Analyzing Programs via Decomposition Slicing: Initial Data and Observations*

Keith Gallagher
Loyola College
Baltimore, MD

Liam O'Brien
Software Engineering Institute
Pittsburgh, PA

Abstract

We use Unravel to perform decomposition slicing on a collection of programs. The number of decomposition slices obtained is overwhelming. We reduce the complexity by displaying equivalently computed slices as a single node. Best case reductions show an 80% reduction in the number of nodes needed to display a decomposition slice graph. We present initial results of a continuing study using this reduction technique.

1 Introduction

Program comprehension is the process by which software engineers gain an understanding of a system. Since the amount of raw information is overwhelming, abstraction techniques must be applied to aid the comprehender. There are a plethora of such techniques that involve processing, abstracting and then visualizing program information for the comprehender. Call graphs are a simple example of such a technique.

Program slicing is another of these techniques. However, program slices can themselves be very large and difficult to comprehend. So, rather than examining slices themselves, we examine the *relationship* between slices, using

*Both authors acknowledge partial support from CSIRO Mathematical and Information Sciences, Canberra, Australia.

the simple is-contained-in relation, and represente the entire slice as one node in a partially ordered graph. Using this relation also proved to be problematic also, since there would be one node in the graph for each variable declared in the program.

Two observations helped us assuage this problem. The first was that the graphs exhibited a wide "fan-out" at the bottom. Examining the content of the slices showed them to be empty; they had no executable statements! It was a simple exercise to remove these from the graph. The second observation was that many of the slices had the same size. This lead us to investigate whether or not the slices themselves were actually equal. It turns out that many of the slices are the same. In the best case analyzed, the reduction removes over 80% of the slices that were equal to some other slice. So we can represent many slices as one node in the graph. Thus, we have a way to significantly simplify the presentation while not losing information content proportional to the size of the reduction.

We have begun to apply this reduction technique to real programs and herein we report some initial data of this work-in-progress. This work is tightly focused and self-contained; we are collecting and examining data.

Organization

This paper is organized into 5 sections. Section 2 provides background and context; section 3 gives the preliminary data; section 4 discusses results; and section 5 outlines future work and concludes.

2 Background

A program slice, $\mathbf{SLICE}_{(v,n)}(p)$, of program p on variable v, or set of variables, at statement n yields the portions of the program that contributed to the value of v just before statement n is executed [8]. The pair (v, n) is called a *slicing criterion*. Slices can be approximated automatically on source programs by analyzing data flow and control flow. Surveys of program slicing may be found in [2].

A decomposition slice [4] does not depend on statement numbers. It is the union of a collection of slices, which is still a program slice [8]. A decomposition slice captures all relevant computations involving a given variable and is defined as follows:

Definition 1 (Decomposition Slice) $\mathcal{DS}(v, p)$
Let
1. *$Out(p, v)$ be the set of statements in program p that output variable v,*
2. last *be the last statement of p,*
3. *N = $Out(p,v) \cup \{\text{last}\}$.*

The statements in $\mathcal{DS}(v, p) = \bigcup_{n \in N} \textbf{SLICE}_{(v,n)}(p)$ form the decomposition slice *on v.*

We take the decomposition slice for each variable in the program and form a graph,[1] using the partial ordering induced by proper subset inclusion. Larger sets will be toward the top; smaller sets will be lower; edges point downward. An edge between A and B means $B \subset A$ and there is no C, such that $B \subset C \subset A$.

Figure 1 is the decomposition slice graph of a differencing program. It has 95 nodes and 364 edges. Unravel [7] is used to compute the slices. Unravel generates a Language Independent Form (LIF) upon which a generic slicing algorithm operates. The LIF is an augmented control flow graph, and the number of nodes in the LIF is approximately the number of executable statements. A tool for Visualizing Compiler Graphs (VCG) [6] is used to display the graphs.

Figure 2 is an enlargement of a node of the graph in Figure 1. The nodes show the variable names in the form name-of-function:variable-name, where name-of-function is the function in which variable-name is declared; the annotation size: xx is the number of nodes in the LIF representation of the decomposition slice. The second parameter of $\mathcal{DS}(v, p)$ is not displayed.

