Analysis and Visualization of Predicate Dependence on Formal Parameters and Global Variables

David Binkley

Loyola College

Baltimore MD

21210-2699, USA

Mark Harman

King's College London

Strand, London

WC2R 2LS, UK

Keywords: Dependence Analysis, Program Slicing, Program Comprehension, Software Maintenance

Abstract

Empirical data concerning the qualitative and quantitative nature of program dependence is presented for a set of 20 programs ranging from 600 Lines of code to 167,000 lines of code. The sources of dependence considered are global variables and formal parameters and the targets considered are a program's predicate nodes. The results show that as the number of formal parameters available to a predicate increases, there is a decrease in the proportion of these formal parameters which are depended upon by the predicate. No such correlation was found for global variables. Results from theoretical and actual computation time analysis indicate that the computation of dependence information is practical, suggesting that the analysis may be beneficial to several application areas. The paper also presents results concerning correlations that provide strong evidence that the global and formal dependence sources are independent of one another and that the numbers of globals and formals are independent of the size of the procedure that contains them. Finally, two visualization techniques for displaying dependence information are introduced. Illustrations show how these visualizations and predicate dependence analysis can assist in activities such as testing, comprehension, and evolution.

1 Introduction

Of the many software engineering applications supported by source code analysis, a large number are highly sensitive to dependences between program components. However, despite its importance, there is little published work on empirical assessment of the *level* of program dependence expected, nor how it varies within and between programs.

Results concerning dependence levels and trends are important for dependence-sensitive applications. Such applications can require considerable analysis effort (by both humans and machines). For example, human program comprehension is an inherently expensive task. Program slicing [8, 21, 42, 65, 75] has been used to reduce this expense, by focusing attention on a particular 'slice' of the source code [12, 15, 17, 36, 51]. Slices are constructed using dependence analysis. The lower the dependence level, the smaller the slice. Using slicing can therefore reduce human analysis effort.

Automated analysis can be expensive too. For example, consider the automated search for branch-coverage adequate test data. Current tools do not integrate dependence analysis [47, 56, 67, 70]. However, the potential savings from dependence analysis are dramatic; *linear* dependence level reductions cause *exponential* search space reductions.

This paper presents the results of an empirical investigation into the nature of program dependence for a set of twenty C programs. The goal of the study was to determine typical levels of dependence, to identify dependence trends, correlations and patterns, and to provide scalable visualization techniques for depicting the dependencies present in large programs.

The study focuses upon the extent to which a program's predicates depend upon two of their potential influencors — the global variables in scope and the formal parameters in the function or procedure which contains the predicate. Predicates are chosen, because they allow dependence to be studied at a detailed and fine level of granularity, while results obtained refer to a semantically important aspect of the computation — the decision logic and control flow of the program.

The principal contributions of the paper can be summarized as follows:

- Results for dependence levels across 50,579 predicates indicate that a typical predicate depends upon approximately

  - 72% of formal parameters in the scope,
  - 48% of the global variables potentially used or defined by a call to the predicate's procedure, and
  - 2.4% of the global variables in scope.

  These results are important for dependence–sensitive activities such as test data generation [16, 22, 31, 52, 53, 56, 67, 70] (where the apparent size of the input space can be reduced by dependence analysis) and program comprehension [6, 13, 33, 39, 55, 46] (where distinguishing true dependencies from apparent dependencies removes an unnecessary cognitive burden).

- The study provides evidence of a strong *inverse* correlation between the number of formal parameters visible to a predicate and the proportion actually used. There is no such correlation for globals.

- The paper investigates correlations between sources and targets of dependence, showing that the dependence sources considered are independent of one another and that, in general, the number of formals available to a predicate is not dependent upon the size of the function in which the predicate occurs.

- Visualizations are introduced for human analysis and interpretation of dependent proportions. The visualizations summarize dependence information about predicates, allowing trends and patterns to be identified.

- The paper illustrates ways in which the visualizations can be used to guide program comprehension and the human evaluation of software evolution. The paper also shows how the analysis can assist with reducing automated test data generation effort.

- Finally, the paper presents performance results from the implementation of the dependence analysis which show that the approach scales to medium/large programs, making it practical as a support for comprehension, test data generation, maintenance, and evolution.

The rest of the paper is organized as follows: Section 2 outlines the experimental setup and Section 3 describes the two visualizations used to view dependencies for large programs within a single graph. Sections 4 and 5 describe dependence ratios, levels and trends over all predicates studied, and correlations between them. Section 6 shows how the visualizations can help with program comprehension and evolution and how the predicate dependence analysis introduced in this paper can assist with automated test data generation. Section 7 considers threats to validity of the findings. Section 8 shows that dependence analysis is practical with regard to both its theoretical complexity and the actual computation time for the implementation used in the study. Section 9 presents related work. Finally, Section 10 summarizes the paper.

2 Experimental Setup

This section describes the experiential context, and the experiment's instrumentation including the statistical tests used and the data collection process.

Name	Size KLOC	Predicates	Predicates per KLOC	Description
replace	0.6	84	140	Regular expression string replacement
copia	1.2	45	38	ESA signal processing code
compress	1.9	88	46	Data compression utility
which	5.4	136	25	Unix utility
barcode	5.9	365	62	Barcode generator
space	11.5	593	52	ESA ADL interpreter
ed	13.6	2627	193	Unix editor
prepro	14.8	619	42	ESA array pre-processing code
bc	16.8	432	26	Calculator
findutils	18.6	1819	98	File finding utilities
diffutils	19.8	2212	112	File comparing routines
flex2-4-7	21.2	911	43	BSD scanner generator (version 2.4.7)
flex2-5-4	21.5	1306	61	BSD scanner generator (version 2.5.4)
espresso	22.1	2497	113	logic simplification for CAD (from SPECmark)
ijpeg	28.2	2038	72	JPEG compressor (from SPECmark)
ntpd	45.6	3061	67	daemon for the network time protocol
a2ps	62.9	5984	95	postscript formatter
gnugo	81.7	5244	64	Gnu game player
sendmail	85.7	4991	23	Linux mail daemon
spice	167.0	15527	93	Digital circuit simulator
average	28.3	2529	73	

Figure 1: Subject programs.

2.1 Experiment Context

The study concerns 20 C language subject programs that range in size from 600 LOC to 167K LOC. Three programs were obtained from the European Space Agency (ESA), while the remaining 17 consist of publicly available utilities and application programs. Figure 1 presents summary information concerning the subjects.

2.2 Instrumentation

The study uses two statistical tests: Pearson's linear correlation and Spearman's rank correlation. Pearson's linear correlation is used to construct quantitative models of the data. Such correlations measure linear associations between variables. The output is a *correlation* *coefficient*, reported as the value R, and the coefficients of a linear *model*. The statistical significance of R can be summarized as 0.8 - 1.0 strong association, 0.5 - 0.8 moderate association, and 0.0 - 0.5 weak or no association. A negative value indicates an inverse correlation. For the effect of X, Y, and Z on A, the resulting model coefficients, m_i, belong to the linear equation

$$A = m_1X + m_2Y + m_3Z + b.$$

The statistical confidence associated with each coefficient is denoted by a t-score; a t-score of 1.94 or greater is considered significant. Spearman's rank correlation is also used. The Spearman test does not attempt to construct a linear correlation between the two variables, but merely gives a correlation value which indicates whether increases in one variable are accompanied by increases in the other.

The dependence data was collected by a modified version of the standard System Dependence Graph (SDG) backward interprocedural slicing algorithm [29]. An SDG represents a program as a collection of Procedure Dependence Graphs (PDGs), the nodes of which represent statements and predicates and the edges of which represent control and data-dependences between nodes. Interprocedural dependence edges connect PDGs to form the SDG. The final step in the construction of the SDG is the addition of '*summary*' edges at call-sites. These transitive dependence edges represent certain paths of dependence edges through called procedures from nodes representing initial parameter values to nodes representing final parameter values.

Slicing 'at a node' in the SDG means finding all the statements which potentially affect the computation represented at that node. In terms of Weiser's definition of a slicing criterion, slicing at an SDG node n means slicing on (V, n), where the set of variables, V is the set of variables referenced at n.

The formal parameters and global variables that potentially affect a branch node b of procedure P can be computed by slicing with respect to the node for b and reporting the formal parameters and global variables whose nodes are in P's PDG and the slice. It turns

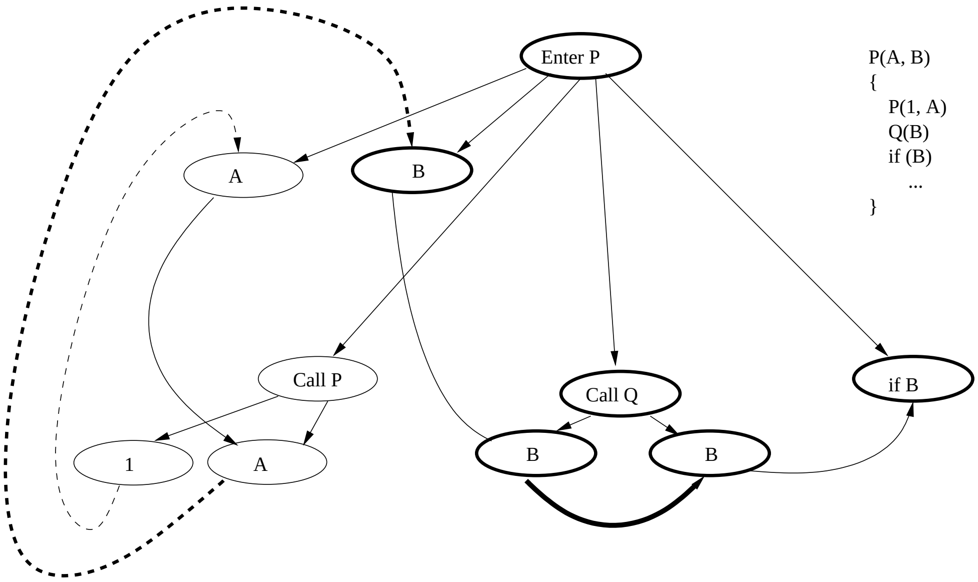


Figure 2: Local slice computation example.

out that this approach is inefficient because it needlessly considers components from other procedures. It is also a mild over approximation in the presence of recursive procedures. Therefore, the slicer was modified to compute 'local' slices [29], which only consider intraprocedural edges when slicing. Local slices safely identify those parameters that affect a predicate on a particular invocation of the procedure.

