Dependence Clusters in Source Code

MARK HARMAN
King's College London
DAVID BINKLEY
Loyola College
KEITH GALLAGHER
Durham University
NICOLAS GOLD
King's College London
and
JENS KRINKE
King's College London

A dependence cluster is a set of program statements, all of which are mutually inter-dependent. This paper reports a large scale empirical study of dependence clusters in C program source code. The study reveals that large dependence clusters are surprisingly commonplace. Most of the 45 programs studied have clusters of dependence that consume more than 10% of the whole program. Some even have clusters consuming 80% or more. The widespread existence of clusters has implications for source code analyses such as program comprehension, software maintenance, software testing, reverse engineering, reuse, and parallelization.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.8 [**Software Engineering**]: Metrics—*Complexity measures*

General Terms: Algorithms, Languages, Measurement

Additional Key Words and Phrases: Dependence, program comprehension, program slicing

Authors' Addresses: Mark Harman, King's College London, Strand, London, WC2R 2LS, UK. David Binkley, Loyola College in Maryland, Baltimore, Maryland, 21210-2699, USA. Keith Gallagher, Durham University, South Road, Durham, DH1 3LE, UK. Nicolas Gold and Jens Krinke King's College London, Strand, London, WC2R 2LS, UK.
This research is supported by EPSRC grant GR/F010443.

1. INTRODUCTION

Program dependence analysis explores the dependence relationships between program statements. This statement-level dependence analysis is the cornerstone of many activities that rely upon program analysis, such as program comprehension [De Lucia et al. 1996; Korel and Rilling 1997; 1998; Harman et al. 2003; Binkley et al. 2000; Ning et al. 1994], procedure extraction [Komondoor and Horwitz 2000; Harman et al. 2004], clone detection [Gallagher and Layman 2003], visualization [Gallagher and O'Brien 2001], impact analysis and reduction [Black 2001; Gallagher et al. 2003; Tonella 2003], reuse [Cimitile et al. 1995; 1996], software measurement [Bieman and Ott 1994; Hall et al. 2005; Lakhotia 1993; Yau and Collofello 1985], software maintenance [Gallagher and Lyle 1991], testing and debugging [Bates and Horwitz 1993; Binkley 1997; Gallagher and Binkley 2003; Gupta et al. 1992; Harman et al. 2004], virus detection [Lakhotia and Singh 2003], validation [Krinke and Snelting 1998], integration [Binkley et al. 1995; Horwitz et al. 1989], and restructuring, reverse engineering and reuse [Beck and Eichmann 1993; Canfora et al. 1998; Canfora et al. 1994; Jackson and Rollins 1994; Lakhotia and Deprez 1998].

Since program dependence is essentially a relation on program statements it is typically represented as a graph. This 'graph nature' of dependence raises a natural question as to whether large connected components are found in real-world programs. In this paper such clusters of interdependent statements are termed *dependence clusters*, because they denote clusters of program components that all mutually depend upon one another.

At higher levels of abstraction, such as modules and functions, clustering has been considered important for the evolution of good software architecture and so these higher-level dependence clusters have been widely studied [Harman et al. 2005; Mahdavi et al. 2003; Gallagher and Binkley 2003; Mitchell and Mancoridis 2002; 2006]. However, despite the relative importance of statement-level dependence, there have been no previous studies of them. This paper presents an empirical analysis of statement-level dependence clusters, providing evidence that such dependence clusters are surprisingly widespread.

A dependence cluster (hereinafter, the term *dependence cluster* refers to a statement-level dependence cluster) is formally defined in the next section as the solution to a reachability problem. This definition is instantiated using reachability over a program's System Dependence Graph (SDG) [Horwitz et al. 1990].

As a simple illustrative example of a dependence cluster, consider the example in Figure 1. In this example, the predicate $\mathsf{i} < \mathsf{10}$ data depends on the assignment to i, this assignment control depends on the predicate of the if statement, and the if control depends on the predicate $\mathsf{i} < \mathsf{10}$. As a result, all three statements are mutually inter-dependence; they form a dependence cluster. Any change to any one of the three will have a potential effect on the others.

The paper examines the prevalence of dependence clusters, revealing them to be surprisingly large and widespread in the code studied. The paper considers both forward and backward dependence and examines some of the causes and implications of the prevalence of large dependence clusters.

The paper uses an approach that approximates whether two or more statements (or, more precisely, dependence graph nodes) are in a dependence cluster by check-

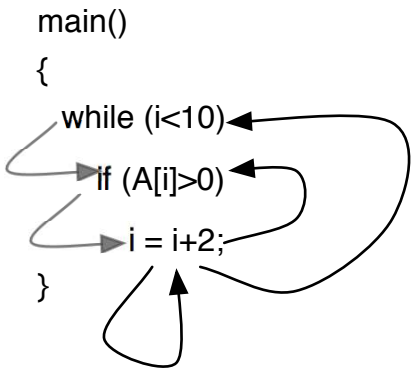

Fig. 1. A cluster caused by data (black) dependences and control (grey) dependences.

ing to see if the sizes of their slices [Binkley and Gallagher 1996] are identical. This 'same size slice' approach is a conservative (and therefore safe) approximation to the true dependence cluster relation; it may produce false positives, but never false negatives. Results are visualized with the aid of the *Monotone Slice-size Graph* (MSG) whose construction is made possible using massive slicing optimizations to the basic SDG-based slicing algorithm [Binkley et al. 2007]. A group of slices of similar size appears as a "plateau" in the monotonically increasing profile of slice sizes in the MSG.

The paper presents results from two empirical studies[1]. These are designed to evaluate the concept of a dependence cluster. One of these studies provides verification, while the other is concerned with validation. The verification question addressed is

> "How precise is the approximation which underpins the MSG?"

Verification is concerned with whether the approach works. Since the approach is a conservative approximation, capable of yielding false positives, it is important to gauge how often these false positives occur in practice. If they are too frequent then the approach is not viable. For very small slices, it is expected that two slices could have the same size and yet be different. However, it is just too much of a coincidence to find two large slices of the same or very similar size but entirely different content. The verification study bears out this informal observation. It shows that, when the slices in dependence clusters are considered, for 99.5% of clusters, the slices in these clusters are all 99+% identical.

The validation question is

> "How common are large dependence clusters?"

Validation is concerned with whether large dependence clusters exist in real programs (making dependence cluster analysis a valid course of study). Of course,

[1]An earlier version of these studies was presented by Binkley and Harman [2005b]. The present paper extends the empirical study to consider more than twice as many programs, presenting results in greater detail and with extended discussion of the findings and their implications.

what constitutes a *large* dependence cluster depends upon the definition of 'large'. The validation study shows that there are considerable numbers of large clusters for all reasonable definitions of *large*. For example, defining a large dependence cluster to contain 10% or more of the program's statements reveals that large dependence clusters are surprisingly common: 40 out of 45 programs studied (totaling over 1.2 million lines of source code) contained one or more dependence clusters of at least 10% of the program.

In both the verification and validation studies, program points whose slices contain fewer than 1% of the nodes of the program's System Dependence Graph are ignored as they are not likely to be of interest in any application of program slicing. This prevents these slices from biasing the results as all programs include such trivial slices. The most common source is the declaration of global variables, including those found in standard library header files, whose slices include no other part of the program other than the declaration itself.

The removal of these slices cannot increase the number of large dependence clusters that are identified by the study. That is, the width of a plateau on the MSG cannot be affected (by definition) so a large cluster will still be large. The number of dependence clusters can therefore be reduced (though it cannot be increased). However, all the additional slices would be extremely small (less than 1% of the program by definition) and therefore not interesting.

Overall, the findings of this paper suggest that dependence clusters are worthy of further study. The paper shows that they are easy to define, to locate, and to investigate, as they occur frequently in real programs. The study also provides evidence to suggest that the MSG visualisation is helpful in analyzing dependence clusters. The paper makes the following primary contributions:

(1) It presents the results of an empirical study into the applicability of the MSG as a technique for identifying dependence clusters (Section 4). The slices of 45 programs were analyzed showing that the MSG approximation is extremely accurate.

(2) It investigates the prevalence of large dependence clusters, presenting results that indicate that such clusters are very common (Section 5).

(3) It investigates one possible cause of dependence clusters, showing that individual predicate nodes can sometimes lead to large clusters, and uses this analysis to show how it is sometimes possible to remove a cluster (Section 6).

(4) It presents results for forward dependence cluster analysis and draws a connection between this analysis and work on static impact analysis (Sections 7 and 8).

These findings verify the approximation approach and validate the study of dependence clusters. The presence of large dependence clusters has widespread implications. The paper discusses the impact of this finding on related source-code analysis work such as that on program comprehension, testing, maintenance, reuse, and automatic parallelization.

The remainder of the paper is organized as follows: Sections 2 and 3 provide formal definitions and describe the experimental setup used for the verification and validation studies. Sections 4 and 5 present the results of the two empirical stud-

ies concerned with verification and validation. The verification study shows that Monotone Slice-size Graphs (MSGs) provide a very good approximation, while the validation study shows that large dependence clusters are quite common in real-world source code. Section 6 reports results that investigate on possible cause of dependence clusters. Up to this point in the paper, the dependence clusters considered have been those present for backward dependence analysis. Section 7 presents results that indicate that large forward dependence clusters are also prevalent in the programs under investigation, while Section 8 shows how the presence of forward dependence clusters can dramatically increase the impact of a software change. Section 9 considers threats to the validity of the results and Section 10 discusses related work. Finally, Section 11 summarizes the paper.

2. DEFINITIONS

This section sets out the definitions of dependence cluster and the Monotone Slice-size Graph (MSG) used in the remainder of the paper. Although the empirical study is based on an instantiation of these definitions using slices of the System Dependence Graph (SDG) [Horwitz et al. 1990], the notion of dependence cluster is not necessarily slice-based; thus, the results of the theory presented in this section can be applied beyond slicing.

2.1 Dependence Clusters

A (statement level) *dependence cluster* is a set of program statements, S, that mutually *depend* upon one another and for which there is no other mutually dependent set that contains S. This definition is parameterized by an underlying *depends-on* relation. Herein *statements* are considered to be *source-code representing* nodes of the SDG. This excludes from analysis *pseudo* nodes introduced, for example, to represent global variables that are modeled as additional pseudo parameters by CodeSurfer [Grammatech Inc. 2002], the tool used to build the SDGs. Ignoring such nodes is merely a convenience as it helps tie the result back to the program source code.

Definition 1. *Mutually-Dependent Set*
A *Mutually-Dependent Set (MDS)* is a set of statements, $\{s_1, \ldots, s_m\}$ $(m > 1)$, such that for all i, j, $1 \leq i, j \leq m$ s_i depends on s_j.

A dependence cluster is simply a maximal set of mutually dependent points.

Definition 2. *Dependence Cluster*
A *dependence cluster* is an MDS not properly contained within any other MDS.

In the SDG, if the slice constructed for slicing criterion n_1 contains n_2 and the slice constructed for criterion n_2 contains n_1, then by construction there is a path of dependence edges from each node in $\{n_1, n_2\}$ to the other. That is, the two nodes are transitively mutually dependent. This observation can be used to formulate a slice-based notion of dependence cluster. In the definition, $Slice(n)$ is used to denote the set of SDG nodes in the static backward slice taken with respect to node n of the SDG [Horwitz et al. 1990].

Definition 3. *Slice-Based Mutually-Dependent Set*
A *Slice-Based MDS* is a set of SDG nodes, $\{n_1, \ldots, n_m\}$ $(m > 1)$, such that for all i, j, $1 \leq i, j \leq m$ $n_i \in Slice(n_j)$.

By the definition of (static backward) slicing, this means that all nodes in a slice-based MDS depend upon all others in the MDS (including themselves), so $\{n_1, \ldots, n_m\}$ satisfies Definition 1.

Definition 4. *Slice-Based Dependence Cluster*
A *Slice-Based Dependence Cluster* is a slice-based MDS contained within no other slice-based MDS.

This paper focuses entirely upon slice-based dependence clusters. That is, the clusters considered in this paper arise from the data and control dependence that exists between SDG nodes. This form of dependence has been widely studied and (as shown in Section 10) the dependence clusters that can be identified using slice-based dependence have far-reaching implications for many source code analyses.