The graph of decomposition slices was originally intended to give software maintainers a method for visual impact analysis [3, 5]. The decomposition slice graph provides information to a software engineer about dependencies that exist between variables in a system. The graph shows what slices for variables are included in the slices for other variables in a system. This information is useful to a software engineer in trying to gain an understanding of a system as it can be used to track the data-flow for a variable, and it can be used to identify the data that impacts on a particular variable (those variables on which it depends) and the impact of a variable (those variables which depend on it).

[1]The term "lattice" was used in [4].

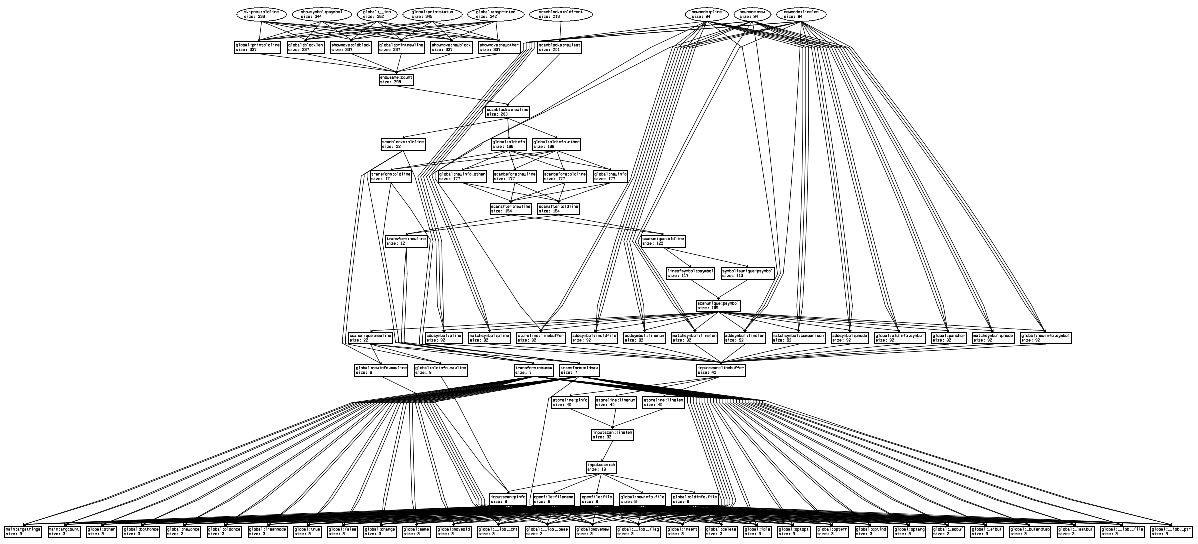


Figure 1: The decomposition slice graph of a differencing program.

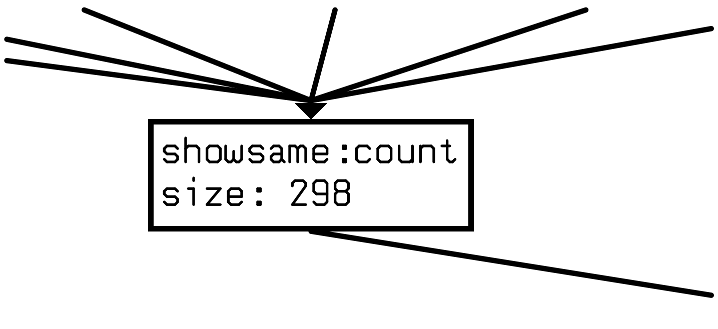


Figure 2: An enlargement of a node from the graph in Figure 1.

3 Data

Programs selected for discussion are

- `dif.c`, a differencing program whose graph was displayed previously;
- `lattice.c`, the program that computes the decomposition slices and the output graph data for input to VCG;
- `unravel.c` is the interface to the Unravel tool;
- `analyzer.c` is the interface, called by Unravel, to the program analysis portions of Unravel;
- `parser.c` is the suite of programs that do the actual parsing, analysis, and generation of the language independent form;
- `P1.c` and `P2.c` are proprietary programs that manage a database.

Table 1 summarizes the original data that we collected when analyzing these programs. Lines of code was obtained by passing the source through the Unix utility `wc`, admittedly a weak measure; however it gives an idea of the size of the systems that we analyzed. The parenthesized value next to the system name is the number of C source files that comprise the system. Column 3 is the number of nodes in the language independent form (LIF) used by Unravel. Column 4 of Table 1 lists the number of nodes in the decomposition graph computed by `lattice`, the number of decomposition slices. This is the count of the number of declared variables and the total of enumeration values declared in the system. The value includes *all* variables declared in header files; `structs` are counted as the number of fields in the struct plus 1 (for the struct itself). Column 5 is the number of edges.