Example. Figure 2 illustrates the time saving and increased precision resulting from the use of local slices. Only the relevant subgraph of `P`'s PDG is shown (*e.g.*, the output of `P` is not shown). Consider the slice on "`if B`". The local slice used to gather predicate dependence includes the bold vertices. The time saving is illustrated by this slice as it traverses the summary edge (bold edge) associated with the call on procedure `Q`. By contrast, (pass 2 of) a full interprocedural slice would include elements from `Q` and incur the cost of discovering them. The increased precision is illustrated by the absence of formal parameter `A` in the local slice. A 'full' interprocedural backward slice, would (during pass 1), traverse the bold interprocedural edge and eventually encounter the node for formal parameter `A`.

For completeness, some details of the operation of CodeSurfer when performing the dataflow analysis used to build SDGs are presented. First, the pointer analysis algorithm used by CodeSurfer is a modification of Andersen's algorithm [2]. The modification maintains the same precision while decreasing the time and space requirements [18]. The result is a flow and context insensitive points-to analysis that is almost as fast as Steensgaard's algorithm [63], but provides a more precise answer [61]. Unfortunately, flow or context sensitive points-to analysis is hard [26, 54, 77] and, at present, does not scale well. There is also empirical evidence that, for program slicing, improved quality points-to sets do not have a significant impact on slice size [11, 49].

Second, it has been shown that structure–field treatment has a potential impact on slice size [76]. This is best understood by way of example. Consider the following `C` code:

```
struct file_ops
{
    int (*write_fn)();
    int (*read_fn)();
} file1;
```

Assume `write_fn` potentially points to the functions `write_ext2` and `write_nfs` and `read_fn` potentially points to the functions `read_ext2` and `read_nfs`. With structure fields expanded, the function call `(*file1.readfn)(···)` potentially calls `read_ext2` and `read_nfs`. With structure fields collapsed, all structure fields are treaded as a single variable and the call `(*file1.readfn)(···)` potentially calls `write_ext2` and `write_nfs` in addition to `read_ext2` and `read_nfs`. For programs with structures that contain pointers, and especially function pointers, this collapse can significantly increase slice size.

Arrays are passed to functions as whole structures. In common with other static analysis techniques, it is not possible (as it is in dynamic analysis) to consider individual array elements separately. Rather, an access to any array element is effectively treated as a may access to all elements.

For a predicate p, the *dependent proportion* refers to the proportion of *parameters* that p depends upon. For two predicates in the same function (*i.e.*, two predicates having the same number of parameters available) a lower dependent proportion implies dependence on fewer parameters. In this definition, the term *parameter* refers to formal parameters and global variables. Global variables are further broken down into *visible globals*, those in scope at p, and *du-globals*, the subset of the visible globals potentially defined or used by a call to the function that contains p. This distinction between du-globals and visible globals is interesting because du-globals represent a first approximation to those globals upon which p depends. As will be seen, this first approximation is rather crude.

3 Visualization Techniques for the Dependent Proportion

Two visualization techniques are introduced to present dependence information. In order to visualize dependent proportions, three variables must be depicted:

Predicate Count. The number of predicates summarized by a data point. Data points summarizing a larger number predicates carry more weight.

Max-Parameters. The maximum number of parameters in scope (visible) at a predicate. Large maxima are more interesting because of the greater potential for savings from using dependence analysis.

Parameters-Used. The number of parameters that affect a predicate according to the dependence analysis described in Section 2.2. The smaller this number the better for dependence-sensitive applications.

In these definitions "parameters" refers to formals and globals taken together. When only formals are being considered the terms *Max-Parameters* and *Parameters-Used* become *Max-Formals* and *Formals-Used*. Similar specialization is done when only globals are considered.

It is important to understand the difference between max-parameters and parameters-used. In the absence of any dependence analysis, *every* formal parameter and *every* global variable must be considered to potentially affect a predicate. Dependence analysis can reduce this number, but by its very nature, must be conservative: determining whether a formal definitely affects a predicate is undecidable. Therefore, a safe (conservative) approximation is computed; parameters-used excludes only those parameters that can be *guaranteed* not to affect a predicate.

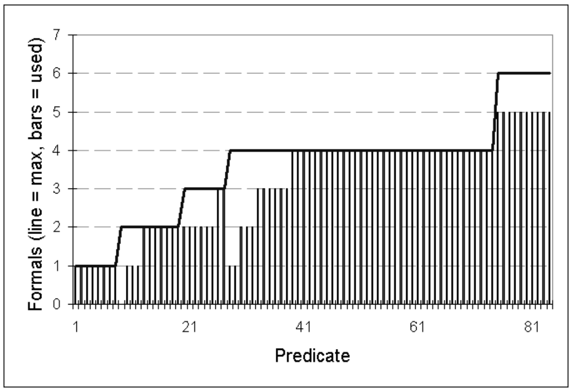
3.1 Dependent proportion visualizations

To visualize the data, two graphical devices are used: the *Dependence Skyline* and the *Dependence Bubble Chart*. This section describes these visualizations and illustrates their interpretation for two of the smaller programs. Figure 3 shows formal parameter dependence data for the programs `replace` and `prepro`. The column labeled 'Predicate Count' indicates the number of predicates summarized in each row of the table. For example, consider the third row (marked '(*)' in Figure 3). This row summarizes predicates in procedures with three formal parameters. Such predicates, by definition, are influenced by at most three formal parameters. As shown in the 'Predicate Count' column, there are 8 such predicates. For these predicates, the average number of formals depended upon (shown in the column labeled 'Average Formals Used') is 2.2. Compare this row with the row immediately below it in the table. For predicates with 4 formal parameters in scope, an average of 3.6 actually affect the predicates. At first glance, this row seems to be telling a similar story: most of the formal parameters affect each predicate. However, Row 4 is more interesting than Row 3 because it summarizes almost six times as many predicates. This highlights the importance of the predicate count attribute.

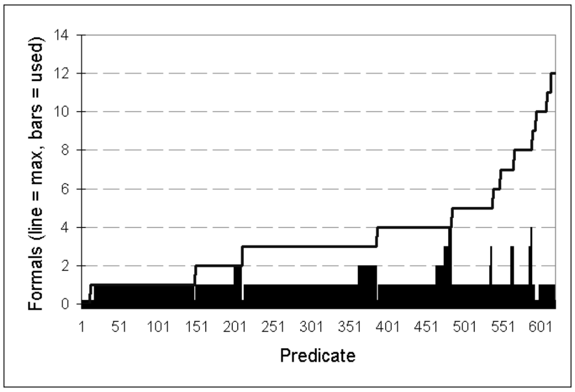
The raw data summarized in Figure 3 is shown in graphical form using dependence skylines in Figure 4. A dependence skyline shows each predicate on the horizontal axis (sorted by max-parameters and, within that, sorted by parameters-used). The vertical axis denotes usage of parameters (both the theoretical maximum and the actual count revealed by dependence analysis). The 'horizon line' which monotonically increases as it travels left to right across the top of the dependence skyline indicates the value of max-parameters,

Program	Max Formals	Average Formals Used	Predicate Count
replace	1	1.0	8
	2	1.5	11
(*)	3	2.2	8
	4	3.6	47
	6	5.0	10
prepro	0	0.0	11
	1	1.0	137
	2	1.2	62
	3	1.1	176
	4	1.3	98
	5	1.1	53
	6	1.0	10
	7	1.6	18
	8	1.4	24
	9	1.0	4
	10	0.7	15
	11	1.0	5
	12	1.0	6

Figure 3: Example formals data.

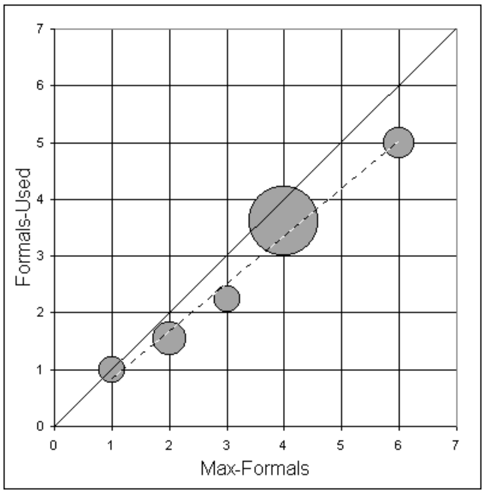


`replace` Dependence Skyline

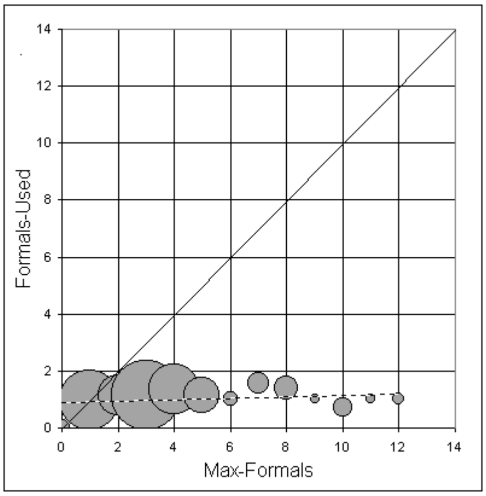


`prepro` Dependence Skyline

Figure 4: Dependence skylines. Bars indicate actual predicate dependence. The horizon line running across the top of the dependence skyline is the 'theoretical maximum' – the number of formals in scope. Predicates are ordered left-right by Max-Formals. The space between these two indicates ignore-able formal parameters.



`replace` Dependence Bubble Chart



`prepro` Dependence Bubble Chart

Figure 5: Dependence bubble charts. In these dependence bubble charts, bubble size indicates predicate-count. The solid line at 45^o indicates the maximum possible dependence (formals used = max formals). The further the bubbles are from this line, the greater saving analysis can produce. The dotted line indicates the linear trend (unweighted by bubble size).

while the usage bars indicate the value of parameters-used.

For example, the data for predicates in procedures with 3 formal parameters (marked with '(*)' in Figure 3) appears in the dependence skyline at points with X values 20 through 27 in the left of Figure 4. Over this range, the horizon line (showing max-formals) is horizontal and represents all predicates with three visible formal parameters. The first six bars in this range have height 2, while the last two have height 3. (This gives the average height of 2.2 as shown in Figure 3.) Each bar in a dependence skyline diagram summarizes a single static backward slice from a single predicate.

