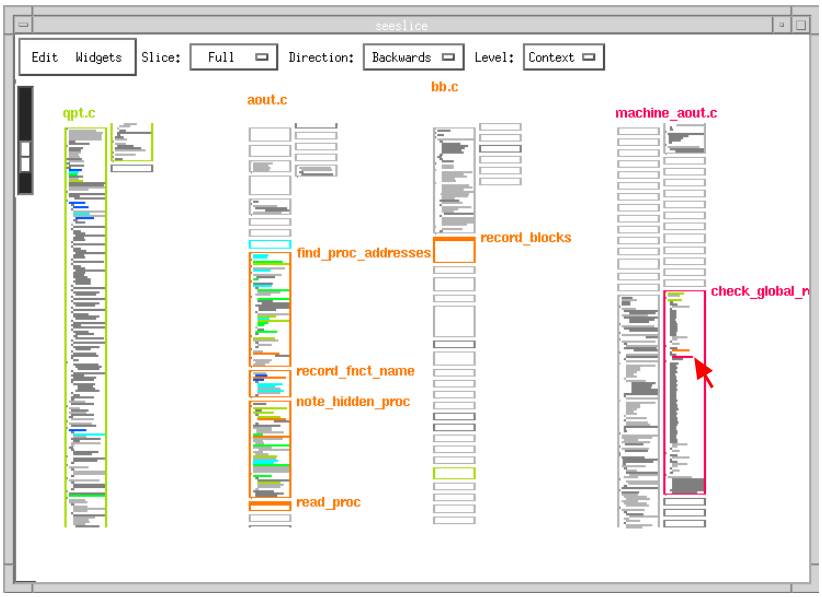

Figure 2: SeeSlice's main display. Mouse indicates slice point

through an application of the slicing visualization to a program. Section 5 discusses related work and Section 6 summarizes.

2 Program Slicing: Preliminaries

Slices are generally classified along two dimensions. First, a slice has a direction: backwards or forwards. A *backwards* slice with respect to a component C identifies those components that affect C, while a $forward$ slice with respect to C identifies those components affected by C. Second, a slice can be computed by analyzing a program's source code (referred to as a *static* slice) or by analyzing one or more executions of a program (a *dynamic* slice). Static and dynamic slices can be automatically constructed by a variety of methods[14, 10, 7, 8]. All of these approaches involve examining a program's data and control dependences, for which we give informal descriptions (for more rigorous definitions, see[5]).

Statement s is *data dependent* on statement r if r writes into a variable V that s subsequently reads from (with no intervening writes to V). In the function in Figure 1, the statement '$return(y)$;' is data dependent on statement '$y = t$;', which is data dependent on '$int\ t = x$;'. Analyzing data dependences in the presence of pointers and arrays is a difficult task given only a program's source code (the problem is generally undecidable). However, the exact data dependences for a particular execution of a program can be easily computed by examining the execution's address trace (the tradeoff is that not all dependences in the program may arise in a given execution). Data dependences can span procedure and file boundaries.

Statement s is *control dependent* on statement r if r is a predicate that can control whether or not s executes. Control dependences span statements from the same proce-

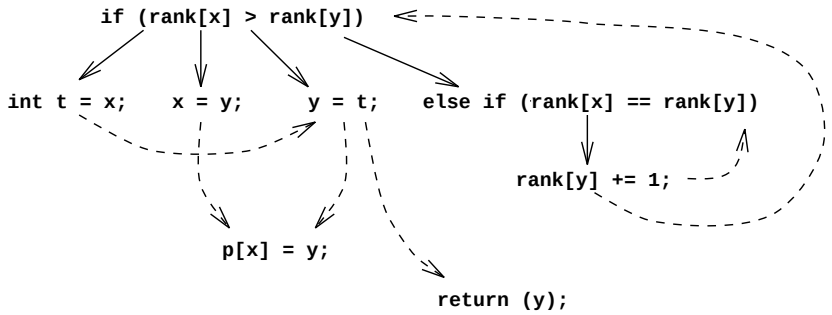

Figure 3: The program dependence graph of the function *link* from Figure 1. Control dependence edges are solid and data dependence edges are dashed

dure but do not cross procedure or file boundaries. In Figure 1, the statement '$y = t;$' is control dependent on its enclosing if conditional. The *return* statement is not control dependent on either if statement since it executes regardless of the control-flow path through the procedure.