Every variable or programmer defined constant (`enum` value) generated a slice. However, most of these decomposition slices were "empty"; that is, when Unravel is handed a variable to slice on, it first generates a slice with 3 nodes (from the .LIF): the entry point to the function and the open and close braces. Thus *many* slices have size 3. This is the cause of the "fan out" at the bottom of Figure 1. Table 2 shows the node and edge counts when these empty slices are removed from our samples. The percentage in column 4 is the reduction from the original count of decomposition slices. In terms of viewing the graphs, the reduction effectively removes the bottom of the decomposition slice graph. Thus, the accompanying edge clutter is

System	LOC	LIF	Nodes	Edges
dif.c (1)	767	513	95	364
lattice.c (6)	1625	685	168	1468
unravel.c (1)	803	388	482	1251
analyzer.c (1)	1287	702	817	8505
parser.c (11)	6314	4751	788	18061
P1.c (5)	2678	2493	344	7660
P2.c (7)	4539	3874	315	7972

Table 1: Analyzed Systems

	Non-empty		Reduction of
System	Nodes	Edges	Node Count
dif.c	66	161	31%
lattice.c	104	139	38%
unravel.c	281	527	42%
analyzer.c	327	2581	59%
parser.c	470	7542	40%
P1.c	267	7429	22%
P2.c	279	7864	11%

Table 2: Node and edge counts after removing “empty” decomposition slices

System	Reduced Nodes	Reduced Edges	Reduction of Node Count
dif.c	34	43	62%
lattice.c	83	104	51%
unravel.c	129	161	73%
analyzer.c	198	230	76%
parser.c	201	238	75%
P1.c	76	89	78%
P2.c	61	67	81%

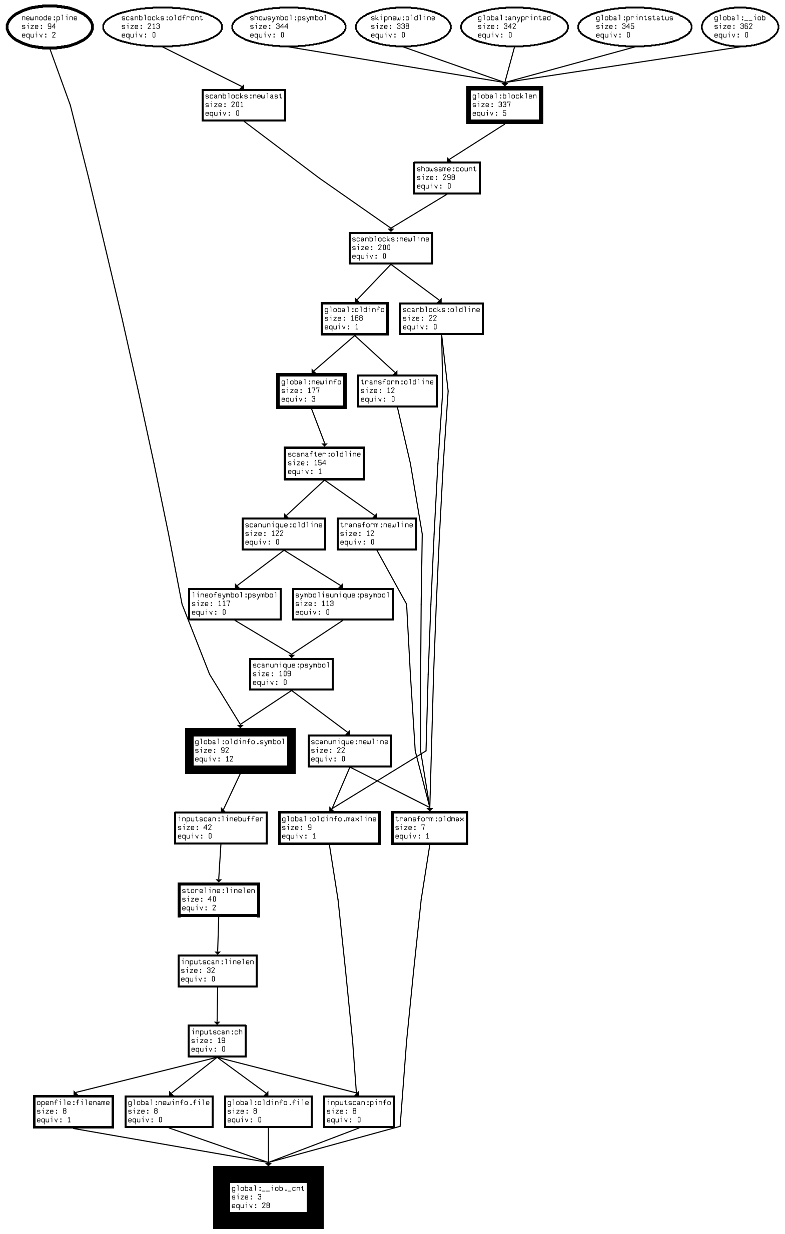
Table 3: Reduction by equivalent slices, including "empty" slices