With a small example like `replace`, the dependence skyline replicates the tabular data in full detail. However, for a larger example, like `prepro`, the dependence skyline provides only a general *feel* for the raw data; it is unrealistic to separate out individual bars due to the large number of data points.

For larger data sets, dependence bubble charts can often better display the three dependent attributes. In such charts, the horizontal axis denotes max-parameters, while the vertical axis denotes parameters-used. Thus, the line $y = x$ represents the worst case (*i.e.*, a predicate can reference no more than the visible parameters). For reference, this line is drawn as the solid line in all dependence bubble charts. Also drawn is a dashed line that represents the linear trend (computed using a linear least squares fit). Finally, bubble size represents predicate-count.

For example, consider `replace`'s dependence bubble chart for formal parameters shown on the left of Figure 5. The largest bubble summarizes the data from Row 4 of the table in Figure 3. Here, the average number of formals used by all predicates from `replace` with 4 formals in scope is just under 4. Recall from Figure 3, the actual value is 3.6. Also, as can be quickly seen, this bubble is considerably larger than the others. In fact, it summarizes over half of the predicates in the program. The second largest bubble represents predicates in procedures with two formals (visually, these can be seen to depend upon an average of approximately 1.5 of the 2 available formal parameters).

Compare this with the `prepro` dependence bubble chart, which appears on the right of Figure 5. Three important facts can be readily seen. First, divergence from the solid $y = x$ line very clearly highlights the *tremendous savings* which occur from dependence analysis of `prepro`. Second, the relative size of the bubbles indicates that most of this saving is to be expected in procedures with fewer than 5 formal parameters. Third, the dashed trend line indicates that formals-used is effectively constant (independent of the number of formals available). This trend line may have important implications for subsequent maintenance and evolution of `prepro`.

The dependence bubble charts and skylines for formal parameter dependence of the twenty programs studied are depicted in Figures 6 and 7. The dependence bubble charts and skylines for du-global dependence of the twenty programs studied are depicted in Figures 8 and 9. Results for visible global dependence are less interesting and so these are only summarized at the end of the next section.

4 Trends in Dependence Over all Predicates

One of the primary goals of this study is to determine levels, ratios, and trends in predicate dependence. This section reports key findings over all 50,579 predicates considered. Three related questions are asked that inquire about dependent proportion between predicates and first formals and then the two kinds of globals. The first question to be addressed is

> *'How does the dependent proportion change as the number of available formal parameters increases?'*

As seen in Section 4.2, the answer, put simply, is that

> *'As the number of available formals increases, the proportion of these depended upon drops.'*

This is an important and optimism-inducing finding. For example, impossibly large search spaces for test data generators may present more tractable problems than previously thought.

A similar question can be asked for globals. Sadly (but importantly), it cannot be said. Not for visible globals nor for du-globals. This provides some evidence for the assertion:

> *'Global variables are harmful.'*

That is, programs with a large number of global variables may present intractable problems for dependence-sensitive applications. There is a "silver lining" here: the percentage of visible globals used by a typical predicate is relatively low.

The data for formals is piecewise linear. Piecewise linear data can be analyzed by introducing a new variable that captures the pivot. For example, with a pivot of 10, this

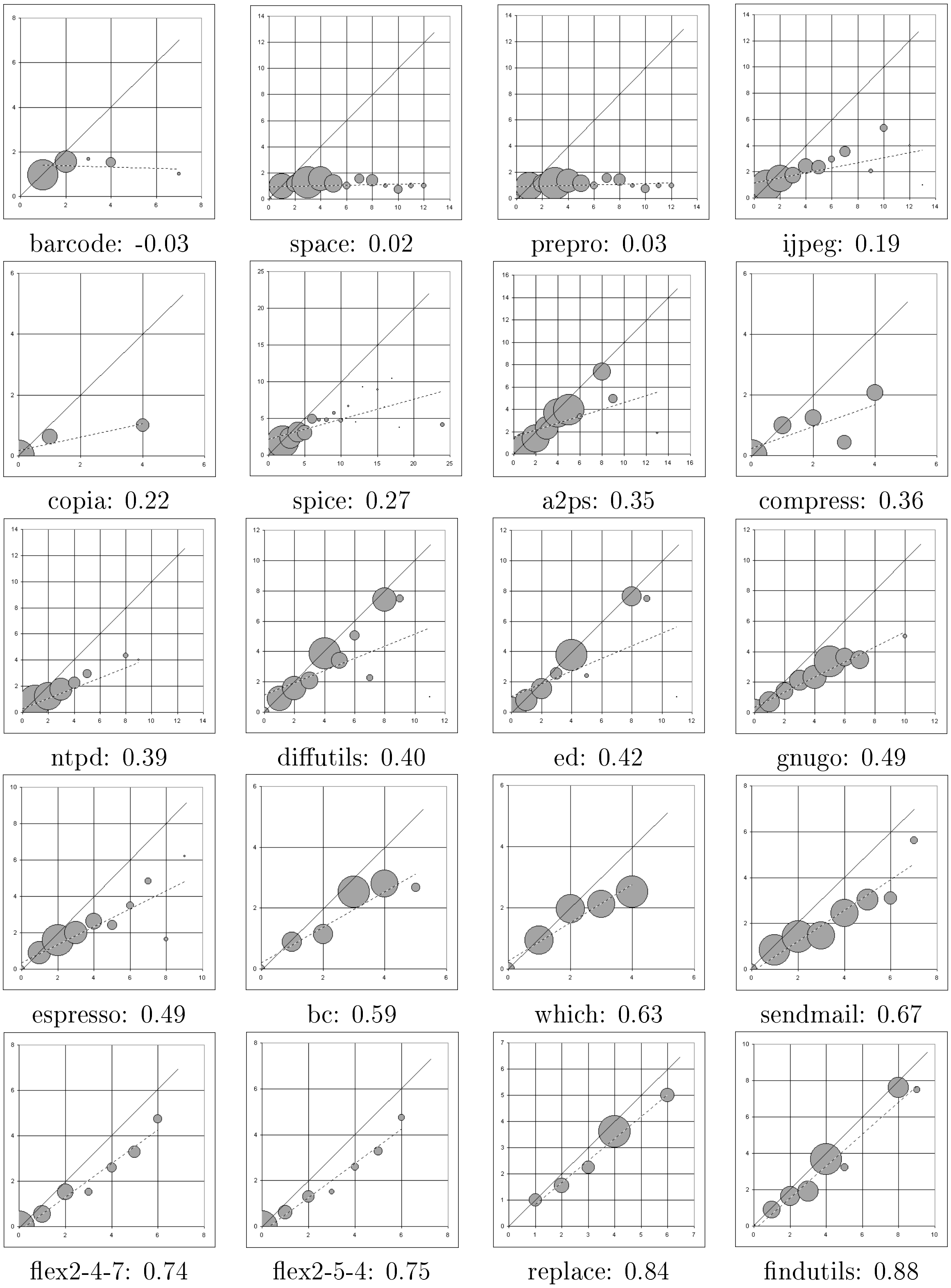
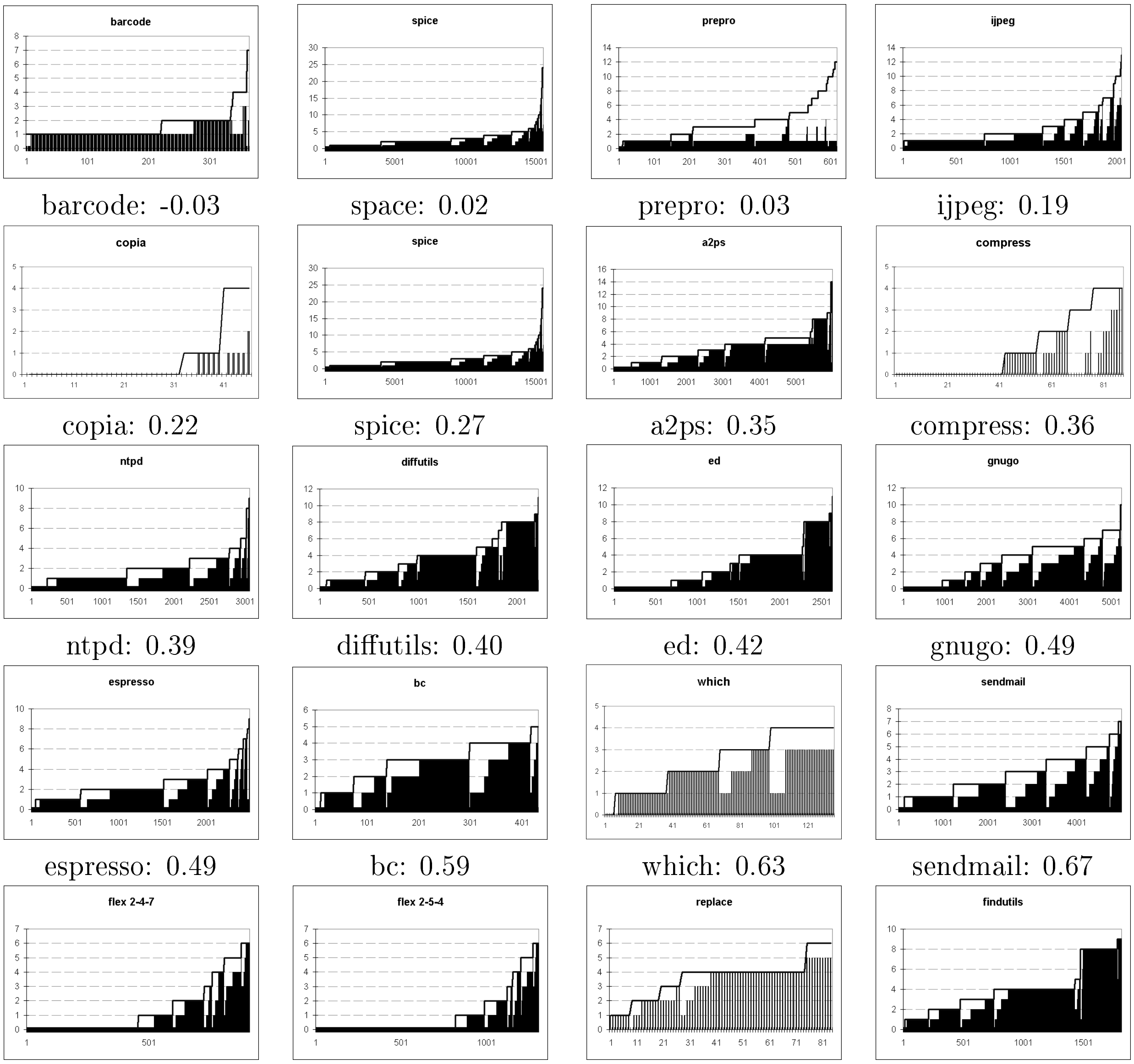