The majority of the paper considers backward dependence using static backward slicing. However, Section 7 considers, more briefly, forward dependence, showing that forward dependence clusters are also highly prevalent in the programs studied and drawing out relationships to work on static impact analysis.

2.2 Monotone Slice-size Graphs

The Monotone Slice Size Graph (MSG) visualisation plots a landscape of monotonically increasing slice size, in which dependence clusters correspond to sheer-drop cliff faces followed by a plateau. The goal of the visualisation is to assist with the inherently subjective task of deciding whether a cluster is large (how long is the plateau at the top of the cliff face relative to the surrounding landscape?) and whether it denotes a discontinuity in the dependence profile (how steep is the cliff face relative to the surrounding landscape?).

Definition 5. *Monotone Slice-size Graph*
A *Monotone Slice-size Graph* is a graph of slice sizes, plotted for monotonically increasing size. That is, slices are ordered according to their size, and the slice sizes plotted in ascending order.

For example, consider the program copia in Figure 2, which will be used as a running example throughout this paper. The program implements a collection of analyses on an input table. As can be seen from Figure 2, the MSG contains a large plateau of slices which appear to have the same size; certainly a large dependence cluster. However, zooming in on the plateau in the MSG reveals that this single plateau actually consists of 15 smaller plateaus. The first 5 of these summarize over 99% of the slices that make up the 'single' plateau and differ by no more than 4 vertices (about 0.27% of the program). This observation provides evidence for the robustness of the MSG visualisation; although the slices are not of identical size, they are all closely related. The interpretation of the visualisation is correct; there is a large dependence cluster.

In this paper the MSG will be used to visualize the dependence structure of programs and the approximation, 'same slice size', will be used to stand in for

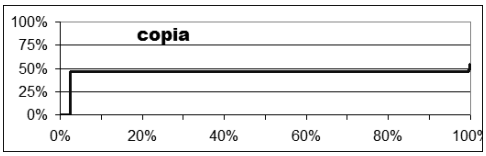

Fig. 2. The MSG of the Program copia

'same slice'. For some of the larger programs studied, the computation of 'same slice' is unrealistically expensive. Fortunately, as will be seen the computation of same size is much faster and the approximation it denotes will be shown to be very close to the computation for 'same slice' (using a subset of the programs studied for which the values of 'same slice' are computable in reasonable time).

To further motivate the use of slice-size comparisons as a proxy for slice-content comparisons, consider the complexity of the two. It is possible to locate dependence clusters by comparing all the slices of a program to see which slices are identical. Pseudo code annotated with complexity for this approach is shown in Figure 3. For an SDG of n nodes and e edges, slicing is a linear operation in the number of edges; thus, computing the slice for all statements (nodes) of a program can be done in $O(ne)$ time. Comparing two slices takes $O(n\log(n)+n) = O(n\log(n))$ steps to first sort the vertices of the two slices and then compare the sorted lists. Pre-sorting the nodes of all of the $O(n)$ slices, takes $O(n \times n\log(n))$ steps. Subsequently, the $O(n^2)$ pair-wise comparisons each take $O(n)$ time; thus, the partitioning the slices into equivalence classes using that algorithm shown in Figure 3 requires $O(n^3)$ time.

In principle, it is possible to reduce this to $O(n^2\log(n))$. For example, by numbering the vertices of the SDG, each slice can be identified by a single binary number obtained by writing down in sorted order its vertex numbers. Building a trie from numbers places identical slices at the same leaf. The length of the single binary number for each vertex is $O(n\log(n))$; thus, to compare all n slices would take $O(n^2\log(n))$ steps. However, for large programs this cost grows prohibitive. It also sacrifices the ability to easily identify slices that differ by a small number of vertices.

PartitionSlices(Program P)	Complexity
foreach vertex, v	$O(ne) == O(n^3)$
s_v = slice(P, v)	$O(e)$
S = set of slices s_v	
foreach slice, s in S	$O(n \times n\log(n))$
s = sort(s)	$O(n\log(n))$
foreach i in $1 \cdots \|S\|$	$O(n^3)$
foreach j in i+1 $\cdots ... \|S\|$	$O(n^2)$
compare si and sj	$O(n)$

Fig. 3. Slice Comparison Algorithm

However, an approximation for *same-slice* can be used that is considerably more efficient. The approximation uses *slice size* in place of the actual nodes of the slice. Rather than checking to see if two nodes yield identical slices, the approach simply checks if the two yield slices of the same size. This slice-size approach is inherently more efficient than comparing slice content: post-slicing to tabulate the number of slices with each possible size requires only $O(n)$ steps to initialize and then increment a counter for each size.

The conjecture that underpins this approximation is that two slices of the same (sufficiently large) size are likely to be the same slice. Clearly, this approximation is conservative because any cluster identified may contain real clusters and no real clusters will fail to be identified. That is, two slices may differ yet, coincidentally, may have identical sizes; however, two slices which are identical must clearly have the same size. The verification study in Section 4 directly addresses the question of the quality of this approximation, which is not only more efficient, but also justifies the use of the MSG as a visualisation for identifying clusters.

The MSG is not only efficient to compute, it also helps with the essentially subjective task of determining whether a cluster is large, relative to the code that contains it. As an example, consider again the MSG for userv. The sharp increase in slice size that occurs after about 44% of slices have been considered is followed by a long plateau in which slice size does not change. The length of the plateau indicates a large cluster of slices of identical size; in other words, a large dependence cluster.

As discussed in the next section, many otherwise identical slices often differ by a small number of statements. For example, consider the slices on the two computations that compute the sum and average of an array of numbers. The two are identical except for the statement average = sum / N. An advantage of the lack of sharpness [Binkley et al. 2006] in the MSG is that such differences are hidden (below the resolution of the visualisation). Thus, the dominant features of the dependence landscape are more readily identified.

3. EXPERIMENTAL CONFIGURATION

The identification of large dependence clusters, as well as the empirical studies contained within this paper, require the computation of massive numbers of program slices. This section provides background on program slicing and the techniques used to compute such a large number of slices. It also introduces the programs studied and the environment in which the data was collected.

In order to compute large numbers of slices a variation of the SDG slicing algorithm [Horwitz et al. 1990] was used. The variation is specifically constructed to cater for massive slicing, using a series of graph-based optimizations [Binkley et al. 2007]. Two main techniques are employed. The first identifies intraprocedural strongly-connected components (SCCs) and replaces them with a single representative vertex. The key observation here is that any slice that includes a vertex from an SCC will include all the vertices from that SCC; thus, there is great potential for saving effort by avoiding redundant work. The second technique removes redundant transitive edges from the graph. Such edges never lead to the discovery of

new nodes for a slice, but still must be examined during the computation of each slice.

We also included a space and time saving efficiency step in the implementation. That is, we compute slice differences incrementally, sharing pointers to subslices that are common to many. This produces a noticeable speed up and reduces space costs, allowing us to cover larger programs, without affecting the results in any way. Using Binary Decision Diagrams (BDDs) would also reduce space costs, but would increase computational costs, because our current representation of slices as bitstrings is optimized for computation (not space). For this reason we chose not to use BDDs for these experiments.

The study considers 45 C programs, mostly open source, with some industrial programs from the European Space Agency, that range in size from 600 LoC to almost 180 KLoC. The programs cover a range of application domains such as utilities, games, and operating system code. Figure 4 provides a brief description of each program and shows two measures of each program's size: lines of code (as counted by the UNIX utility wc) and the non-comment non-blank lines of code (as counted by the utility sloc [Wheeler 2005]). In total, just over 1.2 million lines of code (895 thousand non-comment, non-blank lines of code) were studied.

4. EMPIRICAL VERIFICATION: HOW PRECISE IS THE DEPENDENCE CLUSTER DETECTION?

This section presents the results of an experiment into whether similarity in slice size can be used as a proxy for similarity of slice content. The experiment seeks to provide evidence to support the claim that MSGs are a suitable and reliable technique for finding dependence clusters. That is, the research question to be answered is whether a set of slices that have the same size will tend to have the same or nearly the same vertices. Of course, the answer will depend upon the interpretation given to 'nearly the same'. This will be referred to as *similarity*; the degree to which two slices can differ in content while being deemed to be essentially the same.

Figure 5 investigates the degree to which agreement is improved by admitting a small amount of difference, or put another way, reducing the level of similarity required for agreement. It plots the similarity (on the horizontal axis) against the agreement between slice size and slice content (on the vertical axis) for thirty six of the programs studied (maintaining all slices in memory for comparison with some of the larger programs would require tens of gigabytes of memory and thus these program were omitted from the verification study). Jaccard index, Both axes in Figure 5 are represented as percentages. Here, a similarity is the Jaccard index of a slice where a similarity of $x\%$ means that the percentage of nodes in common (the intersection) is $x\%$ of the total number of nodes (i.e., the nodes in either slice (the union)); thus, it is possible to speak of slices being 'similar up to a certain point.' For a given value of similarity, $x\%$, an agreement of $y\%$ means that $y\%$ of the total number of slices are $x\%$ similar.

The figure shows the minimum, average, and maximum agreement starting at the far right with 100% similarity (i.e., equal slices). As can be seen in the figure, almost total agreement is reached for most programs with a very high similarity: the

Program	LoC	SLoC	Vertices	Edges	Slices	Brief Description
a2ps	63,600	40,222	707,623	1,488,328	58,281	ASCII to Postscript
acct-6.3	10,182	6,764	21,365	41,795	7,250	Process monitoring utilities
barcode	5,926	3,975	13,424	35,919	3,909	Barcode generator
bc	16,763	11,173	20,917	65,084	5,133	Calculator
byacc	6,626	5,501	41,075	80,410	10,151	Parser Generator
cadp	12,930	10,620	45,495	122,792	15,672	Protocol toolbox
compress	1,937	1,431	5,561	13,311	1,085	File Compressor
copia	1,170	1,112	43,975	128,116	4,686	ESA signal processing code
cook	49,026	34,870	4,754,735	13,010,556	49,027	file construction tool
csurf-pkgs	66,109	38,507	564,677	1,821,811	43,044	C ADT library
ctags	18,663	14,298	188,856	405,383	20,578	C tagging
cvs	101,306	67,828	8,949,186	28,033,287	103,265	Revision Control
diffutils	19,811	12,705	52,132	104,252	17,092	File differencing
ed	13,579	9,046	69,791	108,470	16,533	Line text editor
empire	58,539	48,800	1,071,321	2,122,627	120,246	Conquest Game
EPWIC-1	9,597	5,719	26,734	56,068	12,492	Wavelet image encoder
espresso	22,050	21,780	157,828	420,576	29,362	Logic simplification for CAD
findutils	18,558	11,843	38,033	174,162	14,445	File finding utilities
flex2-4-7	15,813	10,654	49,580	105,954	11,104	Lexical Analyzer Builder
flex2-5-4	21,543	15,283	55,161	234,024	14,114	Lexical Analyzer Builder
ftpd	19,470	15,361	72,906	138,630	25,018	File Transfer
gcc.cpp	6,399	5,731	26,886	96,316	7,460	GCC's pre processor
gnubg-0.0	10,316	6,988	36,023	104,711	9,556	Gnu Backgammon
gnuchess	17,775	14,584	56,265	165,933	15,069	Chess player
gnugo	81,652	68,301	396,010	1,087,038	68,298	Go player
go	29,246	25,665	144,299	321,015	35,863	Go player
ijpeg	30,505	18,585	289,758	822,198	24,029	JPEG compressor
indent	6,724	4,834	23,558	107,446	6,748	Text formatter
li	7,597	4,888	1,031,873	3,290,889	13,691	XLisp interpreter
named	89,271	61,533	1,853,231	8,334,948	106,828	DNS lookup
ntpd	47,936	30,773	285,464	1,160,625	40,199	Time Daemon
oracolo2	14,864	8,333	27,494	76,085	11,812	Antennae array set-up
prepro	14,814	8,334	27,415	75,901	11,745	ESA array pre-processor
replace	563	512	1,406	2,177	867	Regular expression
sendmail	46,873	31,491	1,398,832	10,148,436	47,344	mail processor
snns	79,170	52,798	2,140,672	4,673,668	79,178	neural network analyzer
space	9,564	6,200	26,841	74,690	11,277	ESA ADL interpreter
spice	179,623	136,182	1,713,251	6,070,256	212,621	Digital circuit simulator
termutils	7,006	4,908	10,382	23,866	3,113	Unix terminal emulation
tile-forth-2.1	4,510	2,986	90,135	365,467	12,076	Forth Environment
time-1.7	6,965	4,185	4,943	12,315	1,044	CPU resource measure
userv	8,009	6,132	71,856	192,649	12,517	Access control utility
wdiff.0.5	6,256	4,112	8,291	17,095	2,421	Diff front end
which	5,407	3,618	5,247	12,015	1,163	Unix utility
wpst	20,499	13,438	140,084	382,603	20,889	Pointer Analysis hline
sum	1,284,742	912,603	26,760,591	86,329,897	1,338,295	
average	28,550	20,280	594,680	1,918,442	29,740	

Fig. 4. The 45 subject programs studied.