removed. The graph that results when this reduction is applied to the graph of Figure 1 is not shown.

Figure 3 shows the result of reducing the graph of Figure 1. The reduced graph of Figure 3 has 34 nodes and 43 edges. The "empty" slices are not removed, but are collapsed into the single node with the wide border at the bottom of the figure. We also augmented the node information to show how many other nodes are equivalent with the annotation `equiv: x`. The border of the node was also widened in proportion to the number of equivalent nodes. Figure 4 shows a sample node.

Due to the vagaries of VCG layout algorithms, the reduced graph is rotated about the vertical axis with respect to graph of Figure 1. The five upper leftmost nodes of Figure 1 are the five upper rightmost nodes of Figure 3. The three nodes to the upper right of Figure 1 are collapsed to the single node in the upper left of Figure 3. Following the edges from these nodes downward in Figure 1 leads to the "fan-out" in the lower center of the figure. This fan-out of 14 nodes is reduced to *two* nodes in Figure 3; and the node reduction induces a drastic reduction in the number of edges.

Table 3 shows the results of applying the reduction to the selected sample programs. The counts include the "empty" slices. The precipitous drop in the number of edges is a side-effect of the reduction of the number of nodes.



newnode:pline size: 94 equiv: 2
scanblocks:oldfront size: 213 equiv: 0
showsymbol:psymbol size: 344 equiv: 0
skipnew:oldline size: 338 equiv: 0
global:anyprinted size: 342 equiv: 0
global:printstatus size: 345 equiv: 0
global:__iob size: 362 equiv: 0
scanblocks:newlast size: 201 equiv: 0
global:blocklen size: 337 equiv: 5
showsame:count size: 298 equiv: 0
scanblocks:newline size: 200 equiv: 0
global:oldinfo size: 188 equiv: 1
scanblocks:oldline size: 22 equiv: 0
global:newinfo size: 177 equiv: 3
transform:oldline size: 12 equiv: 0
scanafter:oldline size: 154 equiv: 1
scanunique:oldline size: 122 equiv: 0
transform:newline size: 12 equiv: 0
lineofsymbol:psymbol size: 117 equiv: 0
symbolisunique:psymbol size: 113 equiv: 0
scanunique:psymbol size: 109 equiv: 0
global:oldinfo.symbol size: 92 equiv: 12
scanunique:newline size: 22 equiv: 0
inputscan:linebuffer size: 42 equiv: 0
global:oldinfo.maxline size: 9 equiv: 1
transform:oldmax size: 7 equiv: 1
storeline:linelen size: 40 equiv: 2
inputscan:linelen size: 32 equiv: 0
inputscan:ch size: 19 equiv: 0
openfile:filename size: 8 equiv: 1
global:newinfo.file size: 8 equiv: 0
global:oldinfo.file size: 8 equiv: 0
inputscan:pinfo size: 8 equiv: 0
global:__iob._cnt size: 3 equiv: 28

Figure 3: The reduced decomposition slice graph of Figure 1.