Figure 6: Dependence bubble charts for formal parameter dependence. The number appearing after the name of each program is the slope of the dependence trend.



barcode: -0.03 space: 0.02 prepro: 0.03 ijpeg: 0.19

copia: 0.22 spice: 0.27 a2ps: 0.35 compress: 0.36

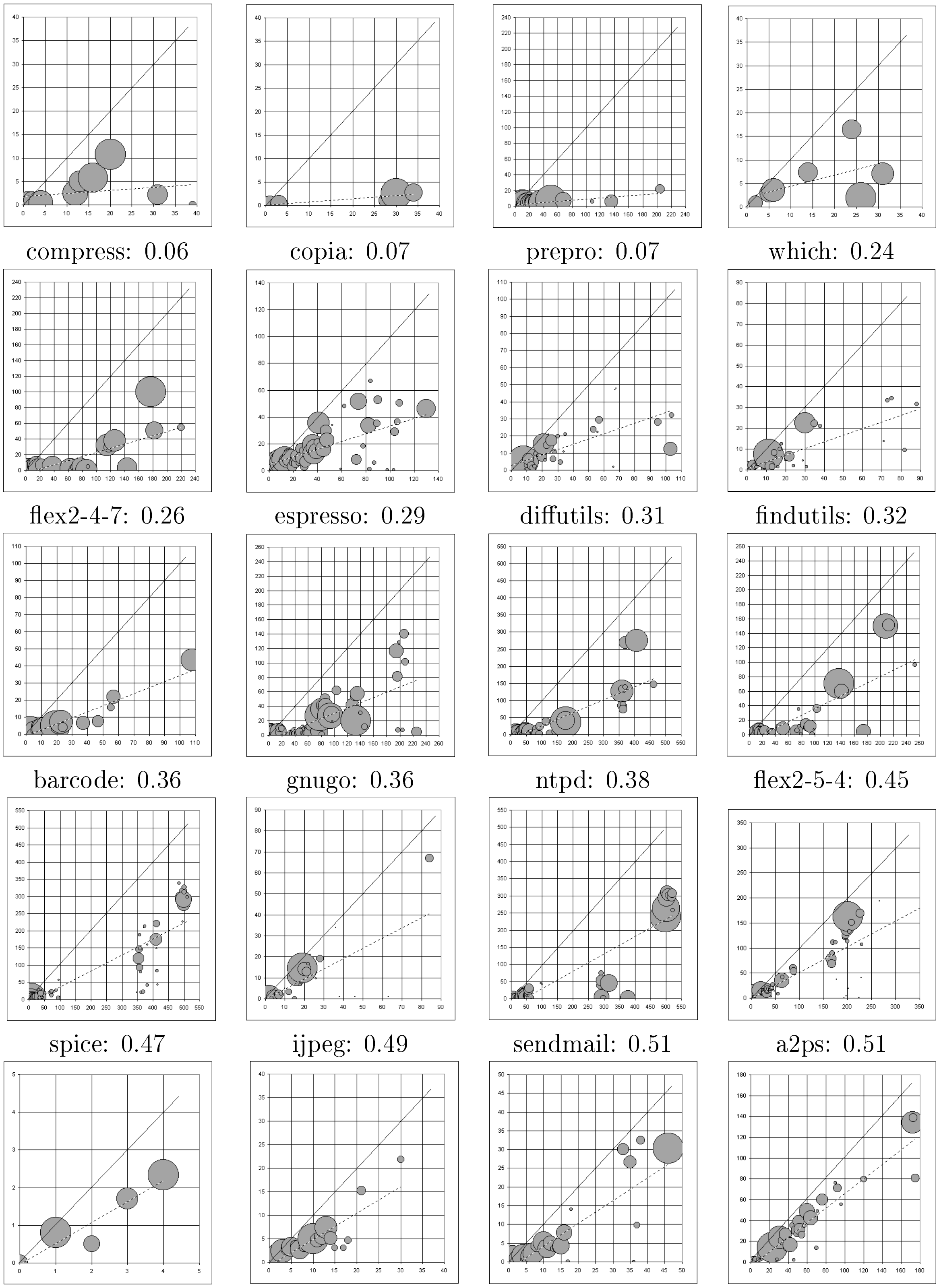
ntpd: 0.39 diffutils: 0.40 ed: 0.42 gnugo: 0.49

espresso: 0.49 bc: 0.59 which: 0.63 sendmail: 0.67

flex2-4-7: 0.74 flex2-5-4: 0.75 replace: 0.84 findutils: 0.88

Figure 7: Dependence skyline for formal parameter dependence. The number appearing after the name of each program is the slope of the dependence trend.

new variables has the value 0 for the first 10 observations, and has the value $x - 10$ for all other observations. Thus, this new variable separates out the effect of the data after the pivot. In this case, the highest R-values are for pivots points of 8 and 10 ($R = 0.96$ and



compress: 0.06 | copia: 0.07 | prepro: 0.07 | which: 0.24

flex2-4-7: 0.26 | espresso: 0.29 | diffutils: 0.31 | findutils: 0.32

barcode: 0.36 | gnugo: 0.36 | ntpd: 0.38 | flex2-5-4: 0.45

spice: 0.47 | ijpeg: 0.49 | sendmail: 0.51 | a2ps: 0.51

replace : 0.56 | space: 0.56 | bc: 0.61 | ed: 0.70

Figure 8: Dependence bubble charts for du-global dependence. The number appearing after the name of each program is the slope of the dependence trend.

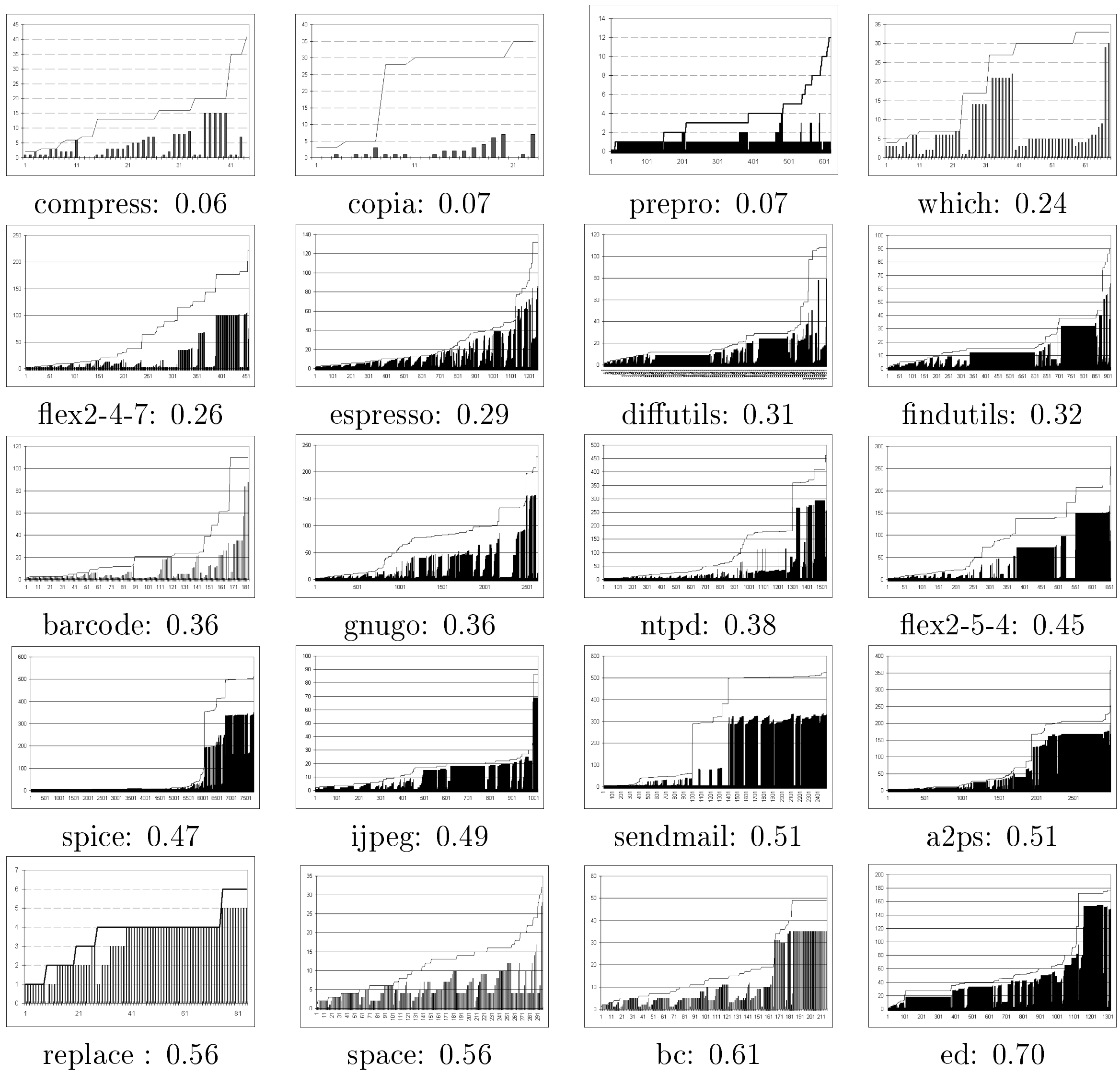
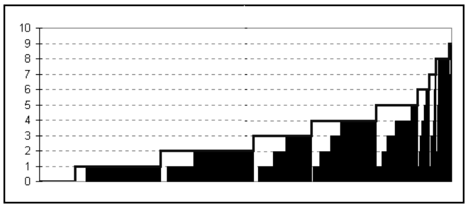


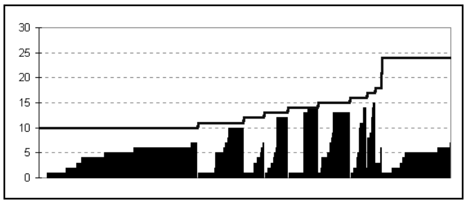
Figure 9: Dependence skylines for du-global dependence. The number appearing after the name of each program is the slope of the dependence trend.