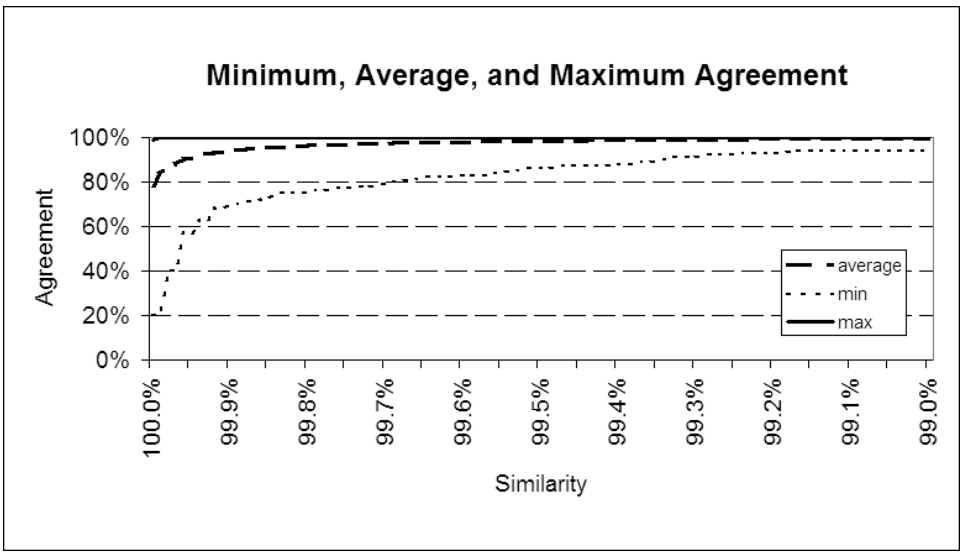

Fig. 5. Agreement Levels down to 99% Similarity

horizontal axis only goes down to 99% similarity, so all the data shown in Figure 5 concern slices within 1% of containing identical nodes. Along the y-axis (i.e., 100% similarity) agreement ranges from 16% to 99%; thus, that same size equals same slice is not unanimous when zero difference is permitted. However, requiring the vertices in the slices to agree by 'only' 99.98%, this range shrinks to 40% to 100% and by 99.9% it shrinks to 76% to 100%. Perhaps of more interest in how quickly the average line crosses the 99% threshold: by 99.55% (within less than one half of one percent of an exact match). Thus, from Figure 5, it is clear that same size is a good proxy for same slice.

In total, 99.5% of the clusters are represented on this graph. That is, 99.5% of clusters are within 1% of total agreement. If the figure were to be redrawn, with a horizontal axis extended all the way down to 0%, then the detail would be completely lost, because almost all programs would immediately reach 100% agreement on the vertical axis.

Of course, there are a few programs where some slices simply *happen* to be the same size, but contain completely different sets of nodes. This should be expected in any suitably large data set. To get a sense for just how common this occurrence is, consider the data presented in Figure 6. This figure shows all the data for which a similarity of less than 99% is required for 100% agreement. The horizontal axis shows each of the thirty six programs studied. The vertical axis shows the percentage of same-size slices that are less than 99% similar. As the figure shows, for over a third of the programs, there are simply *no* slices of the same size that require less than 99% similarity to agree 100%. Over all thirty six programs studied, only 0.58% of the clusters required a similarity of less than 99% in order to achieve 100% agreement.

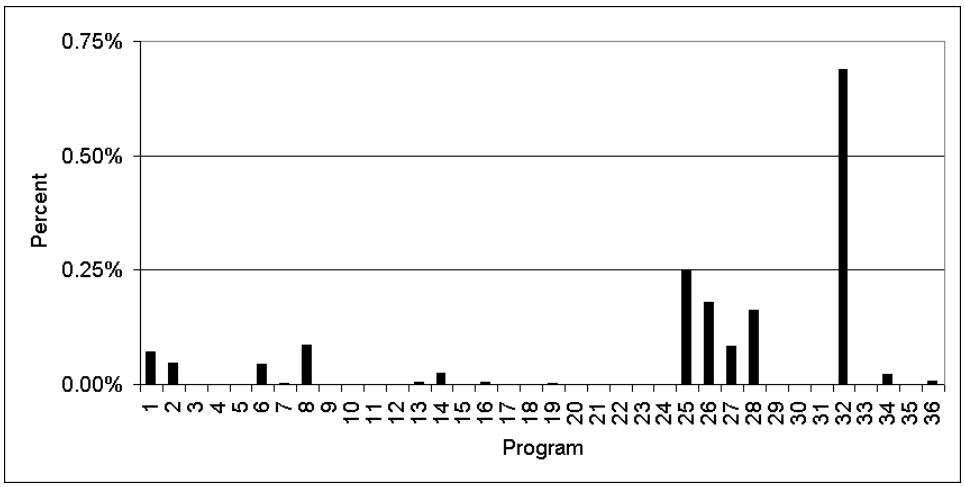

Fig. 6. Sparsity of high tolerance

Furthermore, even this low figure of 0.58% is perhaps unduly pessimistic because it records the number of *clusters* which require less than 99% similarity. However, even in such clusters, many of the individual slices in the cluster, may fully agree. The figure for the number of pairwise slice comparisons which fail to agree within 99% similarity is only 0.00572%. These results provide strong evidence for the claim that 'slice size agreement' is a good approximation for 'slice content agreement' and thus for locating dependence clusters using MSGs.

5. EMPIRICAL VALIDATION: DO LARGE DEPENDENCE CLUSTERS OCCUR IN PRACTICE?

This section considers the validation question—first visually and then quantitatively. Two visualizations are used: first the MSG and second size-distribution graphs. To begin with, Figures 7, 8, 9, 10, and 11 show the MSGs of the 45 programs studied. The MSGs were visually categorized according to whether or not they appear to contain no clusters (and thus a line sloping up to the right), large clusters, or enormous clusters (the latter deemed to be those occupying 70% or more of the program). Figure 7 shows the ten MSGs for programs that essentially contain no large clusters. These programs show, at most, small 'cliff drops' in the landscape of their MSG.

In contrast, most of the programs studied were found to contain large dependence clusters. Some were so large that they suggest possibly severe problems for continued software evolution. Figures 8 and 9 show the MSGs for the middle category of programs; they contain visual evidence of large clusters. Finally, Figures 10 and 11 show the MSGs of eighteen programs where these clusters were particularly pronounced.

Visually, the MSGs help to assist human identification of large dependence clusters: compare the relatively smooth landscapes of the MSGs in Figure 7 to those in

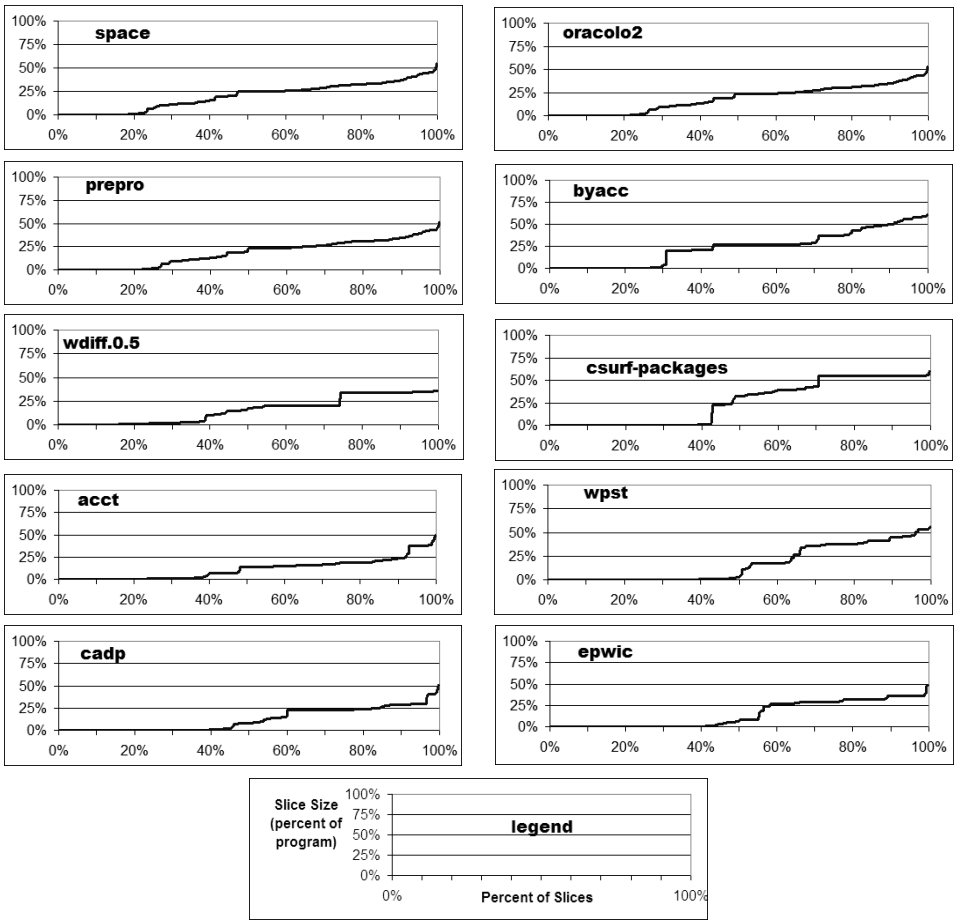

Fig. 7. MSGs for programs with an *absence of large dependence clusters* (no cliff faces in the MSGs).

Figures 8 and 9 or for a more dramatic difference, those in Figure 10 and 11 (which clearly show large, tell-tale, cliff faces).

The second visualization bands together slice sizes into 10% wide bands to reduce visual clutter. Figures 12, 13, and 14 show cluster sizes (expressed on the vertical axis as a percentage of program size (to facilitate comparison). The plane of each chart summarizes slice sizes (again as a percentage) for each program. Figure 12 shows programs with an absence of dependence clusters (parallel to the MSGs from Figure 7). For example in Figure 12, the tallest bar for csurf-packages shows that approximately one third of the clusters involve slices of size 41% to 50% of the program.