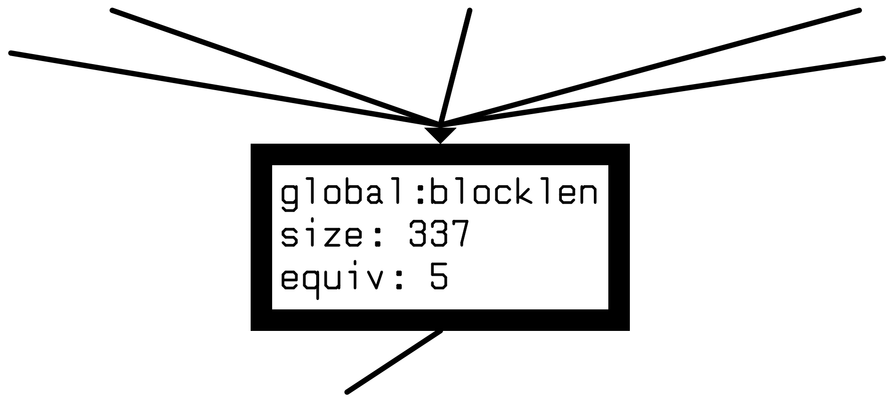


Figure 4: A node from the reduced graph.

4 Discussion

The reductions are significant. In removing empty slices, in the worst case improvement was an 11% reduction in the number of nodes; the best was nearly 60%. The large variance in these values is determined by the number of included library files. `P2.c` has only 2 included files, `string.h`, and one declared by the programmer. `analyzer.c` uses the X-windowing libraries, with its myriad of `struct`s and `typedef`s.

In reducing by equivalent slices, the worst improvement was still over 50%; the best over 80%. This turns a graph of 279 nodes and 7864 edges into one with 61 nodes and 67 edges. The second graph is comprehensible; the first is not. Even in the worst case, `lattice.c`, after the removing the empty slices, some 20% of the variables yielded the same decomposition slice.

The data base programs, `P1.c` and `P2.c`, had a reduced graph structure markedly different from the other systems. [AUTHOR'S NOTE: I'll bring the pictures!] They were very wide at the top and quickly "fanned-in" to a single large node, which was the decomposition slice on some 200 variables! This large node had a few more levels below it. (It looks like the letter "Y", but with many arms extending upward from the join.) We are not sure of everything that this means; our best current guess is that the data base is embedded in these computations and that the other computations compute with the data. One thing we are sure of is that this reduction has a drastic impact on system testing. Since, in this instance, some 200 variables induce the same computation, any testing criterion that is used to validate one of

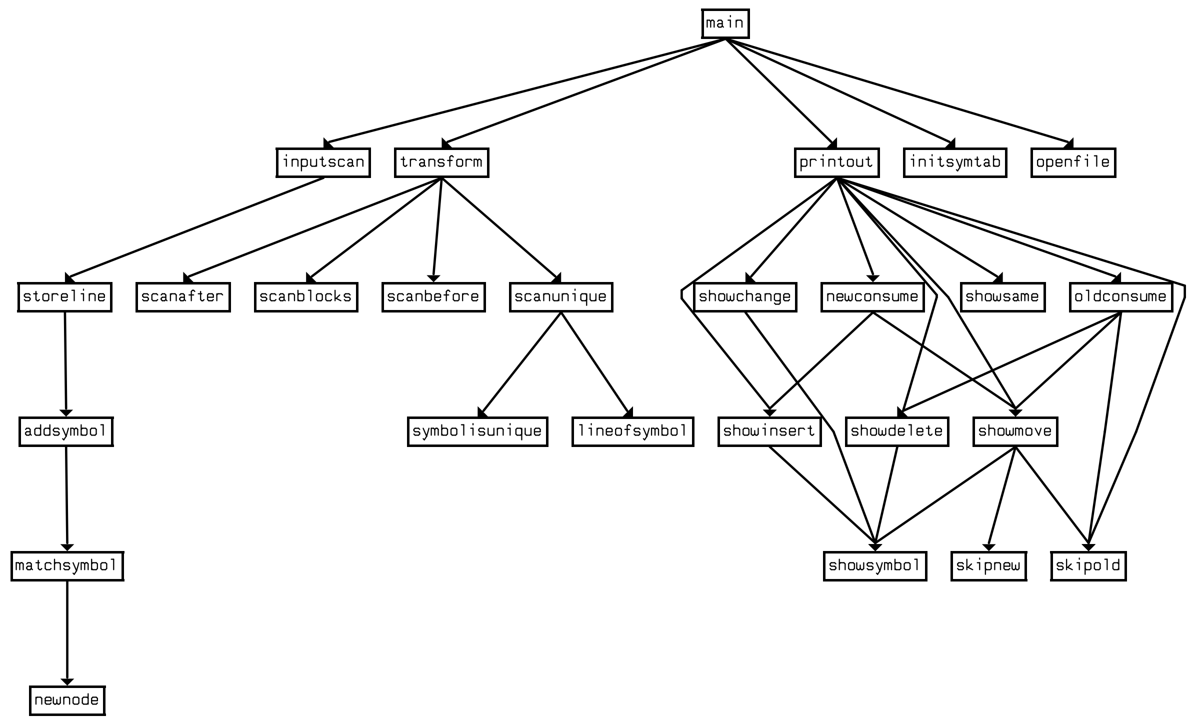


Figure 5: The call-graph of the differencing program.

these variables would meet all the rest. This may turn out to be the deepest insight of all.