$R = 0.95$, respectively). There is little to choose between these two. Visually the split starts at max-formals of 10, where the figures clearly show different dependence profiles.

For this reason, the following visualizations are split into two diagrams; one for values of max-formals less than 10 and one for max-formals at least 10. This piecewise linearity is starkly evident in the trend lines of the two dependence bubble charts. The following



Max-Formals < 10



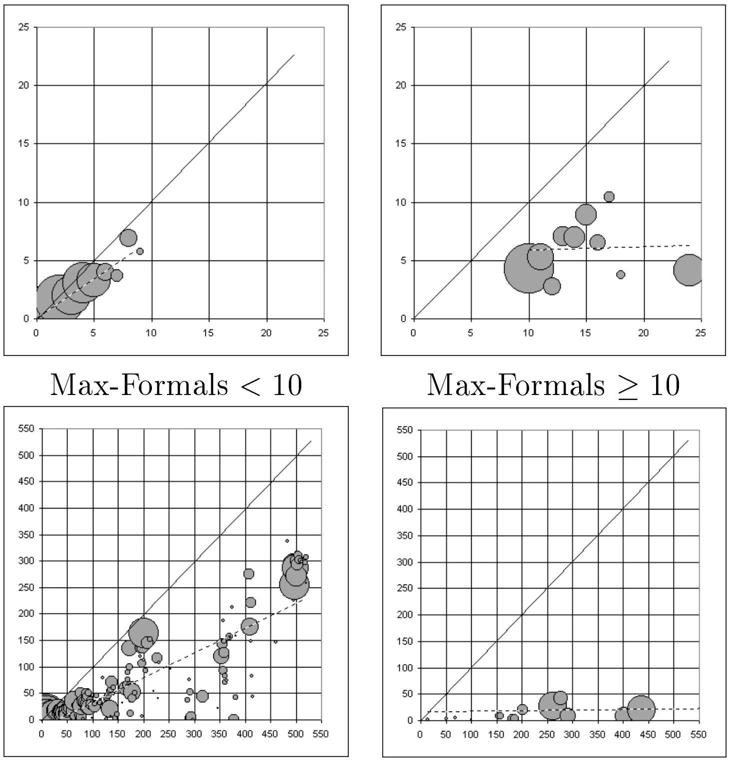
Max-Formals ≥ 10

Figure 10: Dependence skylines for all 50,579 predicates.

two equations separate out the two pieces. For values of max-formals below 10 the model is $\textit{formals-used} = 0.14 + 0.68 \times \textit{max-formals}$. For values of max-formals of at least 10 the model is $\textit{formals-used} = 8.54 - 0.16 \times \textit{max-formals}$. In essence, the number of formals-used increases linearly at roughly $0.68 \times \textit{max-formals}$ until max-formals is 10, where formals-used begins a slight decline, but is rather independent of max-formals.

The decreasing dependence on formal parameters, can be readily appreciated visually in both dependence skylines and dependence bubble charts. The dependence skyline for all 50,579 predicates is shown in Figure 10. The corresponding dependence bubble charts are shown at the top of Figure 11.

When considering du-globals, the trend line for globals is just below the '$y = x/2$' line and reflects the 48% average usage (see the lower left of Figure 11). The resulting Pearson model is $\textit{globals-used} = 0.47 \times \textit{max-du-globals} - 13$ with an R value of 0.83 (a strong linear association). For visible globals the R value of 0.40 indicates there is no linear correlation. Thus, there are two possibilities. The more likely case is that max-visible-globals is simply not a good predictor of globals-used; however, it is it is possible (but unlikely considering the visualization of the data shown in the lower right of Figure 11) that some non-linear relationship exists. These two models for globals are important as they show that dependent proportions do not fall as the number of available globals increases.

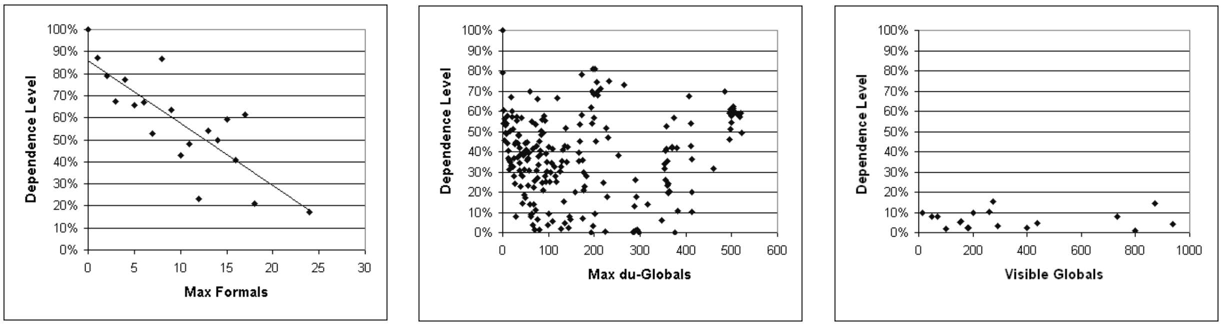


Max-Formals < 10 | Max-Formals ≥ 10

All du-Globals | All visible Globals

Figure 11: Bubble charts for all 50579 predicates. Note: scale of bubble size is different in each chart.

Finally, consider the relationship between max and *dependent proportion* (*i.e.*, percentage used). Scatter plots showing this relationship for formals, du-globals, and visible-globals appear in Figure 12. Over all 20 programs, there is strong evidence for an inverse linear correlation between max-formals and dependence level. The resulting trend line, which has an R value of -0.833, is captured by *dependent proportion* $= 86\% - 2.8\% \times$ *max-formals*. While this linear equation better explains the data, the average dependent proportion for formals is about 72%. These linear equations were produced using unweighted averages. The fit is almost identical when weighted averages are used (*dependent proportion* $= 85\% - 2.7\% \times$ *max-formals*). However, the R value for the weighted average is slightly lower, -0.66 rather than -0.833.



Dependent Proportion for Formals | Dependent Proportion for du Globals | Dependent Proportion visible Globals

Figure 12: Predicate dependence trends.

No linear correlations exist for globals. When considering du-globals (center chart in Figure 12) R has the values 0.080. For all visible globals (lower chart in Figure 12) it has the value -0.27. Since there is no linear correlation, the averages become more interesting for globals than they are for formals (where a good linear fit was found). On average 48% of du-globals and 2.4% of visible globals are used.

5 Correlations Between the Variables Studied

This section presents the results of various statistical tests to determine whether the values of interest in this study are correlated. The results tend to suggest that most of the variables of interest are not correlated to one–another. This is important, because it means that the results from Section 4 *really do* suggest that the nature of dependence of predicates upon formals is qualitatively and quantitatively different from the nature of dependence of predicates upon globals.

Tests were run for correlations between the numbers of nodes, edges, predicates, formals, and du-globals found in each procedure. Over all 20 programs studied, there were 7,191

X	vs.	Y	Pearson R Value	Spearman R Value
Nodes	vs.	Edges	**0.714**	**0.987**
Nodes	vs.	predicates	0.397	*0.663*
Nodes	vs.	formals	0.060	0.409
Nodes	vs.	du-globals	0.475	**0.840**
Nodes	vs.	vis-globals	0.063	0.198
Nodes	vs.	f+du-globals+vis-globals	0.086	0.321
Nodes	vs.	f+du-globals	0.476	**0.866**
Nodes	vs.	f+vis-globals	0.063	0.234
Nodes	vs.	du-globals+vis-globals	0.086	0.322
predicates	vs.	formals	0.117	0.374
predicates	vs.	du-globals	0.165	0.417
predicates	vs.	vis-globals	0.062	0.247
predicates	vs.	f+du-globals+vis-globals	0.070	0.301
predicates	vs.	f+du-globals	0.167	0.419
predicates	vs.	f+vis-globals	0.063	0.276
predicates	vs.	du-globals+vis-globals	0.070	0.304
formals	vs.	du-globals	0.064	0.247
formals	vs.	vis-globals	0.135	0.195
formals	vs.	du-globals+vis-globals	0.137	0.222
du-globals	vs.	vis-globals	0.177	0.059
du-globals	vs.	f+vis-globals	0.177	0.071
vis-globals	vs.	f+du-globals	0.179	0.096

Figure 13: Pearson/Spearman correlations (bold values are strongly significant, italic values are weakly significant).

procedures, so each test concerns 7,191 data points. Both Spearman Rank Correlations and Pearson Linear Correlations were used. The only significant correlations found are those where (as one would expect) increases in the size of the procedure led to a corresponding increase in the number of predicates and the number of du-globals. Interestingly, there is no evidence to suggest that increases in the size of a procedure are accompanied by corresponding increases in the number of formals.

Figure 13 shows the R scores for Pearson and Spearman tests. Using Pearson's test, the only linear correlation (having an R score greater then 0.50) was between the number of nodes and edges in a PDG ($R = 0.714$). Because nodes and edges are correlated the remaining tests are reported for nodes only. The values for edges are quite similar; however as the number of edges includes some measure of the level of dependence in the procedure and not just its size, the number of nodes is a better measure of the size of the procedure.

Using the Spearman test, in addition to the correlation between nodes and edges, three other correlations emerge. First, the number of predicates increases with the size of the procedure, but, as the Pearson R value is low, this relation is probably not a linear one. The remaining two relate the number of nodes to the number of du-globals and to the sum of the number of du-globals and the number of formals. In the final case, the number of du-globals being typically far larger than the number of formals dominates the statistics. The correlation with du-globals may seem surprising at first glance. However, du-globals are, by definition, "defined or used" in the procedure and thus roughly track its size.