Figure 13 shows the programs with large dependence clusters. As can be seen, the presence of these clusters is manifested by the presence of large bars for the higher values of slice size. Compared to Figure 12, the distribution is more polarized into

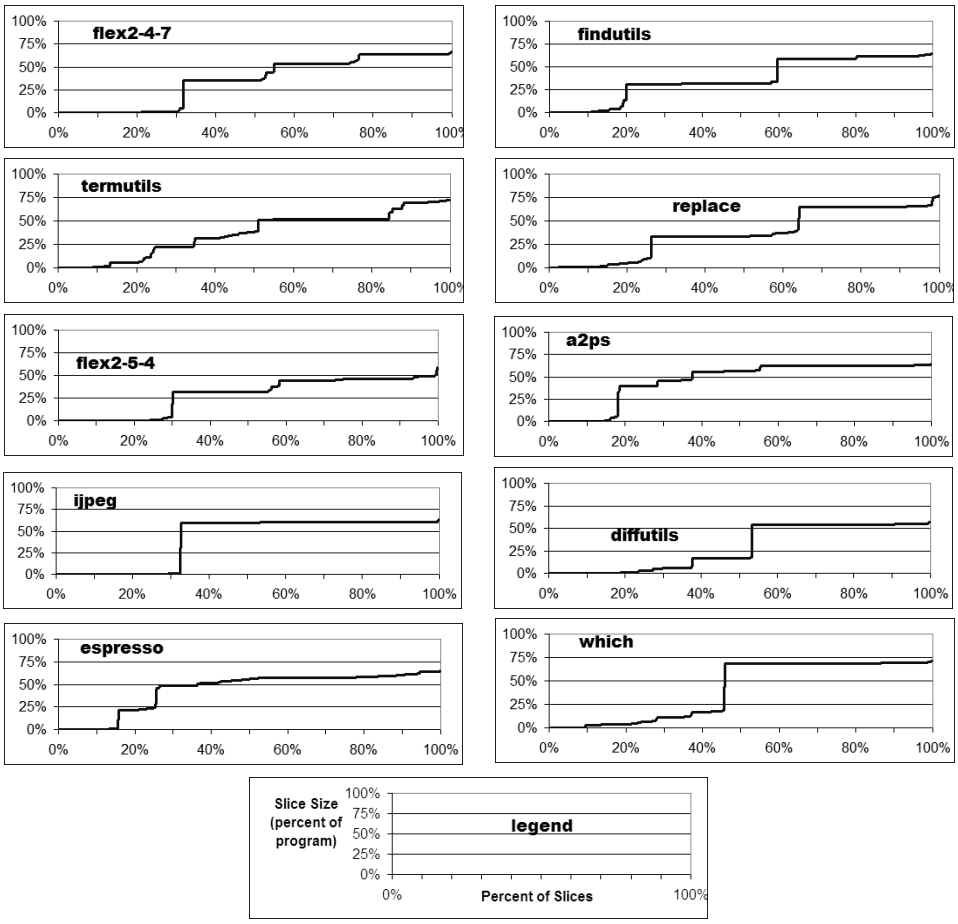

Fig. 8. Set 1 of MSGs for programs with *large dependence clusters* (denoted by cliff faces in the MSGs).

the extremes of large and small sizes. This trend toward polarization of slice sizes is even more strikingly evident in Figure 14, which depicts the distribution of slice sizes for programs with enormous clusters. In this figure, it is very clear that the programs' slice sizes are almost entirely bi-modal with comparatively few small and a majority of large slices.

Having seen the visual evidence for the existence of dependence clusters, the next step is the examination of the question as to whether there is a high prevalence of large clusters in real programs. To address this question it is necessary to consider what precisely constitutes 'large'. Earlier work [Binkley and Harman 2005b] arbitrarily set a threshold of 10%. That is, should 10% of a program be found to lie inside a single cluster, then this cluster was deemed to be 'large'. In this paper, a more elaborate approach is adopted, which allows the reader to make a choice as

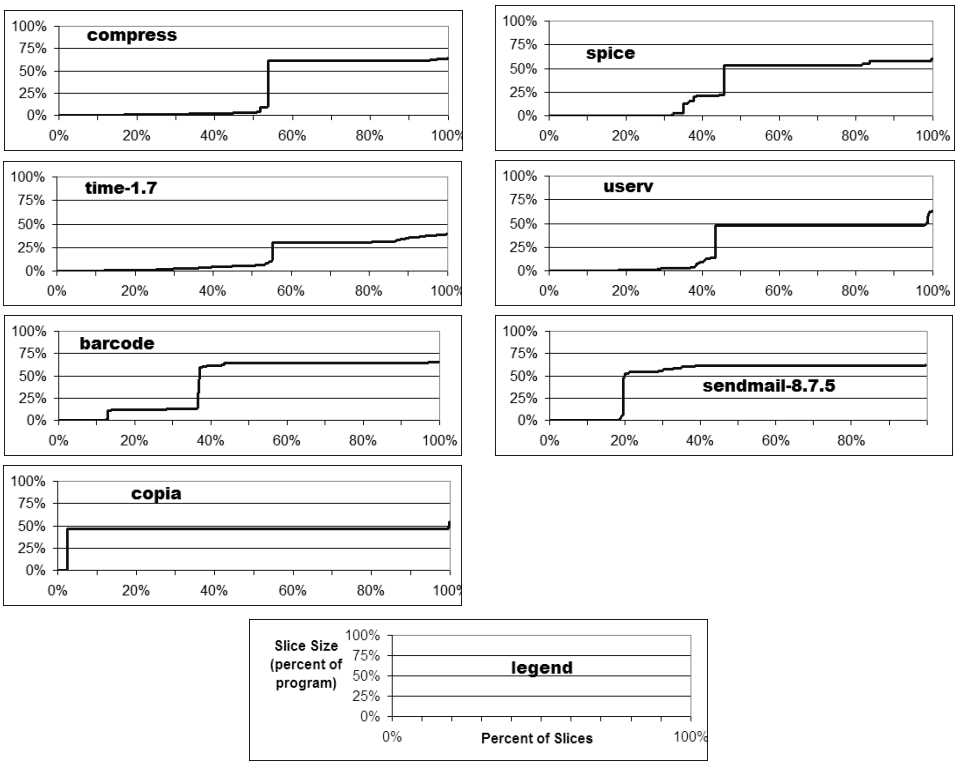

Fig. 9. Set 2 of MSGs for programs with *large dependence clusters* (denoted by cliff faces in the MSGs).

to what a reasonable threshold should be, and to examine the impact of this choice on the outcome of the question

"How many programs studied contain large clusters?"

Figure 15 shows a count of programs with large dependence clusters for various *largeness thresholds*. For example, at the extremes a threshold of zero causes all 45 program to include a cluster, while none of the programs include a clusters that consumes 100% of the program. Setting the threshold for largeness at 10% all but 5 of the 45 programs have large dependence clusters. While the choice of 'largeness threshold' is arbitrary, it would seem that a cluster that consumed 10% of the program would be worth investigation and so would be sufficiently large to appear on the 'investigation radar' of many programmers. Furthermore, at the quartile points 24, 18, and 4 programs include at least one cluster of at least 25%, 50%, and 75% of the code in the program. It is clear from this data that for any reasonable definition of large, there is a considerable number of programs that contain large clusters. Furthermore, some of these clusters are enormous relative to the size of

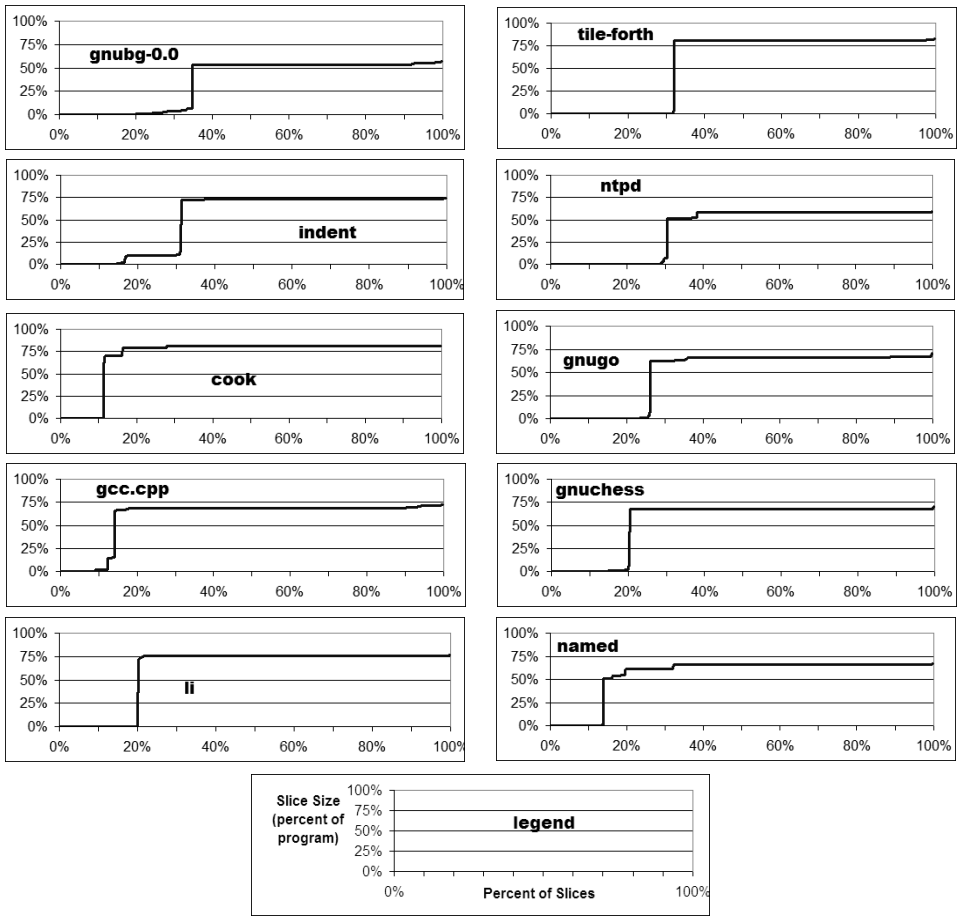

Fig. 10. Set 1 of MSGs of programs with *enormous dependence clusters*.

the program. This result has far-reaching implications for analysis techniques that rely upon program dependence analysis.

The data in Figure 15 takes a rather strict view on the slices that potentially form a cluster: it requires that slice size match exactly. A common pattern found in the source code studied involves overlapping slices of *similar* size. For example, consider the statement if (i > 10) within the loop for(i=0; i<N; i++). Often the only difference between the slices on these two statements is the presence of the if statement; thus, the two are identical excepting for a single statement.

This suggests that provision should be made for minor tolerance in slice size. As with the largeness threshold, *tolerance* is treated as a parameter to the analysis, so that the effect of different levels of tolerance can be judged by the reader. Formally, tolerance is incorporated into the analysis as follows: let $s[i]$ denote the number of slices of having size i. The count of slices having size i within tolerance t is computed as the sum of $s[i]$ through $s[i+t]$. For example, when tolerance is 2, the

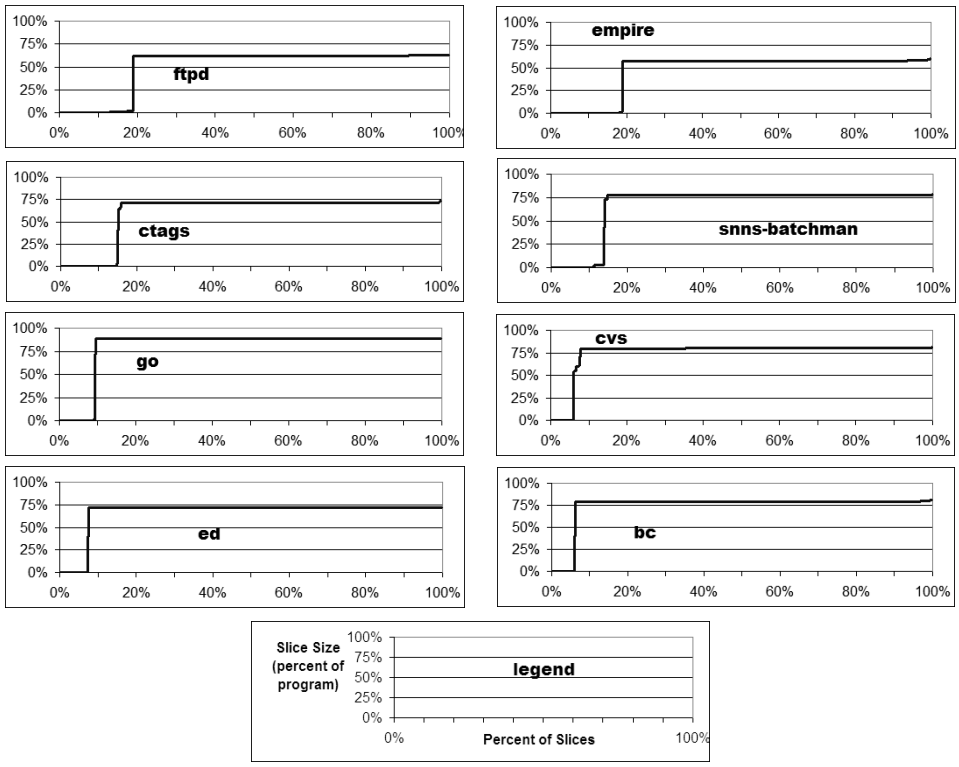

Fig. 11. Set 2 of MSGs of programs with *enormous dependence clusters*.

number of slices in the cluster for size i is given as the sum $s[i] + s[i+1] + s[i+2]$; thus, Figure 15 shows the data for tolerance of $t = 0$.