Figure 5 shows the call graph of the differencing program. The program describes itself as "pedagogic," i.e., "an example of one professional's style of keeping things organized and maintainable." It has a simple structure: two initialization functions; an input, a transformation and an output function. What is lost in the call graph and shown by the decomposition slice graph is the fact that the functions `scanblocks` and `showmove` have some 200 LIF nodes in common! This means that approximately a *common* 40% of the program is used in the evaluation of each of these functions. The decomposition slice graph shows how the *computations* are interleaved. We presume that this is useful information to an engineer contemplating a change to either of these functions.

5 Conclusion and Future Directions

We are continuing to analyze programs and examine the decomposition slice graphs. The analysis itself has generated more questions than it has answered. The most important seems to be: "Why do so many different variables yield the same decomposition slice?" We are currently studying this. The answer to this requires a lot of old-fashioned code reading. Once we actually ascertain why so many decomposition slices are the same we can generate a hypothesis for an experiment.

The attempts to answer these questions have generated hypotheses for other empirical studies.

1. Does this graph actually assist the comprehension process? If so, how? We are currently evaluating this. The graphs tell us what decomposition slice(s) to examine. Is there some other application of this in the comprehension process?
2. Does the structure of the graph yield insights about the architecture of the system?
3. Why all those empty slices?
4. Do the properties of these graphs yield metrics? That is, is "wide and shallow" or "narrow and deep", or some combination thereof a good or bad property? There is a natural relation to Beiman and Ott's [1] work on slice-based measures of cohesion that needs to be explored.

We also have a system that has gone through 18 releases; we are analyzing these in turn to observe the change, if any, in the graphs pass through in the evolution of this system.

Other ways to improve the graphical presentation are:

1. List all node names that have been collapsed into a single reduced node, rather than showing only the count.
2. Remove variables included from libraries, while leaving programmer defined ones.
3. Visualize the non-empty intersections in the partial order. The decomposition graph shows the is-contained-in partial order; however, items can have a non-empty intersection and not be in the partial order. We would like to display this.

A software engineer is presented with an overwhelming amount of information in attempting to understand a program. Easily computable reductions by equivalent information is one way to reduce this clutter. By using set equivalence, we can reduce the complexity of graphs of decomposition slices. The significant size of the reduction provides a maintainer/comprehender with a valuable tool for impact analysis.

Authors' notes to reviewers

The bibliography is brief for two reasons.

1. This is a workshop paper, reporting a work-in-progress.
2. These *are* the only works used in this work. The paper has only a "background" section, intended to make the paper self-contained. It does *not* have a "related work" section in which would contain a thorough literature survey.

Yes, I (kbg) am aware of the current work in program slicing.

References

[1] J. Bieman and L. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, August 1994.

[2] D. Binkley and K. Gallagher. A survey of program slicing. In M. Zelkowitz, editor, *Advances in Computers.* Academic Press, 1996.

[3] K. B. Gallagher. Visual impact analysis. In *Proceedings of the Conference on Software Maintenance - 1996*, 1996.

[4] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[5] K. B. Gallagher and L. O'Brien. Reducing visualization complexity using decomposition slices. In *Proceedings of the 1997 Software Visualization Workshop, SoftVis97*, number ISBN 0725806303, Dec 1997.

[6] I. Lemke and G. Sander. *Visualization of Compiler Graphs: Design Report and Documentation.* Universitat des Saarlandes, Saarbrucken, Germany, May 1994. VCG.

[7] J.R. Lyle, D.R. Wallace, J.R. Graham, K.B. Gallagher, J.E. Poole, and D.W. Binkley. *A CASE tool to evaluate functional diversity in high integrity software.* U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD, 1995. http://hissa.ncsl.nist.gov/~ jimmy/unravel.html.

[8] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.