Many of the tests considered in this section resulted in no correlation (neither strong nor weak being found). In many cases the absence of a correlation (non-bold values in Figure 13) is just as important as the presence of one. These results establish that the number of formal parameters is not correlated to the size of the procedure and that the number of formals and du-globals (the two primary dependence sources studied) are independent of one–another. The observation of an absence of correlation between the number of formals in a procedure and the number of globals led to a set of further hypotheses about the relationship between the dependence sources available to a predicate (max-formals, max-globals and max-visible-globals). No correlations were found, so a predicate which has larger than average number of formals available to it does not necessarily have a larger than average number of du-globals available, etc. This provides evidence that these two dependence sources are independent of each other. The results over all predicates in all programs are shown in Figure 14.

A test was also carried out for correlations between the number of formals and the number

X	Y	Pearson R Value	Spearman R Value
Formals available	du-globals available	0.124	0.23
Formals available	visible globals available	0.008	0.27
du-globals available	visible globals available	0.0035	0.21

Figure 14: Pearson/Spearman tests for dependence sources available to a predicate showing no correlation.

of du-globals available to a predicate in each program in isolation[1]. The results are shown in Figure 15. Here, there does exist evidence of weak correlations for a few programs, but there is little compelling evidence for a trend.

In addition to asking whether the available numbers of each kind of dependence source are correlated, it is also sensible to ask whether the number actually used by a predicate are correlated. For example, it might be that, while the max-formals and the max-du-globals are not correlated, the number that the predicate uses are. If detected, this would suggest that predicates can be thought of as more or less dependent upon dependence sources *per se*, and that the distinction between particular *kinds* of dependence source is less important. Once again, however, both Spearman and Pearson tests show no correlations, yielding further evidence for the claim that the kinds of dependence source are independent of each other. The results are presented in Figure 16.

The overall conclusion from this series of investigations into correlations between formals and globals, is that the evidence all points to the finding that sources of dependence from formal parameters are wholly different in their nature and behaviour from sources of dependence from globals. Combined with the findings from Section 4, it is reasonable to conclude that there is a qualitative and quantitative difference between formal and global dependence.

[1] The number of visible globals is essentially identical for all predicates in a program, so questions involving visible globals are not meaningful when considering each program independently.

Program	Pearson R Value	Spearman R Value
`a2ps`	0.156	0.361
`barcode`	0.530	0.844
`bc`	0.274	0.561
`compress`	0.055	-0.070
`copia`	-0.396	0.524
`diffutils`	0.146	0.460
`ed`	0.248	0.764
`espresso`	0.347	0.745
`findutils`	0.354	0.537
`flex`	-0.047	0.110
`gnugo`	0.209	0.293
`ijpeg`	-0.137	-0.100
`ntpd`	0.090	0.263
`prepro`	0.181	0.648
`replace`	0.542	0.532
`sendmail`	0.326	0.389
`space`	0.628	0.750
`spice`	-0.036	-0.036
`which`	0.571	0.443

Figure 15: Pearson/Spearman tests for dependence sources available to a predicate for each program in isolation.

X	Y	Pearson R Value	Spearman R Value
Formals Used	du-globals Used	0.0081	0.002
Formals Used	visible globals Used	0.088	0.032
du-globals Used	visible globals Used	0.082	0.221

Figure 16: Pearson/Spearman tests for dependence sources used by a predicate showing no correlation.

6 Applications for Dependence Visualizations

This section illustrates the application of the dependence analysis and the visualizations introduced in the paper. Applications that potentially benefit from dependence analysis and visulazations include program comprehension including understanding software evolution [6, 33, 39, 15], software surgery [17], testing including testability transformations and automatic structural testing, [16, 22, 52, 56, 70] array safety analysis, [12], and program flaw identification [67]. Three of these are illustrated in this section: Section 6.1 shows how the analysis of a predicate's dependence on formal parameters can be used to reduce test data generation effort in evolutionary testing. Sections 6.2 and 6.3 show the application of the visualizations as support and guidance for evolution and comprehension.

6.1 Application to search–based test data generation

Search–based test data generation uses search techniques (such as genetic algorithms [27, 48]), to search for test data which meets some chosen test objective. The test objective is translated into a fitness function which guides the search, by determining when one test case is closer to meeting the objective than another.

More details about search–based test data generation can be found elsewhere [31, 32, 47, 50, 56, 57, 68, 71, 72]. For the work reported here, the important aspect of the technique is that it is search based. The space of possible inputs to the program or procedure constitutes the search space from which search-based testing attempts to locate a suitable test case. Consider, for example, the case of branch coverage. The test objective is to find an input that executes some branch of interest in the software under test. The search space is formed from the set of global variables and formal parameters available to the branch. For some chosen branch, controlled by a predicate p, not all of the search space will be relevant. Those global variables and formal parameters that cannot affect the value computed at p can be safely excluded from the search. In this way, the analysis used in the present paper can be used to augment search based testing by reducing the search space size.

To explore the potential of this possibility, a simple experiment was conducted, using the DaimlerChrysler Evolutionary Testing System [70]. The system uses genetic algorithms to search for test data that meet a variety of criteria. For this experiment, the system was configured to attempt branch coverage for the program shown in Figure 17. The procedure takes, as input, 16 values. The first six (in variables `a` to `f`) affect the path through the procedure. The remainder (in the array `du`) do not. By varying the size of `du` the effect of search space size can be investigated. In this case, the size of 0 and 10 provide the ratio 6 to 16, which approximates the average reduction for du-globals found in Section 3. The version with 16 input values has a larger search space, which potentially increases the test effort required to find the 'right' input values to hit all branches. Predicate dependence analysis would determine that only the 6 formals `a` to `f` need be considered, thereby reducing the search space size and (hopefully) reducing test effort. To explore this potential, test data were generated for the 6 and 16 input versions of the procedure. Finally, observe that in this example, each predicate is dependent on 2 of the parameters. If however, for example, the second predicate were nested within the first, then it would be dependent on 4 parameters because the slice would include the first predicate (via a control dependence edge).

Evolutionary algorithms are non-deterministic and contain some randomness in their behaviour. To ensure that the results obtained were robust, data were collected from ten separate trial runs of the test data generation system for each of the two versions of the program. The results are shown in Figure 18. As can be seen, the optimized search (with the reduced search space) requires far fewer fitness evaluations. Here the analysis of predicate dependence assists with the discovery of branch adequate test data generation. Even over all 10 runs, the worst case effort for the optimized version (the tallest gray bar) is far better than the best case for the unoptimized version (the shortest white bar).

Of course, this initial set of results indicates merely that the approach is feasible and promising. A more detailed investigation of the application of predicate dependence analysis to search–based testing is the topic of ongoing and future work. However the initial results

```
void testfunc(int a,int b, int c, int d, int e,int f, int du[10])
{
  if (a==100000 || b==100000)
    a++;
  if (c==100000 || d==100000)
    c++;
  if (e==100000 || f==100000)
    e++;
}
```

Figure 17: Code used to experiment with the effect of dependence reduction.

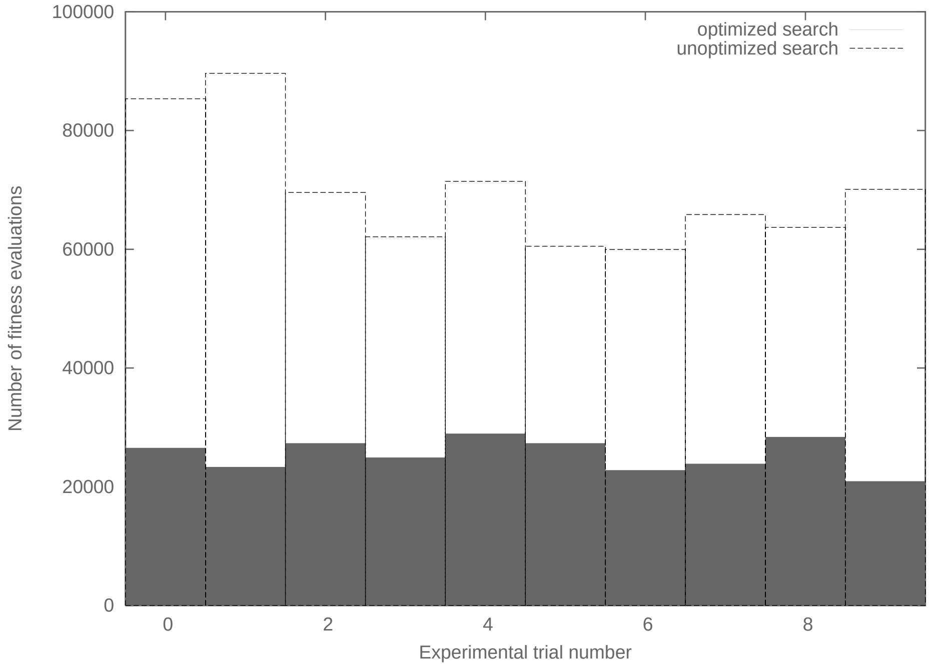


Figure 18: Predicate Analysis to Optimize Search–Based Test Data Generation.

are encouraging, particularly, since none of the current work on search-based testing [31, 47, 50, 56, 57, 60, 68] uses search–space reduction through source-code analysis.

6.2 Understanding Software Evolution

This section shows how the visualizations can be used to explore software evolution. It presents a case study in which two versions of the `flex` program were analyzed. These will be referred to as the 'earlier version', `flex-2.4.7`, and the 'later version', `flex-2.5.4`. In this section, the global variable bubble chart visualizations shown in Figure 8 for each of the two versions will be used to explore some of the changes between the two versions. In this way the section illustrates the way in that the bubble chart visualization might be used to support understanding of software evolution.

In general, looking at the bubbles in the global variable bubble charts of two versions of a program, a shift of bubbles to the right indicates that more globals are available to a typical predicate, while a shift upwards indicates that more globals (whether new or original) are being depended upon. An increase in bubble size indicates that predicates have been added.

Turning to the specific case of `flex`. A comparison the du-global bubble charts for the earlier and later versions shows both a rightward and an upward movement of bubbles. In general, tool support is required to identify comparable bubbles (those summarizing predicates of the same functions) with certainty. In practice, it can be done using only the visualizations with reasonable accuracy. For flex, comparable bubbles show a right shift of approximately 20-30 du-globals. For example, the earlier version's farthest right bubble is located at 220 on the horizontal axis (220 globals are available), while in the later version, the corresponding bubble is located at 253 (indicating 253 globals are available). Something has happened during the evolution of `flex` to increase the number of globals available to these predicates.