Figure 16 expands Figure 15 to include other values of tolerance. In the figure, a point at (x, y, z) for (Tolerance, Largeness Threshold, Program Count) means that there are z programs with clusters larger than y% of the program's nodes when clusters are permitted to be constructed from slices that differ by no more than x nodes. The 140 mode maximum x value for tolerance corresponds to about 10 statements. Note that, in the figure, there is an evident increase going from a tolerance of zero to a tolerance of one. After this the increases continues, but at a slower rate.

While it is clear from Figure 15 that even when no tolerance is permitted, there are programs that have large clusters, admitting a small amount of tolerance initially has a dramatic effect on the results. This tolerance essentially allows more nodes to be considered to have identical slices (give or take a "few nodes of tolerance"). As seen in Figure 16, allowing a little tolerance initially produces more programs that fall into the category of having large clusters. However, this effect soon diminishes, indicating that there are a large number of 'near miss' large clusters, which are only a few nodes away from being counted.

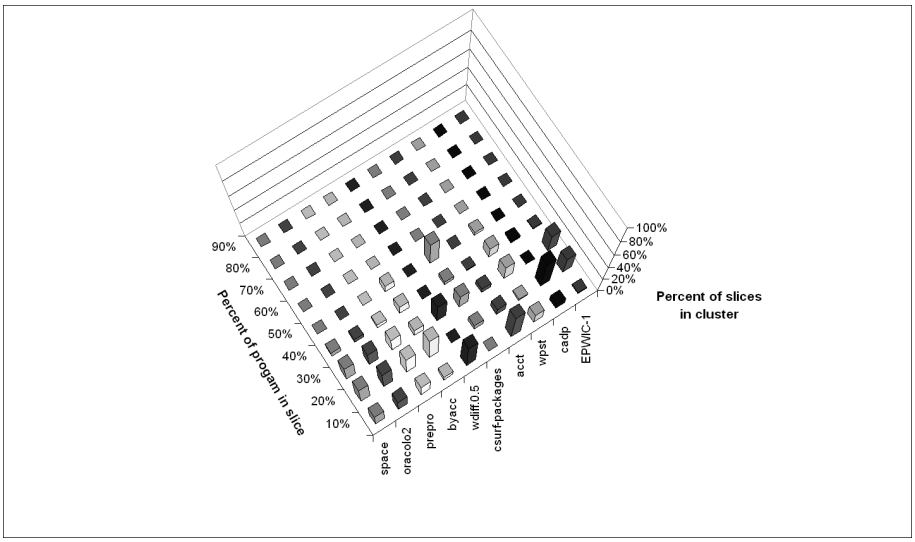

Fig. 12. Size/Frequency for programs with no large clusters. See Figure 7 for the corresponding MSGs.

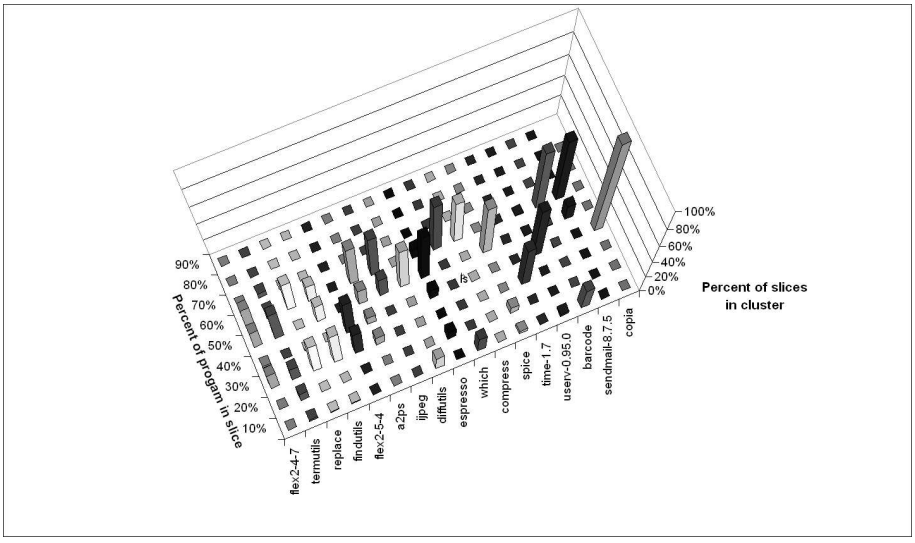

Fig. 13. Size/Frequency for programs with large clusters. See Figures 8 and 9 for the corresponding MSGs.

6. CAUSES OF DEPENDENCE CLUSTERS

A complete treatment of the possible causes of dependence clusters is beyond the scope of the present paper. However, such an investigation is a priority for future work, since the present paper has presented evidence to suggest that large

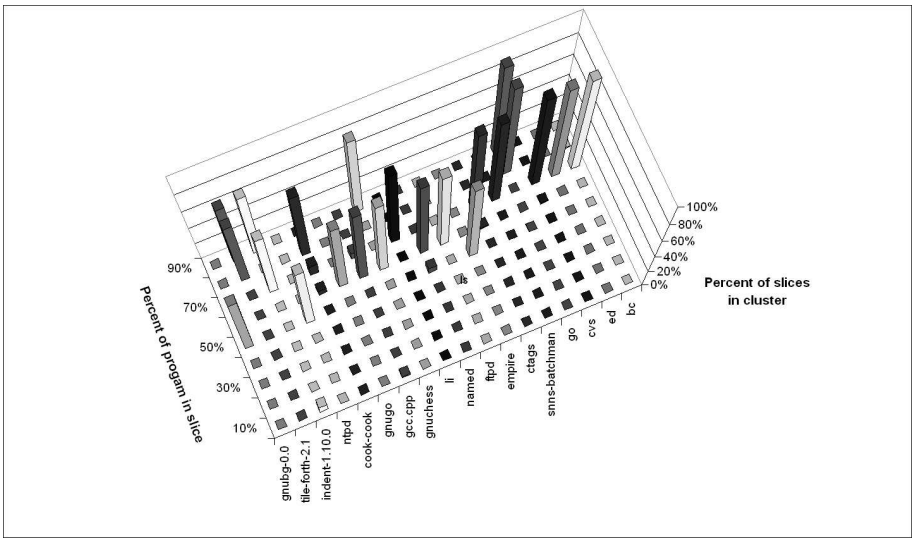

Fig. 14. Size/Frequency for programs with enormous clusters. See Figure 10 and 11 for the corresponding MSGs.

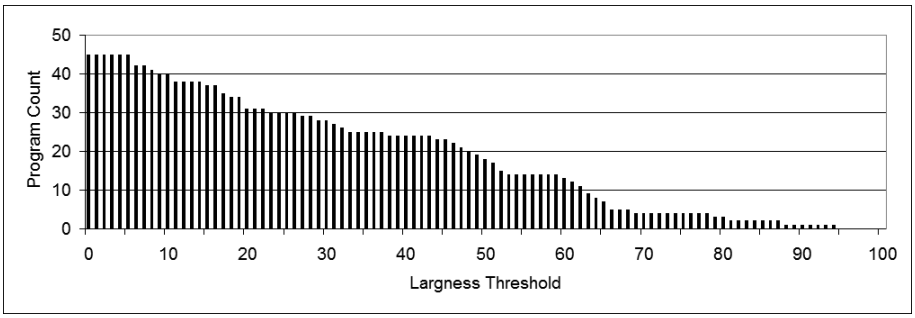

Fig. 15. The number of the 45 programs having large clusters for various largeness thresholds.

dependence clusters may be highly prevalent and may have detrimental effects on software evolution.

In this section, results are presented that give a glimpse of the possibilities for future study of dependence cluster causes. The section shows that consideration of the nodes of the dependence graph, each in turn, can be used to reveal those nodes that can be deemed to cause clusters to occur. The results are promising, because they reveal that there do, indeed, exist nodes that account for large portions of the code found to reside inside some of the large dependence clusters.

This observation is encouraging. If the analysis can identify causes and if these can be removed, then the pernicious effects of large dependence clusters may be avoidable. More work is required to investigate this possibility.

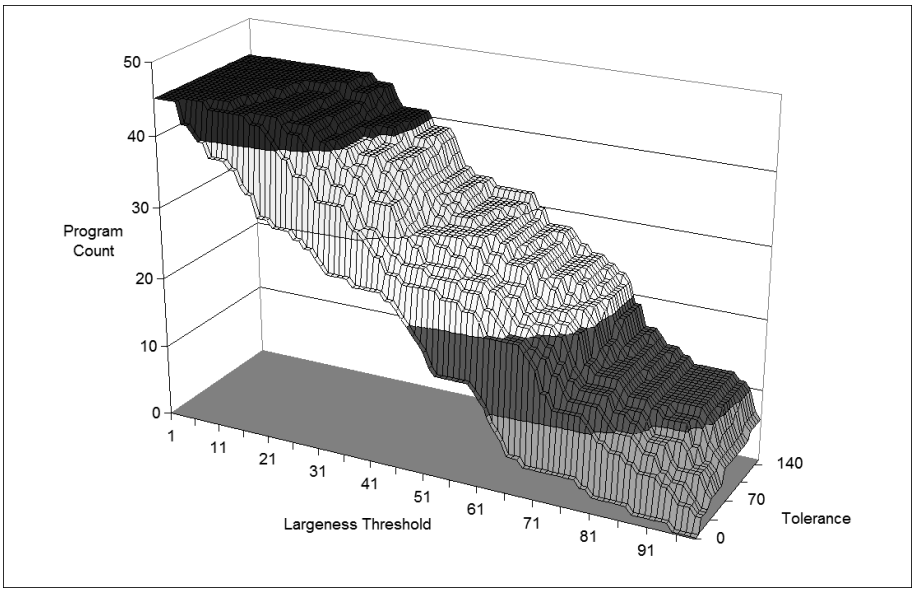

Fig. 16. The effect of tolerance on programs having large clusters. Tolerance is measured in SDG nodes. There are several nodes per statement. As the figure shows, a small allowance of this tolerance increase the number of programs that would be considered to contain large dependence clusters.

In order to identify causes of clusters, each SDG node in each of the programs studied was considered in turn. For each node, its effect on large clusters was assessed by marking the node and recomputing the slices without traversing the marked node. This approach provides a quick and simple technique for identifying nodes that have a big impact on the presence of large clusters. The results for three of the programs studied are presented in Figure 18. These are typical of the distribution of 'node effects' for the other programs studied. The results for all vertices (in all programs studied) that have a greater than 1% effect on dependence are shown in Figure 19.

As can be seen, for most nodes, there is little effect; the nodes cause little overall dependence on their own, and marking them has little or no effect on slice sizes. However, there are a few 'important' nodes that do have a strong and noticeable effect on the sizes of dependence clusters in the programs studied. It turns out that these are all predicate nodes, that denote important decisions in the control structure of the program. These decision points gather together and control important computations, leading to a cluster. Simply removing the decision point from slice computation breaks up the cluster. Of course, the automated identification of clusters and these causes can only point to the cause. It remains a task for the software engineer to consider whether or not the program can be re-structured to avoid this cause. However, the automated analysis helps to inform and guide this human decision making process.

Program	Percentage	Vertex Type	Source
copia	80.20%	control-point	switch (a)
time-1.7	47.06%	control-point	switch (*++fmt)
replace	13.82%	control-point	if (in_set_2())
copia	13.70%	formal-in	int a
copia	13.66%	actual-in	m
which	13.57%	control-point	while (next)
copia	13.57%	expression	m=urna[n]
copia	13.11%	expression	urna[j]=probtab[i].posizione
replace	12.58%	actual-out	in_set_2()
findutils	11.92%	expression	parse_function = find_parser ()
findutils	11.91%	actual-out	find_parser ()
findutils	11.63%	indirect-call	(*parse_function) ()
copia	10.75%	control-point	while (i<riga)
compress	10.73%	expression	text_buffer[bufindex]=c2
replace	10.56%	actual-out	amatch()
compress	10.56%	expression	c2=getranchar()
copia	10.55%	formal-in	int riga
replace	10.49%	expression	m = amatch()
compress	10.43%	actual-out	getranchar()
which	10.11%	call-site	find_command_in_path()

Fig. 17. The source code corresponding to the top vertices that cause dependence in the programs studied

Figure 17 presents a table listing the code fragments that denote each of these important predicates. The predicates are listed in descending order of their contribution to the size of large dependence clusters. For example, consider the copia running example. This program has a large dependence cluster and it turns out that this cluster has *one* node that causes it.