Combined, the data and control dependences form the edges of a directed graph (the *program dependence graph*[5]) in which the vertices are the statements of the program. The backwards or forwards slice with respect to a statement may be found by backwards or forwards transitive closure in this graph, identifying the set of statements in the slice. Figure 3 shows the program dependence graph of the function from Figure 1. The dashed curved edges are data dependences and the solid straight edges are control dependences. The backwards closure from '$return(y);$' includes every vertex in the graph except for '$x = y;$' and '$p[x] = y;$', which corresponds to the slice shown in Figure 1.

We have constructed a tool for building the dynamic program dependence graph of a program's execution. We use the QPT instrumentation tool to generate program traces[2, 9]. QPT instruments an executable file with code to generate a trace of interesting events (such as the addresses written to and read from). This trace is piped to a trace analyzer that extracts the dynamic data and control dependences. Since QPT instruments executable files, dependences are found between machine instructions. The symbol table in an executable file allows us to map the dependences between machine instructions to dependences between their corresponding statements at the source level. The result of the analysis is a statement-level dynamic dependence graph. Performing the execution analysis at the object code level allows us to analyze programs written in different languages.

3 Slicing Visualization and Interface

The SeeSlice interface displays a program in a compact representation. Programs are organized into three levels: files (modules), procedures, and statements. Previous slicing interfaces focus on slicing at the statement level. We believe that it should be equal easy to slice at any level in the hierarchy (statements, procedures, files).

Therefore, the reduced representation must clearly show these three levels of hierarchy. In order to examine many slices, the interface must be highly interactive. Slicing is accomplished simply by pointing at the representations of files, procedures, and statements. The display is updated in real-time to show the slice associated with the current mouse position, which allows the user to quickly examine many different slices and find interesting patterns. We now describe the visual and interactive aspects of the interface in greater detail.

3.1 Visualization

As shown in Figure 2, each file in a program is represented as a column of procedures (the column may wrap around if the file is large), with the file's name at the top of the column. Procedures in a file are displayed in one of two forms: in *closed* form, a procedure is a (partially filled) rectangle whose size encodes the number of lines in the procedure (*i.e.*, procedure *record_blocks*); in *open* form, each line of text in a procedure maps to a line of pixels, with indentation and length reflecting that of the text (*i.e.*, procedure *find_proc_addresses*). This reduced representation shows file and procedure sizes, as well as the internal control structure of open procedures. The display also shows the name of the procedure containing the slice point (*check_global_regs*) and the names of the procedures one step away in the slice (*i.e.*, procedure *record_blocks*). Displaying more procedure names than this tends to clutter the display.

A slice that spans many procedures may contain relatively few statements in proportion to the number of executed statements in those procedures. On the other hand, we may encounter "heavy" slices that contain most of the executed code in each procedure. To distinguish these cases, there is an option to fill each closed procedure to indicate the percentage of executed statements in the procedure that are in the slice. By quickly scanning the display, a user can gauge the size of a slice (in terms of number of statements) in addition to the number of procedures and files that it spans. In Figure 2, we immediately see that only a small amount of the executed code in procedure *record_blocks* is in the slice.

Colors are used in the display to distinguish executed code from unexecuted code (dark gray vs. light gray), and the code in a slice from the code outside a slice (color vs. gray). Distinguishing executed code from unexecuted code in the display is necessary so that it is clear which code can be sliced. Components in the slice are color-coded from the rainbow scale by their shortest path distance to the slice point (in the dependence graph), where red is the slice point and each color lower in the rainbow spectrum represents one step away in the shortest path. This allows easy identification of the immediate predecessors of the slice point. A closed procedure's color is the "hottest" color in the procedure. A file's name is similarly colored.