Next, consider the two groups of four bubbles that have $x > 100$ and $y > 20$. That is, more than 100 globals available, but only 20 used. In moving from the earlier to the later version of `flex`, all four of these bubbles shift upwards (an indicator of increased reliance on globals).

Any such up–shift should be considered a warning that evolution has introduced potentially unwanted and undesirable data-dependencies. The bubbles direct attention towards the relevant sections of the code and the slices taken from it. For example, in the earlier version of `flex`, the slice on `if (performance_report > 0)` from function `readin` includes 32 LoC and 1 global (`performance_report`). By contrast, in the later version, the slice on this same statement includes 567 LoC and just over 100 globals. This is clearly a dramatic change in the dependence profile of the program and it merits further investigation.

The source code was analyzed to explore the reason for the increase in global dependence. It turns out that it had three principal causes. First, evolution of `flex` has added support for character class expressions, second, calls to standard I/O functions were replaced by calls to custom I/O functions, and finally, there was an increase in command line option error checking. Centralization of I/O increases the inter-dependence between many of the globals variables. The consequence is that any predicates that depend upon some, tend to depend upon more of the rest. The increased error check ensures, *inter alia*, that certain combinations of options are not permitted. Since these options are encoded in global variables, the increased error checking causes a dramatic increase in the number of globals, both available to and genuinely used by, the predicates. It also compounds the increased inter–dependence between globals.

Finally, all four bubbles that show significant up–shift also show a predicate count increase (depicted as a bubble size increase in the chart). For example, in `yyparse`, to track the beginning-of-line (bol), the action '`YY_USER_ACTION`' from each `switch` case in the earlier version is replaced, in the later version, by

```
#define YY_RULE_SETUP \
    if ( yyleng > 0 ) \
```

```
        yy_current_buffer->yy_at_bol = \
                (yytext[yyleng - 1] == '\n'); \
    YY_USER_ACTION
```

This results in the insertion of a new predicate into each `switch` case.

Adding command line checking and character classes were clearly a move to improve the functionality of `flex`. The visualizations draw attention to these changes, but whether they are justified will depend upon the perceived importance of the improved functionality. However, the centralization of I/O is more of a perfective maintenance activity. It does not aim to improve functionality, but to re-organize the software, presumably with the goal of assisting further evolution in some way. The use of the bubble charts draws attention the impact of this change and raises questions about its justification.

6.3 Using the visualizations to assist comprehension

In addition to the patterns seen over multiple version of a program, bubble charts provide visual clues as to abnormalities within a program. In this section, two features that stand out in bubble charts are explored in more detail. These features, the existence of outliers and gaps in the occurrence of bubbles on the horizontal axis, can aid in the understanding potential trouble spots within the code.

Outliers are evident in the global dependence bubble charts of `ijpeg`, `barcode`, `gnugo`, and `flex2-5-4` (See Figure 8). The outliers are collections of predicates (all having access to the same number of globals). However, all the outlier predicates tend to come from a single function; thus, in the following discussion "outlier function `f`" is often used in place of "outlier predicates from function `f`."

To begin with, `ijpeg` has a clear outlier in its dependence on globals (the small bubble on the top right). This bubble is from predicates for which the value of max-du-globals is large.

function	# Nodes	# Predicates	# Formals	# du-Globals
go_..._compression	220	2	0	46
go_..._compression_and_decompression	258	1	0	63
main	301	0	2	75
parse_args	1,352	45	2	84

Figure 19: Functions in `ijpeg` with largest numbers of du-globals available.

```
parse_args
{
    ...
    for(; argv[i]; i++) {
        ...
        else if( !strcmp( argv[i], "-difference.histogram" ) ) {
            histogram_flag = atoi( argv[++i] );
        }
    ...
}
```

Figure 20: Fragment of code from `ijpeg` corresponding to outlier.

The top 4 functions from `ijpeg`, when sorted by max-globals, are presented in Figure 19. The outlier comes from predicates in the function `parse_args`, which has 45 predicates with 84 globals available to them. All of these predicates test for command-line options. Each is contained in a `for` loop that depends (transitively) on 67 globals. As each predicate is dependent on the `for` loop each predicate depends on 67 globals. A fragment of the code is presented in Figure 20. Interestingly, the other three functions from Figure 19 are from the same file as `parse_args`. These three functions all have very low dependence and contain very few predicates (0, 1, and 2); thus, they produce the three very small bubbles (dots) near the horizontal axis.

The second example with an outlier is `barcode`. `Barcode`'s `main` function has 108 du-globals available and 33 predicates. It looks like an outlier because its bubble is much further

function	# Nodes	# Predicates	# Formals	# du-Globals
main	3,809	47	2	226
do_move	1,069	8	4	204
play_test	940	12	2	197

Figure 21: Functions in `gnugo` with largest numbers of du-globals available.

to the right than the others. Further investigation reveals that the call tree for `barcode` is broad and shallow. The function `main` (transitively) calls all functions. Therefore, it potentially uses all global variables. Because the call-tree is broad and shallow, no other function potentially uses such a large number of globals.

However, the `main` function appears on the trend line for dependence. Its du-globals are those of its called functions. It therefore has more du-globals available to it, but depends upon a similar proportion of these to the average for all the other functions. This can be contrasted with the outlier for `ijpeg` that, not only had more globals available to it, but contains predicates with a higher-than-average dependent proportion.

The `gnugo` program has three bubbles that denote sets of predicates with about 200 available du-globals, but that use fewer than 10 of these. These are the three bubbles that appear on the right hand side of the bubble chart, near the horizontal axis. Each comes from a different file, but each is an outlier for a similar reason; all call most of the program, initialize a large number of globals, and contain predicates that are largely unaffected by the initialized variables. Data for the three functions, corresponding to the three outlier bubbles are presented in Figure 21.

For example, the function `main` contains 320 lines of code, 180 of that form a `switch` statement, used to process command-line arguments. A fragment of code that illustrates this is contained in Figure 22. This accounts for most of the direct global uses. The remaining lines of `main` are primarily concerned with determining the 'play mode' for the game.

```
/* Now weed through all of the command line options.  */
 while ((i=getopt_long(argc, argv, "-ab:B:d:D:Egh::H:l:L:o:p:r:fsStTvwm:",
      long_options, NULL)) != EOF)
  {
    switch (i) {
      case 'T' : printboard++; set_opt_display_board(1); break;
      case 't' :  ++verbose; break;
      case 'a' :  allpats = 1; break;
...
```

Figure 22: Fragment of code from `gnugo` for command line processing.

The rest of the globals that `main` may use or modify come from called functions. For example, `main` directly calls function `play_test` (also an outlier), which potentially uses 197 globals. This function is a test harness that calls most of the rest of the program. Likewise, the function `do_move` calls most of the rest of the program.

By contrast to `barcode`, `gnugo` has longer transitive call chains in its call graph. This creates several functions, (`main`, `do_move` and `play_test`), which potentially access a large number of globals. However, unlike `barcode` (and `ijpeg`), the globals being initialized do not affect the initialization process logic (*i.e.*, its predicates). This explains why the three bubbles are 'unusually close' to the horizontal axis.

The final outlier bubble chart is for `flex2-5-4`. Here there is one bubble on the 'far right' (near $x = 17$) that touches the horizontal axis. This is an outlier, likely to draw the viewer's attention, since it contains a large number of predicates potentially affected by a large number of globals, but that turn out to be affected by very few of these 'potential' globals. The bubble corresponds to the function `make_tables`, which has 174 globals available to it and 56 predicates. This function merely outputs information including the values of many global variables. However, these variables do not affect any of the 56 predicates (which are used to determine that data to output). Furthermore, the predicates are all linear (un-nested) and simple, for example

`if ( C_plus_plus ).`

In addition to outliers, a number of the bubble charts for global variables (see Figure 8) show gaps in the occurrence of bubbles along the horizontal axis. This occurs, for example, in the global variable bubble charts for `copia`, `findutils`, `bc`, and `a2ps`. Examination of the code reveals that these were all caused by the occurrence of a set of mutually recursive functions. For example, program `bc` has a gap between the values 18 and 33 on the horizontal axis. There are 6 bubbles to the right of the gap, which have varying numbers of du-globals available to them. The data for these six functions and the others in `bc` are presented in Figure 23.

The call graph for these six functions is depicted in Figure 24. In `bc`, the function `main` calls `try_file`. As a result, it has available to it, all the du-globals of `try_file` together with a few of its own. Mutually recursive functions have broadly the same number of du-globals available to them, since they are all in the transitive closure of each one's call graph. The reader may wonder why, therefore, the three mutually recursive functions `dc_func`, `dc_eval_an` and `dc_evalstr` do not have an *identical* set of du-globals. They do not because some of them contain killing assignments to one of the potential du-globals. These killing assignments occur before the value of the potential du-global is referenced. In such situations, the dependence algorithm used in this study does not include them in the set of du-globals (a safe approximation, since no predicate can depend on the initial value of such a global).

In all the cases studied, the presence of gaps on the horizontal axis was caused by the existence of mutual recursion among a set of functions. Of course, the fact that a program contains mutual recursion can also be determined, quite naturally, from the call graph of the program. However, the bubble charts allow for greater comprehension of the impact of such mutual recursion upon dependence. Consider the six mutually recursive functions in `bc` for

function	# Nodes	# Predicates	# Formals	# du-Globals
dc_binop	211	5	2	13
dc_triop	275	7	2	14
dc_out_num	85	2	4	15
out_num	557	20	3	15
dc_getnum	481	20	3	16
dc_print	95	2	4	17
dc_printal	70	1	1	18
··· gap ···				
dc_evalfil	498	11	1	33
dc_evalstr	332	12	1	35
dc_eval_an	149	1	1	36
try_file	180	4	1	37
main	472	6	2	38
dc_func	2315	69	3	46

Figure 23: Analysis of the gap in the bubble chart for `bc` global dependence.

an example. The size and vertical location of the bubbles indicates the number of predicates involved and the average dependence of these upon the set of du-globals. For example, `dc_func` has a large number of predicates (69) and they tend to depend heavily upon the set of du-globals. By comparison, `dc_eval_an` and `try_file` have very few predicates (1 and 4 respectively) and these depend upon very few of the du-globals available to them. This indicates, *inter alia*, that `dc_eval_an` and `try_file` will be easier to generate test data for than `dc_func`.