Inspection of the code reveals that copia has 234 small functions that call one large function, seleziona, which in turn, calls the smaller functions. The choice as to which of the smaller functions is called is made by passing a numeric token that plays the role of a kind of 'run time program counter'. The counter is checked in seleziona and each of the smaller functions passes a value indicating the next function to be called. The node that causes the dependence cluster occurs in seleziona; it is the switch statement that determines which of the 234 smaller functions is called. This creates a mutual recursion that involves most of the functionality of the program. The mutual recursion that occurs because, statically, all functions may call all other functions producing the large dependence cluster.

7. FORWARD DEPENDENCE CLUSTERS

Hitherto, this paper has considered backward dependence and the dependence clusters that can occur as a result of inter dependencies. The dependence clusters considered so far in the paper can therefore be thought of as 'backward dependence clusters'. Since dependence can also be computed in a forward direction, it is natural to consider whether forward dependence clusters are also prevalent in the programs studied and whether they are similarly large.

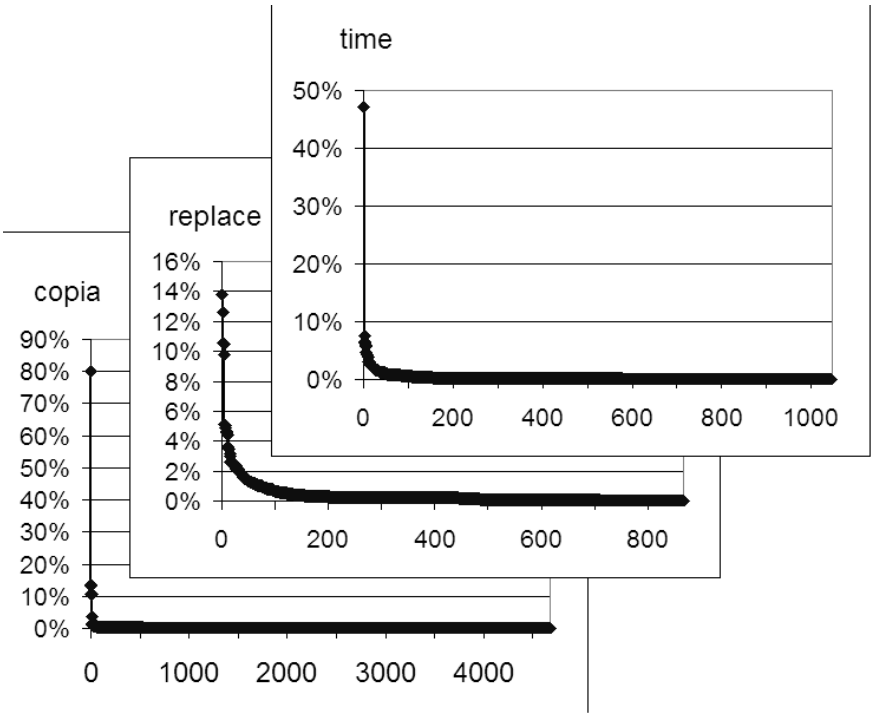

Fig. 18. Stacked figures: Vertex Reduction for three of the programs studied, illustrating the distribution of vertices that cause large amounts of dependence

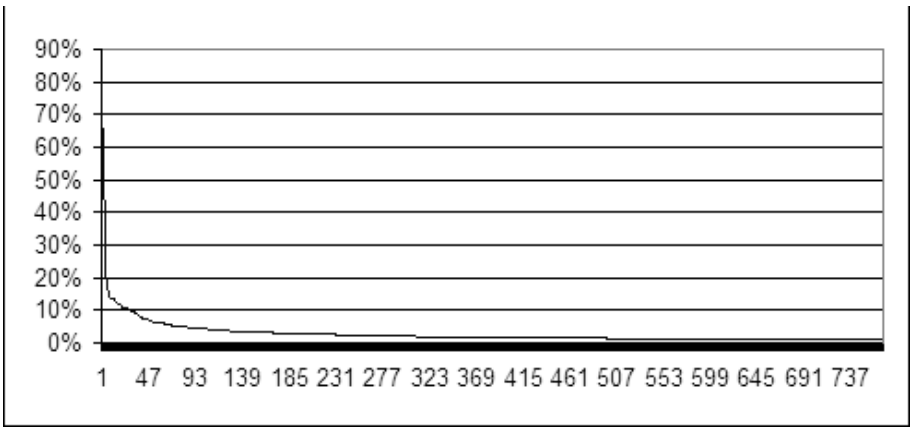

Fig. 19. The effect of vertices on dependence: Top 1% of vertices.

Of course, forward dependence analysis is a dual of backward dependence [Horwitz et al. 1990] and so it would be reasonable to expect a relationship between forward and backward dependence clusters. However, Binkley and Harman [2005a]

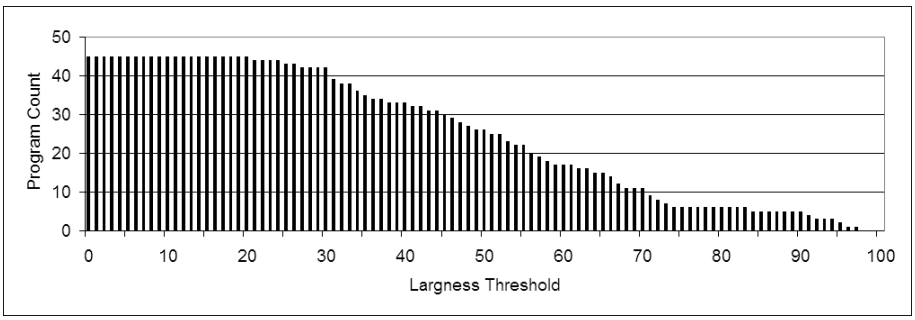

Fig. 20. Programs with large Dependence Clusters for Forward Dependence

showed that forward slice distributions are different to their backward counterparts; the forward slice distribution contains far more smaller slices and a few super large slices compared to the backward slice distribution counterparts.

Figure 20 shows the numbers of programs that have large dependence clusters on the y-axis, against the size above which a cluster is considered to be large on the x axis. As can be seen the number and distribution is similar (though not identical) to the results for backward dependence clusters (shown earlier in Figure 15).

8. THE IMPACT OF DEPENDENCE CLUSTERS ON IMPACT ANALYSIS

There is a relationship between forward dependence and impact analysis. The forward slice of a program p at a point n in p indicates all those statements that can be statically determined to have potential effects arising from changes to n. The relationship is not quite as simple as 'all those statements in the forward slice are affected should n be changed' for two reasons:

(1) Static dependence caters for all inputs and so the effect may not occur in practice if the experience of practical execution avoids those inputs that lead to the effect manifesting itself

(2) Static analysis is conservative and so it may indicate dependencies between nodes of the SDG even where no such dependence is present (for any possible input). This is a necessary limitation arising from the undecidability of minimal slicing.

However, the size of the forward slice is one possible indication of the potential impacts of a change. Such an analysis is essentially a static analysis of potential change impact in the spirit of other dependence based static change impact analyses [Yau and Collofello 1985; Black 2001]. Recent work on change impact analysis has also considered more dynamic forms of analysis, based on existing pools of test data and on the original and changed versions of the program [Ren et al. 2006; Ren et al. 2005]. Forward dependence cluster analysis could also be used to consider dynamic dependencies by adopting a union slicing approach [Beszédes and Gyimóthy 2002; De Lucia et al. 2003] over dynamic slices and checking for dependence clusters. However, such a problem remains a topic for future work. Impact analysis has also recently been applied to other software engineering artifacts as well as pro-

grams [Canfora and Cerulo 2005; Sherriff and Williams 2008]. It is possible that dependence clustering among these artifacts may also be worthy of consideration in future work.

Programs with large forward dependence clusters are potential courses of problems for on-going software maintenance and evolution, because many of the possible changes to these programs may have very far reaching effects. That is, a change to any statement inside the forward dependence cluster, potentially affects all other statements inside the cluster. Even where these effects are partly the product of over-conservatism in the static analysis, it may, nonetheless be a requirement that potential effects are explored following any such change. There is a large human cost associated with this checking process, whether the effects are real or merely apparent. As such, programs with large forward dependence clusters are worthy of special consideration if they are likely to undergo change.

Furthermore, it may be possible to determine the cause of a large forward dependence cluster. In such situations, the developer can use dependence analysis, not only to identify the presence of dependence clusters, but also to locate their cause. This may permit remedial action or other steps to ameliorate the effects of the large forward dependence cluster, with potential benefits to on-going maintenance and evolution.

As an example of this possibility, consider again the running example program: copia. The program has a large forward dependence cluster. The analysis of the effect on dependence of each vertex suggests that the switch statement may be the primary cause of this cluster. In order to further explore this possibility, we performed a simple 'by hand' refactoring that simulates the replacement of the integer variable next_state with direct recursive function calls. This removed the potentially problematic switch statement. The 'before' and 'after' MSGs are shown in Figure 21. As can be seen from this figure, removing the switch and its control dependence removes the cluster. As a result of this, the potential impact of changes to the program could be greatly reduced.

The primary reason for the high level of dependence in the original program lies in the statement switch(next_state). This causes what might be termed 'conservative dependence collateral damage'; the static analysis could not determine that when function f() returned a 5 this caused the switch statement to eventually invoke function g(). Instead, the analysis made the conservative assumption that a call to f() might be followed by a call any of the functions appearing in the switch statement. Once the program was refactored, this conservative dependence collateral damage was removed.

Of course, in order to reap the benefit of this analysis, the programmer will have to consider ways in which the program can be re-written to change the flow of control. Though the switch statement is the primary cause, there may be other changes required in order to re-write the program so as to avoid the cluster. Such a refactoring is a change in itself, and for such a large forward dependence cluster, we argued that change should be considered a potential hazard. Whether such a refactoring is deemed cost-effective is a decision that can only be taken by the engineers and managers who have to maintain the program in question. However,

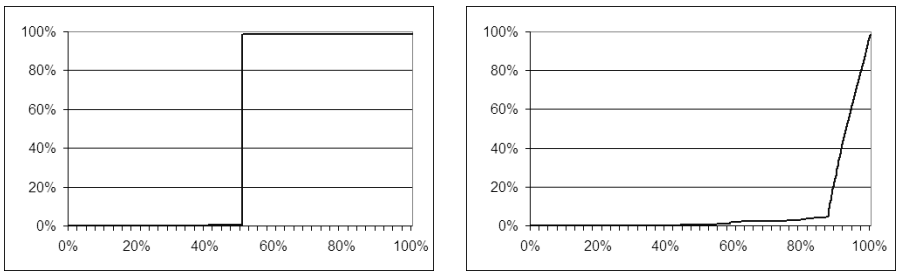

Fig. 21. The copia program with and without its large cluster

this section illustrates the way in which dependence cluster analysis can provide an automated mechanism for identifying potential problems and their causes.

9. THREATS TO VALIDITY

This section considers the threats to the internal and external validity of the results presented in the two empirical studies. In the experiments, the primary external threat arises from the possibility that the selected programs are not representative of programs in general, with the result that the findings of the experiments do not apply to 'typical' programs. This is a reasonable concern that applies to any study of program properties. To mitigate this concern, the study considered a large code base covering a wide variety of different tasks including applications, utilities, games, and operating system code. The code base also contained both commercial and open source programs. There is, therefore, reasonable cause for confidence in the results obtained and the conclusions drawn from them. However, all of the programs studied were C programs, so there is greater uncertainty that the results will hold for other programming paradigms such as object-oriented or aspect-oriented [Kiczales 1997].

Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variables on the dependent variable. In this experiment, the possible threats come from the potential for faults in the slicer and the values chosen for acceptable similarity (which affects the verification study) and largeness threshold (which affects the validation study). A mature and widely used slicing tool (CodeSurfer) was used to mitigate the first concern.

For similarity, the results showed that an overwhelming proportion (99.5%) of clusters of same size slices have over 99% the same nodes. For the applications of dependence clustering, this level of similarity is well within acceptable limits.

For threshold, a range of values was used. However, other than at very small values, the number of large clusters is not strongly affected by this number. Once again, this was a conservative choice, well within that which would be considered important in the application of dependence cluster analysis. Furthermore, some of the programs studied were found to have enormous clusters consuming 70% or more of the program (e.g., those shown in Figures 10 and 11), suggesting that the evidence for the existence of large dependence clusters is extremely strong.