3.2 Interaction

Figure 4 shows all the components of the slicing interface. Our interface supports the continuous slicing of a program. Rather than require the user to invoke a slice command for each desired slice, slicing continues until the user requests that it halt (through a button click). Slicing is accomplished simply by pointing at an entity in the reduced representation. Pointing to a closed procedure forms a slice with respect to all

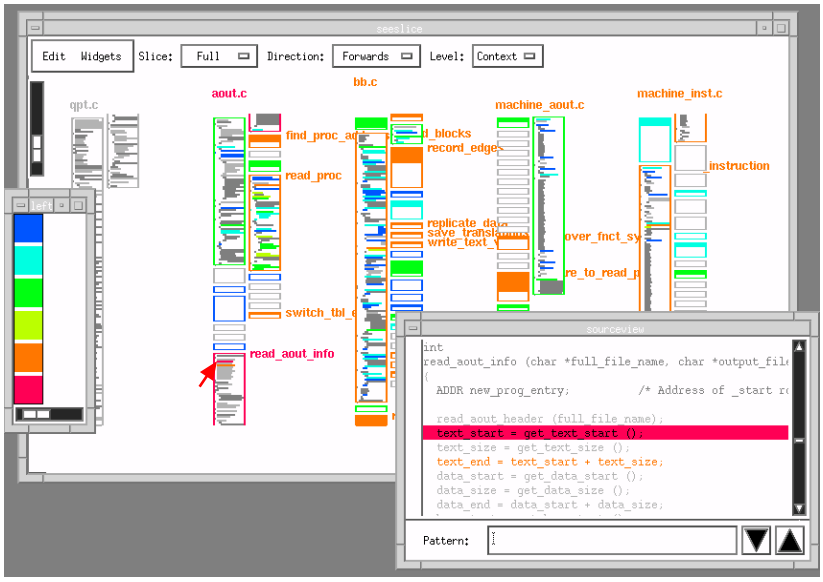

Figure 4: The full SeeSlice interface. A forward slice on the assignment to *text_start*

the executed statements in that procedure (*i.e.*, a transitive closure from all the vertices representing statements in the procedure). Similarly, pointing to a file's name forms a slice with respect to all the statements in the file. If a procedure has been opened, then slices may be taken with respect to the individual statements in that procedure. Figure 4 shows a forward slice on an assignment statement. Each time a slice is formed, the previous slice is erased. However, a mouse drag disables erasure of the previous slice, allowing the user to slice with respect to a set of program components.

A slider on the left of the main display scales the size of the closed procedures. When browsing at the procedure level it is easier to identify patterns if all the procedures are of equal size. Figure 5 shows an example of procedures scaled to reflect their true size and Figure 6 shows an example of procedures scaled to nearly equal size.

The interface supports forward and backwards slicing (or both at the same time). In addition, there are several filters for controlling the transitive closure. For example, it is possible to restrict the slicing operation so that it does not propagate out of the strongly-connected component containing the slice point. This is useful for identifying recurrences (cycles) in the program's computation that would not be apparent from a simple backwards closure.

In a separate window on the left side of the display (see Figure 4) is a mouse-sensitive color-selector with an associated slider. The slider controls how many steps the transitive closure will expand for each slice operation. As the user will often be interested in a small area around the slice point rather than points very far away, it is not necessary to compute the full transitive closure for each slice operation. The color selector represents the number of steps in the closure. Through this selector, the user may selectively enable and disable slice colors, which is reflected in the reduced representation.