Another possible concern comes from the precision of the pointer analysis used. An overly conservative, and therefore imprecise, analysis may tend to increase the levels of dependence and potentially may also increase the size of dependence clusters. There is no automatic way to tell whether a cluster arises because of imprecision in the computation of dependence or whether it is 'real'. One source of potential imprecision comes from the pointer analysis used. We used Codesurfer's most precise points-to analysis options for the study in order to address this potential concern. The algorithm, based on the work of Fahndrich et al. [1998], computes points-to sets by applying a general constraint solver.

10. RELATED WORK

This paper uses a well-established source code analysis technique (slicing) to identify dependence clusters. The first applications of slicing were for debugging and testing [Weiser 1982; Lyle and Weiser 1987]. There have been several surveys of slicing techniques, applications and variations [Binkley and Gallagher 1996; Binkley and Harman 2004b; De Lucia 2001; Harman and Hierons 2001; Tip 1995]. Slicing has been applied to many problems related to software engineering, most notably re-engineering [Canfora et al. 1994], program comprehension [De Lucia et al. 1996], debugging [Lyle and Weiser 1987], testing [Binkley 1998; Gupta et al. 1992; Harman and Danicic 1995], cohesion measurement [Bieman and Ott 1994], and impact analysis [Gallagher and Lyle 1991].

The work reported in the present paper follows a recent trend toward the use of slicing as *a means to an end* rather than an end in itself. This trend finds echoes in the development of the literature on program slicing. For example, early work on slice-based metrics began to move away from slices as an end product and toward their use as a part of the computation of metrics. Bieman and Ott [1994; 1989], Lakhotia [1993], and later Meyers and Binkley [2004] use slicing and dependence analysis as part of a technique to measure the level of functional cohesion, while Black [2001], uses dependence analysis to measure the ripple effect. Balmas [2002], and Korel and Rilling [1997; 1998] use dependence analysis indirectly for comprehension, Krinke and Snelting [1998] use a variant of static slicing for information flow validation, Beszédes et al. [2001] and Canfora et al. [1998] use dependence analysis as part of a wider approach to maintenance and re-engineering.

The remainder of this section considers the implications of the discovery of large and widespread statement-level dependence clusters. There has been considerable work concerning *clustering in the large* where the focus is typically at the function or module level [Hutchens and Basili 1985; Mancoridis et al. 1999; Eisenbarth et al. 2003; Mahdavi et al. 2003]. By comparison, the clusters of interest in the present paper are more fine-grained; they are the nodes of the SDG. This represents clustering at the statement level.

The present paper is the first to study statement-level dependence clusters using slicing as an identifying mechanism. It is also the first to report the prevalence of these clusters. This finding is important because of the many related source code analyses that rely upon dependence based analysis and which, implicitly, require freedom from such large clusters for optimal performance. The impact of large de-

pendence clusters are now considered in several related analyses including program comprehension, testing, maintenance, reuse, and parallelization.

Much work on program comprehension deals with a comprehension process that inherently revolves around source code. Many authors [Binkley et al. 2000; De Lucia et al. 1996; Komondoor and Horwitz 2003; Korel and Rilling 1997; Ning et al. 1994; Zhao 2002] have advocated dependence analysis as a way of aiding the engineer in the difficult and time-consuming task of program comprehension. Much of this work rests upon empirical evaluation of relatively small-scale programs. As such, the widespread presence of large dependence clusters has gone unnoticed and has, therefore, remained unreported.

The discovery that large dependence clusters may be considerably more widespread than previously appreciated will have a significant impact upon the application of dependence analysis to program comprehension. This will not render dependence based comprehension approaches inapplicable. However, it does mean that the presence of dependence clusters needs to be accounted for in practical applications.

For example, Balmas describes a technique for controlling the complexity of source code comprehension by hierarchically partitioning source code into nested blocks [Balmas 2002]. The approach is backed up by a tool that supports zooming into and out of the hierarchical dependence structure so-created. For programs with large dependence clusters, such hierarchical decomposition will be seriously constrained. A dependence cluster will not naturally submit to decomposition, rendering zooming into a cluster impossible. Further research may be required in order to find techniques for breaking up dependence clusters in such scenarios. Similar dependence-based source code browsing is advocated by Deng et al. [Deng et al. 2001] and by Rilling et al. [Rilling et al. 2001; Rilling and Mudur 2002]. These approaches will also need to consider the impact of dependence clusters.

In software testing, dependence analysis has been shown to be effective at reducing the computation effort required to automate the test data generation process [Binkley and Harman 2004a; Harman et al. 2007]. This is important, because test automation is widely held to be the key to a successful software testing program [Baresel et al. 2002; Colin et al. 2004; Ferguson and Korel 1996; Korel 1990; Li and Wu 2004; Tracey et al. 2000]. Using dependence analysis, it is possible to reduce both the amount of code to be tested and the size of the input domain. For example, the analysis may start with a branch to be covered and proceed to use dependence analysis to determine the part of the program and input space upon which the branch depends [Harman et al. 2007].

While programs free from dependence clusters will submit to such analysis, for those with large clusters no such dependence-based optimization is possible. It is thus likely that such programs will be harder to test. This suggests the development of dependence cluster study as a proxy for source code testability [Binder 1994; Voas and Miller 1995]. It is also possible that testability transformation [Harman et al. 2004] may be one way to ameliorate the difficulties posed by dependence clusters; transformation algorithms for cluster decomposition will be required to improve testability.

In software maintenance, dependence analysis is used to protect the software maintenance engineer against the potentially unforeseen side effects of a maintenance change. This can be achieved by measuring the impact of the proposed change [Black 2001; Gallagher 1996; Hutchens and Gallagher 1998; Yau and Collofello 1985] or by attempting to identify portions of code for which a change can be safely performed free from side effects [Gallagher 1992; Tonella 2003]. For this work, the presence of large dependence clusters will tend to produce higher values for change impact metrics, because any change that affects any of a cluster's nodes will, by definition, affect all the nodes of the cluster. Furthermore, any effort to identify safe portions of the code to which changes can be applied will be hampered by the presence of large dependence clusters; such readily safely-changeable portions of code will be few and far between.

Program dependence analysis has been proposed as a supporting technology for salvaging [Canfora et al. 1994] and re-use [Beck and Eichmann 1993; Komondoor and Horwitz 2000] of pieces of source code, in order to gain value from the presence of well tested, workable sub-components within legacy systems. The functionality to be reused may be textually scattered throughout the legacy system's source code, making it difficult to extract without dependence analysis. This scattering often arises due to the widely observed phenomenon of gradual architectural decay [Lehman 1980; 1998]. Dependence analysis can assist in this difficult extraction process, by teasing out and collecting those computations needed to support the functionality to be reused. For a legacy system with large dependence clusters, any attempt to reuse a computation denoted by a source code elements lying inside a cluster will draw out the entire cluster. This makes reuse expensive, cumbersome, and error prone.

Program dependence also underpins attempts to transform programs into forms more amenable to parallel execution [Ryan 2000]. Once again, the presence of large dependence clusters will impact on the degree to which such automated parallelization will be successful. The transformations considered in work on auto-parallelization focus on loop computation, because loops denote likely computational bottlenecks. The algorithms attempt to separate computations within a single loop body, enabling independent execution of several reduced loop-body instances. However, should the body of a loop be located within a large dependence cluster, then such separation attempts will fail; the precondition of the transformation will never be satisfied, since no code motion is possible within a dependence cluster.

11. SUMMARY AND FUTURE WORK

This paper introduced the concept of statement-level dependence clusters and shows how they can be identified using a slice-based visualisation called the Monotone Slice-size Graph. The overall approach was evaluated by two empirical studies that verified the approximation used to identify dependence clusters and validated the approach in terms of the prevalence of these clusters in a large code base of 45 real-world programs.

The results indicate that dependence clusters are surprisingly prevalent, making their study both timely and important. The paper shows that most of the 45

programs studied contain large clusters; furthermore, a non-trivial subset contain truly enormous dependence clusters. The paper also discusses the implications of this widespread occurrence of large dependence clusters on comprehension, testing, maintenance, reuse, and automated parallelization.

The paper considers both forward and backward dependence clusters and illustrates the connection between forward dependence cluster analysis and static impact analysis. The paper argues that causes of dependence clusters can and should be identified, presenting results for one possible cause: individual predicate nodes. The paper illustrates the application of this analysis by showing how the removal of a forward dependence cluster may reduce the impact of future changes.

Future work will consider whether there is a relationship between dependence clusters and other problematic features such as code clones [Kamiya et al. 2002; Gallagher and Layman 2003] and whether the results for the prevalence of large dependence clusters are replicated for the OO and AOP programming paradigms. It will also be interesting to see whether the previously studied relationships between dependence and faults [Kusumoto et al. 2002] indicate that programs with large dependence clusters are either more or less fault prone. Future work will also consider the size and ramifications of clusters of dependence at higher levels of abstraction than the program level.

REFERENCES

Balmas, F. 2002. Using dependence graphs as a support to document programs. In 2^{st} *IEEE International Workshop on Source Code Analysis and Manipulation* (Montreal, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 145–154.

Baresel, A., Sthamer, H., and Schmidt, M. 2002. Fitness function design to improve evolutionary structural testing. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference* (New York). Morgan Kaufmann Publishers, San Francisco, CA 94104, USA, 1329–1336.

Bates, S. and Horwitz, S. 1993. Incremental program testing using program dependence graphs. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, Charleston, South Carolina, 384–396.

Beck, J. and Eichmann, D. 1993. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)*. IEEE Computer Society Press, Los Alamitos, California, USA, 509–518.

Beszédes, A., Gergely, T., Szabó, Z. M., Csirik, J., and Gyimóthy, T. 2001. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*. IEEE Computer Society, 105–113.

Beszédes, A. and Gyimóthy, T. 2002. Union slices for the approximation of the precise slice. In *IEEE International Conference on Software Maintenance* (Montreal, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 12–20.

Bieman, J. M. and Ott, L. M. 1994. Measuring functional cohesion. *IEEE Transactions on Software Engineering 20,* 8 (Aug.), 644–657.

Binder, R. V. 1994. Design for testability in object–oriented systems. *Communications of the ACM 37,* 9, 87–101.

Binkley, D. and Harman, M. 2005a. Forward slices are smaller than backward slices. In 5^{th} *IEEE International Workshop on Source Code Analysis and Manipulation* (Budapest, Hungary, September 30th-October 1st 2005). IEEE Computer Society Press, Los Alamitos, California, USA, 15–24.

Binkley, D. and Harman, M. 2005b. Locating dependence clusters and dependence pollution. In 21^{st} *IEEE International Conference on Software Maintenance* (Budapest, Hungary, September

30th-October 1st 2005). IEEE Computer Society Press, Los Alamitos, California, USA, 177–186.

Binkley, D., Harman, M., and Krinke, J. 2006. Animated visualisation of static analysis: Characterising, explaining and exploiting the approximate nature of static analysis. In 6^{th} *International Workshop on Source Code Analysis and Manipulation (SCAM 06)*. Philadelphia, Pennsylvania, USA, 43–52.

Binkley, D. W. 1997. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering 23,* 8 (Aug.), 498–516.

Binkley, D. W. 1998. The application of program slicing to regression testing. *Information and Software Technology Special Issue on Program Slicing 40,* 11 and 12, 583–594.

Binkley, D. W. and Gallagher, K. B. 1996. Program slicing. In *Advances in Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1–50.

Binkley, D. W. and Harman, M. 2004a. Analysis and visualization of predicate dependence on formal parameters and global variables. *IEEE Transactions on Software Engineering 30,* 11, 715–735.

Binkley, D. W. and Harman, M. 2004b. A survey of empirical results on program slicing. *Advances in Computers 62*, 105–178.

Binkley, D. W., Harman, M., and Krinke, J. 2007. Empirical study of optimization techniques for massive slicing. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS) 30*, 3:1–3:33.

Binkley, D. W., Harman, M., Raszewski, L. R., and Smith, C. 2000. An empirical study of amorphous slicing as a program comprehension support tool. In 8^{th} *IEEE International Workshop on Program Comprehension* (Limerick, Ireland). IEEE Computer Society Press, Los Alamitos, California, USA, 161–170.

Binkley, D. W., Horwitz, S., and Reps, T. 1995. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology 4,* 1, 3–35.

Black, S. E. 2001. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice 13*, 263–279.

Canfora, G. and Cerulo, L. 2005. Impact analysis by mining software and change request repositories. In *IEEE Metrics Symposium.* IEEE Computer Society, 29.

Canfora, G., Cimitile, A., De Lucia, A., and Lucca, G. A. D. 1994. Software salvaging based on conditions. In *International Conference on Software Maintenance* (Victoria, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 424–433.

Canfora, G., Cimitile, A., and Munro, M. 1994. RE^2: Reverse engineering and reuse re-engineering. *Journal of Software Maintenance: Research and Practice 6,* 2, 53–72.

Canfora, G., De Lucia, A., and Munro, M. 1998. An integrated environment for reuse reengineering C code. *Journal of Systems and Software 42*, 153–164.

Cimitile, A., De Lucia, A., and Munro, M. 1995. Identifying reusable functions using specification driven program slicing: a case study. In *Proceedings of the IEEE International Conference on Software Maintenance* (Nice, France). IEEE Computer Society Press, Los Alamitos, California, USA, 124–133.

Cimitile, A., De Lucia, A., and Munro, M. 1996. A specification driven slicing process for identifying reusable functions. *Software maintenance: Research and Practice 8*, 145–178.

Colin, S., Legeard, B., and Peureux, F. 2004. Preamble computation in automated test case generation using constraint logic programming. *Software Testing, Verification and Reliability 14,* 3 (Sept.), 213–235.

De Lucia, A. 2001. Program slicing: Methods and applications. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation* (Florence, Italy). IEEE Computer Society Press, Los Alamitos, California, USA, 142–149.

De Lucia, A., Fasolino, A. R., and Munro, M. 1996. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension* (Berlin, Germany). IEEE Computer Society Press, Los Alamitos, California, USA, 9–18.

De Lucia, A., Harman, M., Hierons, R., and Krinke, J. 2003. Unions of slices are not slices. In 7^{th} *IEEE European Conference on Software Maintenance and Reengineering (CSMR 2003)* (Benevento, Italy). IEEE Computer Society Press, Los Alamitos, California, USA, 363 – 367.

Deng, Y., Kothari, S., and Namara, Y. 2001. Program slice browser. In 9^{th} *IEEE International Workshop on Program Comprenhesion* (Toronto, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 50–59.

Eisenbarth, T., Koschke, R., and Simon, D. 2003. Locating features in source code. *IEEE Transactions on Software Engineering 29,* 3. Special issue on ICSM 2001.

Fahndrich, M., Foster, J. S., Su, Z., and Aiken, A. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montréal, Canada). Association for Computer Machinery, 85–96.

Ferguson, R. and Korel, B. 1996. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology 5,* 1 (Jan.), 63–86.

Gallagher, K. and Binkley, D. 2003. An empirical study of computation equivalence as determined by decomposition slice equivalence. In *Proceedings of the 10th Working Conference on Reverse Engineering, WCRE–03*. 316 – 322.

Gallagher, K. and Layman, L. 2003. Are decomposition slices clones? In *Proceedings of the 11th International Workshop on Program Comprehension*.

Gallagher, K. and O'Brien, L. 2001. Analyzing programs via decomposition slicing. In *Proceedings of International Workshop on Empirical Studies of Software Maintenance, WESS*.

Gallagher, K. B. 1992. Evaluating the surgeon's assistant: Results of a pilot study. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, Los Alamitos, California, USA, 236–244.

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

Gallagher, K. B., Harman, M., and Danicic, S. 2003. Guaranteed inconsistency avoidance during software evolution. *Journal of Software Maintenance and Evolution 15,* 6 (Nov/Dec), 393–416.

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

Grammatech Inc. 2002. The codesurfer slicing system.

Gupta, R., Harrold, M. J., and Soffa, M. L. 1992. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance* (Orlando, Florida, USA). IEEE Computer Society Press, Los Alamitos, California, USA, 299–308.

Hall, T., Rainer, A., and Jagielska, D. 2005. Using software development progress data to understand threats to project outcomes. In 11^{th} *International Software Metrics Symposium (METRICS 2005)*. IEEE Computer Society Press, 18.

Harman, M., Binkley, D., Singh, R., and Hierons, R. 2004. Amorphous procedure extraction. In 4^{th} *International Workshop on Source Code Analysis and Manipulation (SCAM 04)* (Chicago, Illinois, USA). IEEE Computer Society Press, Los Alamitos, California, USA, 85–94.

Harman, M., Binkley, D. W., and Danicic, S. 2003. Amorphous program slicing. *Journal of Systems and Software 68,* 1 (Oct.), 45–64.

Harman, M. and Danicic, S. 1995. Using program slicing to simplify testing. *Software Testing, Verification and Reliability 5,* 3 (Sept.), 143–162.

Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., and Wegener, J. 2007. The impact of input domain reduction on search-based test data generation. In *ACM Symposium on the Foundations of Software Engineering (FSE '07)*. Association for Computer Machinery, Dubrovnik, Croatia, 155–164.

Harman, M. and Hierons, R. M. 2001. An overview of program slicing. *Software Focus 2,* 3, 85–92.

Harman, M., Hu, L., Hierons, R. M., Wegener, J., Sthamer, H., Baresel, A., and Roper, M. 2004. Testability transformation. *IEEE Transactions on Software Engineering 30,* 1 (Jan.), 3–16.

Harman, M., Swift, S., and Mahdavi, K. 2005. An empirical study of the robustness of two module clustering fitness functions. In *Genetic and Evolutionary Computation Conference (GECCO 2005)*. Association for Computer Machinery, Washington DC, USA, 1029–1036.

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

Horwitz, S., Reps, T., and Binkley, D. W. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12,* 1, 26–61.

Hutchens, D. and Basili, V. 1985. System structure analysis: clustering with data bindings. *IEEE Transactions on Software Engineering SE-11,* 8, 749–757.

Hutchens, M. and Gallagher, K. 1998. Improving visual impact analysis. In *Proceedings of the 1998 International Conference on Software Maintenance–98*.

Jackson, D. and Rollins, E. J. 1994. A new model of program dependences for reverse engineering. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*. 2–10.

Kamiya, T., Kusumoto, S., and Inoue, K. 2002. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering 28,* 6, 654–670.

Kiczales, G. 1997. Aspect oriented programming. *ACM SIGPLAN Notices 32,* 10 (Oct.), 162–162.

Komondoor, R. and Horwitz, S. 2000. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*. ACM Press, N.Y., 155–169.

Komondoor, R. and Horwitz, S. 2003. Effective automatic procedure extraction. In *11th IEEE International Workshop on Program Comprehension* (Portland, Oregon, USA). IEEE Computer Society Press, Los Alamitos, California, USA, 33–43.

Korel, B. 1990. Automated software test data generation. *IEEE Transactions on Software Engineering 16,* 8, 870–879.

Korel, B. and Rilling, J. 1997. Dynamic program slicing in understanding of program execution. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)* (Dearborn, Michigan, USA). IEEE Computer Society Press, Los Alamitos, California, USA, 80–89.

Korel, B. and Rilling, J. 1998. Program slicing in understanding of large programs. In 6^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'98)* (Ischia, Italy). IEEE Computer Society Press, Los Alamitos, California, USA, 145–152.

Krinke, J. and Snelting, G. 1998. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology Special Issue on Program Slicing 40,* 11 and 12, 661–675.

Kusumoto, S., Nishimatsu, A., Nishie, K., and Inoue, K. 2002. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering 7*, 49–76.

Lakhotia, A. 1993. Rule–based approach to computing module cohesion. In *Proceedings of the* 15^{th} *Conference on Software Engineering (ICSE-15)*. 34–44.

Lakhotia, A. and Deprez, J.-C. 1998. Restructuring programs by tucking statements into functions. *Information and Software Technology Special Issue on Program Slicing 40,* 11 and 12, 677–689.

Lakhotia, A. and Singh, P. 2003. Challenges in getting formal with viruses. *virus bulletin*. September 2003.

Lehman, M. M. 1980. On understanding laws, evolution and conservation in the large program life cycle. *Journal of Systems and Software 1(3)*, 213–221.

Lehman, M. M. 1998. Software's future: Managing evolution. *IEEE Software 15,* 1 (Jan. / Feb.), 40–44.

Li, K. and Wu, M. 2004. *Effective Software Test Automation: Developing an Automated Software Testing Tool*. Sybex.

Lyle, J. R. and Weiser, M. 1987. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications* (Peking). IEEE Computer Society Press, Los Alamitos, California, USA, 877–882.

Mahdavi, K., Harman, M., and Hierons, R. M. 2003. A multiple hill climbing approach to software module clustering. In *IEEE International Conference on Software Maintenance* (Amsterdam, Netherlands). IEEE Computer Society Press, Los Alamitos, California, USA, 315–324.

Mancoridis, S., Mitchell, B. S., Chen, Y.-F., and Gansner, E. R. 1999. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings; IEEE International Conference on Software Maintenance*. IEEE Computer Society Press, 50–59.

Meyers, T. and Binkley, D. W. 2004. Slice-based cohesion metrics and software intervention. In 11^{th} *IEEE Working Conference on Reverse Engineering* (Delft University of Technology, the Netherlands). IEEE Computer Society Press, Los Alamitos, California, USA, 256–266.

Mitchell, B. S. and Mancoridis, S. 2002. Using heuristic search techniques to extract design abstractions from source code. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference* (New York). Morgan Kaufmann Publishers, San Francisco, CA 94104, USA, 1375–1382.

Mitchell, B. S. and Mancoridis, S. 2006. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering 32,* 3, 193–208.

Ning, J., Engberts, A., and Kozaczynski, V. 1994. Automated support for legacy code understanding. *Communications of the ACM 37,* 5, 50–57.

Ott, L. M. and Thuss, J. J. 1989. The relationship between slices and module cohesion. In *Proceedings of the* 11^{th} *ACM Conference on Software Engineering*. 198–204.

Ren, X., Chesley, O., and Ryder, B. G. 2006. Identifying failure causes in java programs: An application of change impact analysis. *IEEE Transactions on Software Engineering 32,* 9, 718–732.

Ren, X., Ryder, B. G., Störzer, M., and Tip, F. 2005. Chianti: a change impact analysis tool for java programs. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, G.-C. Roman, W. G. Griswold, and B. Nuseibeh, Eds. ACM, 664–665.

Rilling, J., A.Seffah, and J.Lukas. 2001. MOOSE – a software comprehension framework. In 5^{th} *World Multi-Conference on Systemics, Cybernetics and Informatics* (Orlando, Florida). 312–318.

Rilling, J. and Mudur, S. P. 2002. On the use of metaballs to visually map source code structures and analysis results onto 3d space. In 10^{th} *Working Conference on Reverse Engineering* (Richmond, Virginia). IEEE Computer Society Press, Los Alamitos, California, USA, 42–52.

Ryan, C. 2000. *Automatic re-engineering of software using genetic programming.* Kluwer Academic Publishers.

Sherriff, M. and Williams, L. 2008. Empirical software change impact analysis using singular value decomposition. In 1^{st} *IEEE International Conference on Software Testing*. IEEE Computer Society, Lillehamer, Norway, 268–277.

Tip, F. 1995. A survey of program slicing techniques. *Journal of Programming Languages 3,* 3 (Sept.), 121–189.

Tonella, P. 2003. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering 29,* 6, 495–509.

Tracey, N., Clark, J., Mander, K., and McDermid, J. 2000. Automated test-data generation for exception conditions. *Software Practice and Experience 30,* 1, 61–79.

Voas, J. M. and Miller, K. W. 1995. Software testability: The new verification. *IEEE Software 12,* 3 (May), 17–28.

Weiser, M. 1982. Programmers use slices when debugging. *Communications of the ACM 25,* 7 (July), 446–452.

Wheeler, D. A. 2005. SLOC count user's guide. http://www.dwheeler.com/sloccount/sloccount.html.

Yau, S. S. and Collofello, J. S. 1985. Design stability measures for software maintenance. *IEEE Transactions on Software Engineering 11,* 9 (Sept.), 849–856.

Zhao, J. 2002. Slicing aspect-oriented software. In 10^{th} *IEEE International Workshop on Program Comprehension* (Paris, France). IEEE Computer Society Press, Los Alamitos, California, USA, 351–260.