To examine a particular slice, slicing can be turned off (which fixes the current slice). Files and procedures not in the slice can be elided and those files and procedures in

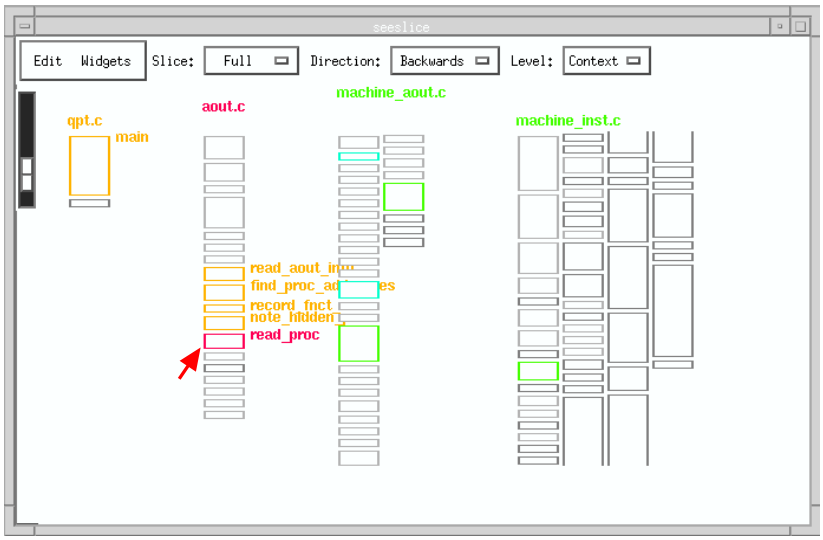

Figure 5: A backward slice on procedure *read_proc* in file *aout.c*

Figure 6: A forward slice on procedure *write_insts* in file *aout.c*

the slice can be rearranged on the display, sorted by their dependence distance to the slice point. This allows exploration of the source code near to the slice point with a minimum of mouse motion, as files and procedures that are close to the slice point (in terms of dependence distance) are now also spatially close to it. The order of statements within a procedure is not altered.

To view the actual source code, a source browser may be opened (see Figure 4). Lines of text in the browser are colored as in the reduced representation. The browser view tracks the current mouse position in the reduced representation, allowing the user to quickly move from the code in one file or procedure to the code in another. If the user desires, the code within a file may be browsed through standard browser commands and slices may be requested from the browser.

4 An Example

This sections shows how the slice visualization can be used to quickly find interesting relationships between a program's components. We applied the slicing tool to the QPT profiling/tracing tool. QPT is written in the C language (about 12,000 lines of code and 300 procedures, not including libraries). Figure 6 shows all the files and procedures in the QPT program. QPT has three basic steps: (1) read in an executable file; (2) determine points in the executable to add instrumentation code to; (3) write out the instrumented executable file.

Slicing can be used to answer a number of questions about the relationships between program components, such as:

- What groups of procedures and files participate in a computation?
- What code and variables are crucial to the computation of the program? (Changing such code and variables will affect the behavior of many parts of the program.)
- Does a file or procedure contain several independent computations or just one?

4.1 Highly interdependent and shared code

Slicing on the procedures in the file *aout.c* immediately reveals that there are five highly interdependent procedures in the file (see Figure 5). A slice on any one of the five procedures includes all five procedures. Furthermore, as Figure 5 shows, the backwards slice with respect to these procedures is very small. However, a forward slice with respect to these procedures includes almost all the procedures in the program. These five procedures (*read_aout_info*, *find_proc_addresses*, *record_fnct_name*, *note_hidden_proc*, *read_proc*) collaborate in reading in an executable file. Not surprisingly, most of the program depends on the data structures initialized by these procedures.

4.2 Inter-file functionality

What are the functions of the other procedures in file *aout.c*? Figure 6 shows the forward slice with respect to the first procedure in *aout.c*, *write_insts*. This forward slice is small and spans four files. These procedures collectively output the instrumented executable file. The procedures that deal with the executable file format are in *aout.c*, those that deal with machine-independent instrumentation are in *prof_trace.c*

and those that handle machine-dependent instrumentation are in *machine_inst.c* and *machine_prof_trace.c*. This is a natural organization for porting the QPT tool to different platforms, but one that makes it harder to discover the inter-file functionality. The slice visualization makes this apparent immediately.

4.3 Important variables

If we examine the text of the procedure *read_aout_info* (see the browser in Figure 4), we find that it initializes a number of global variables (*text_start*, *text_size*, *data_start*) corresponding to the starting addresses and sizes of various segments in the executable file. The size of the forward slice with respect to each variable's definition will indicate how crucial each variable is to the program's computation. Figure 4 shows a forward slice with respect to the variable *text_start*. Each procedure displayed with a name directly references *text_start*. Other procedures in the slice are indirectly influenced by the value of *text_start*. While only five files are shown in this figure, the forward slice with respect to *text_start* influences most of the code in the program.

In this quick analysis (performed by slicing only on procedures and statements in the file *aout.c*) we have learned the following:

- There are five highly interdependent procedures in the file that read the input to QPT, on which most of the program is dependent.
- There is a set of interdependent procedures spanning four files that collectively output the instrumented file.
- The variable *text_start* influences a large portion of the program.

5 Related Work

None of the program slicing systems that we are aware give the user a global overview of a program or allow quick examination of many slices. These systems are based around a text or syntax-directed browser and support slicing mainly at the statement level. A few examples of such program slicers are the Wisconsin Program-integration System[12], Spyder[1], and the Surgeon's Assistant[6].

The SeeSlice display presented here is based on the previous work of Eick et. al on the SeeSoft program visualization system[4]. In the SeeSoft display, each file of a program is displayed in what we refer to here as open form, in which each line of a file is mapped to a line in the display, with length and indentation reflecting that of the underlying source text. A line's color encodes an associated statistic (for example, a count of how many times the line executed[3]). We found that the totally line-based orientation of this display does not work well for slicing, as program abstractions such as procedures are not visually apparent and cannot be pointed to or selected, and as there is no way for the user to suppress or eliminate line-level detail when it is not needed.

The problem of visualizing program slices is part of a larger program visualization problem: How can visualization make the interdependencies and relationships between program components apparent to the user? Two main challenges to visualization are the huge number of such relationships and the fact that many of these relationships are not necessarily localized in the code. Slicing eliminates relationships (and code) that

are not relevant to a point of interest, but still leaves us with the above challenges for the remaining, relevant relationships. SeeSlice effectively shows what code is inside and outside a slice in a global fashion, but does not provide direct assistant in navigating through the dependence relationships in this code, as do other tools[11].

6 Summary

We have presented a visualization method and interface for querying and displaying program slices. The SeeSlice tool allows a user to quickly request and examine many different program slices. A reduced visual representation of a program that displays the various levels of program hierarchy makes it possible to query, display, and browse slices in a highly interactive fashion.

There are many different possible applications for SeeSlice. As part of a system for program understanding and reverse engineering, the tool could be used to identify related procedures and files and to extract and restructure code. Applied to debugging, SeeSlice provides help in identifying code that contributes to anomalous program behavior.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software-Practice and Experience*, 23(6):589–616, June 1993.

[2] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages, (Albuquerque, NM)*, pages 59–70. ACM, Jan. 19-22 1992.

[3] S. G. Eick and J. L. Steffen. Visualizing code profiling line oriented statistics. In *Proceedings of Visualization '92*, pages 210–217. IEEE Computer Society Press, Oct. 19-23 1992.

[4] S. G. Eick, J. L. Steffen, and E. E. Sumner Jr. Seesoft - a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, Nov. 1992.

[5] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(5):319–349, July 1987.

[6] K. B. Gallagher. Surgeon's assistant limits side effects. *IEEE Software*, 7:64, May 1990.

[7] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[8] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(10):155–163, Oct. 1988.

[9] J. R. Larus. Efficient program tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[10] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (ACM SIGPLAN Notices)*, 19(5):177–184, May 1984.

[11] S. P. Reiss. A framework for abstract 3d visualization. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 108–115, Aug. 24-27 1993.

[12] T. Reps. The wisconsin program-integration system reference manual: Release 2.0. Technical Report Unpublished report, University of Wisconsin, Madison, WI, July 1993.

[13] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7), July 1982.

[14] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.