7 Threats to Validity

For an experiment not involving human subjects, only two potential threats to validity need to be considered. These are threats to external validity and internal validity. External validity, sometimes refereed to as selection validity, is the degree to which the findings can be

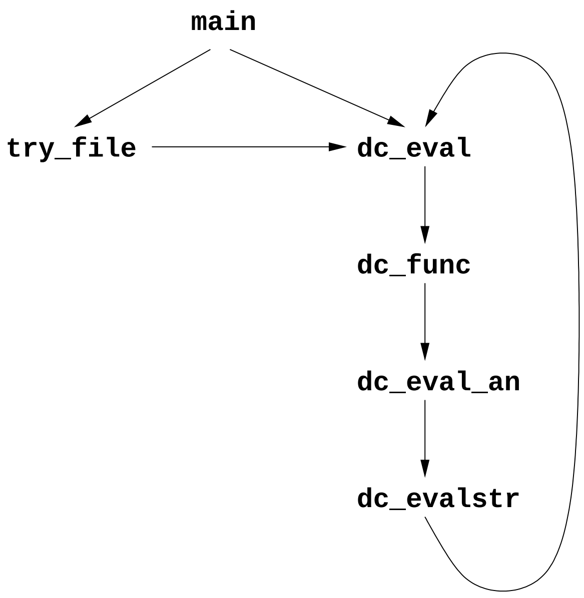


Figure 24: Call graph of mutually recursive functions from `bc`.

generalized. In this experiment, it is possible that the selected programs are not representative of programs in general and thus results from the experiment do not apply to 'typical' programs. The wide variety of programs used in the study helps to mitigate this concern. As most of these are open-source programs, it remains possible that non-open source programs will exhibit different behaviour.

Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variable on the dependent variable. In this experiment, potential threats come from inappropriate statistical tests or violations of statistical assumptions, inaccurate underlying analysis, and the degree to which the variables used in the study accurately measure the concepts they purport to measure (a form of construct validity). The statistical tests used were chosen based on past experiments and guidance of those trained in statistics. This serves to mitigate the possibility of an inappropriate test being employed. The analysis assumes that program slicing can determine the parameters used by a predicate. While, in theory, slicing does this, in practice, there are potential implementation issues. For example, the slicing tools used might contain an error, or employ imprecise analyses (*e.g.*, imprecise data-flow analysis or imprecise points-to analysis). To mitigate these concerns, the tools used were all mature and well tested. These concerns could be further addressed by replication of the experiment using alternative analyses (*e.g.*, Steensgaard's pointer analysis) and alternative slicing tools (*e.g.*, Sprite [49] or Unravel [44]).

8 Theoretical and actual time complexity

The previous three sections show the significant benefits possible using dependence analysis. However, these benefits can only be realized if it is possible to compute the dependence information in reasonable time. Fortunately, this is possible. This section describes the predicted and actual computational effort required to compute the dependence information presented in Sections 3 and 4. The algorithms takes as input and SDG. The construction and complexity of building an SDG are described elsewhere [29, 58, 18].

The algorithmic complexity of the analysis can be given in terms of the size of a procedure, S, and the number of predicates, P. The complexity of a brute force algorithm is $O(P \times S)$. This algorithm takes P slices, each having linear ($O(S)$) complexity. It is possible to reduce this to $O(S)$ assuming $O(1)$ bit vector operations, by propagating bit vectors over a topologically sorted DAG produced by first finding strongly connected components (SCCs) in the subgraph reachable from predicate nodes [64].

The actual data was collected on a PII 450 with 386Mb memory using an implementation of the brute force algorithm. The average time per predicate was 7.6ms. Statistical modeling of the computation time shows a tight fit with that predicted by its complexity. Using the sum of the number of edges and nodes in a PDG as the size measure, S, yields the following

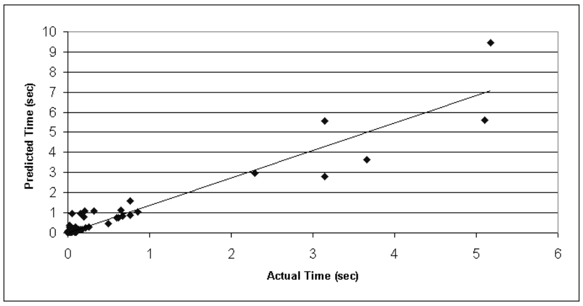


Figure 25: Actual computation time vs. predicted computation time per function.

model ($R = 0.99995$)

$$\textit{analysis time (s)} = 0.17 \times P \times S$$

Furthermore, the t-score for the coefficient of $P \times S$ is 7,881, which, being much greater that 1.94, is statistically significant. This indicates an excellent fit as seen the scatter plot of actual and predicted time shown in Figure 25. Note that most of the points in the plot occur right on the line with values less then 0.020.

Figure 26 shows computation times per program and per predicate for each of the 20 programs. The table separates out from the total running time, which is dominated by reading in the SDG, the time to perform just the dependence analysis. Programs `a2ps`, and to a lesser extent `sendmail`, are outliers in the per-predicate costs because they contain rather large functions. For example the largest function, from `a2ps`, contains 34,252 nodes, which is more than the total number of nodes for some of the mid-sized programs.

9 Related Work

The analysis reported here was first presented by the authors in a preliminary version of this paper [9]. The analysis is a variable dependence analysis based upon a modified form of a

			Time			
			All Predicates (seconds)		per Predicate (ms)	
Program	KLOC	Predicates	Total	Analysis	Total	Analysis
`replace`	0.6	84	0.10	0.01	1.19	0.12
`copia`	1.2	45	4.12	0.14	91.56	3.11
`compress`	1.9	88	0.22	0.01	2.50	0.11
`which`	5.4	136	0.43	0.02	3.16	0.15
`barcode`	5.9	365	1.34	0.06	3.67	0.16
`space`	11.5	593	2.06	0.10	3.97	0.17
`ed`	13.6	2627	11.34	5.60	4.32	2.13
`prepro`	14.8	619	2.67	0.10	4.31	0.16
`bc`	16.8	432	1.64	0.08	3.80	0.19
`findutils`	18.6	1819	10.58	6.43	5.82	3.53
`diffutils`	19.8	2212	11.59	6.83	5.24	3.09
`flex2-4-7`	21.2	911	4.38	0.96	4.81	1.05
`flex2-5-4`	21.5	1306	16.86	8.47	12.91	6.47
`espresso`	22.1	2497	11.68	0.84	4.62	0.34
`ijpeg`	28.2	2038	7.45	0.44	3.62	0.22
`ntpd`	45.6	3061	33.81	4.75	11.05	1.55
`a2ps`	62.9	5984	761.79	707.82	127.30	118.29
`gnugo`	81.7	5244	32.29	2.75	6.16	0.52
`sendmail`	85.7	4991	1248.98	122.74	250.21	24.60
`spice`	167.0	15527	1401.72	59.30	90.27	3.82
average	28.3	2529	121.90	43.25	20.52	7.64

Figure 26: Computation times.

standard program slicing algorithm [29]. Slicing was introduced by Weiser [73] in 1979 as a code extraction technique to determine, statically, the set of statements that potentially affect a set of variables at some chosen program point. It has since been extended to allow for dynamic analysis [35] and conditioned analysis [13]. There are several surveys on slicing in the literature, which cover slicing applications and techniques [8, 66] and empirical results collected about slicing [11].

There has been considerable previous work concerning program slice size and trade–offs between algorithmic precision and slice size [3, 10, 24, 37, 43, 49, 55, 62, 69, 79]. To some extent, these studies are also concerned with program dependence, albeit, indirectly, due to the way in which slices are constructed from a model of program dependence [29, 75]. However, the work reported here is the first large–scale study of predicate dependence.

The initial work on program slicing envisaged that slices would be an end product, presented to the user, to assist with comprehension [74] and debugging [45]. Much of the subsequent work on slicing has concerned the extension of slicing algorithms to cover language features such procedures [28], jumps [1, 5, 14, 20, 38] and pointers [43, 49, 54, 55], and to improve the efficiency of slicing algorithms [3, 30].

The work reported in the present paper follows an emerging trend in slicing research to use slicing analysis as a *means to an end rather than an end in itself.* For instance, previous work has shown that slicing can be used to construct metrics for assessing cohesion [7] and coupling [23]. Slicing has also been used as a support for software testing [19, 22, 25], re-engineering [33, 34, 40], virus detection [41] and security [78]. In all these applications, though the slices do influence the overall results, they are 'invisible' to the slice user.

There has also been previous work on using slicing and dependence analysis in visualization tools. For instance Ball and Eick [4] developed a tool called SeeSlice, which allows the user to see the entire text of the program at once, and to view the relative directness of dependence (in terms of distance from the slice node in the SDG). Balmas [6], developed a tool for dependence visualization, which allows the user to view the whole system at a high level of abstraction and to zoom in on particular aspects of dependence. Rilling [59], uses a hyperbolic tree (a kind of 'fish eye view') to allow dependence information to be displayed in such as way that the size of the view corresponds to the focus of attention; less important details appear at the edge of the tree, where they are barely visible, while important dependencies appear much bigger.

All these visualization techniques are concerned with managing the sheer quantity of information present in a large system when one wishes to display all dependencies. The work reported here differs from this previous work on dependence visualization, because it does not aim to show the dependency relationships themselves. Rather it shows the *impact* of the dependencies upon the predicates of the program. The goal of the bubble chart is to allow observations to be made at a high level about the effect of dependence upon the predicates (and thereby the control logic) of the program.

10 Summary and Future Work

This paper presented results from a study of 50,579 predicates in 20 C programs ranging from 600 lines of code to 167,000 lines of code. In general, it found that a typical predicate depends upon 72% of the formal parameters in the scope, 48% of the global variables potentially used or defined by a call to the predicate's procedure, and 2.4% of the global variables in scope.

The study discovered a strong inverse correlation between an increase in the number of formal parameters visible to a predicate and the proportion actually depended upon. There is no such correlation for either kind of global variable. The results of analysis for correlations provide strong evidence that the global and formal dependence sources are independent of one another and that the number of formals is independent of the size of the procedure that contains them.

It was shown how predicate dependence analysis can be useful as a supporting analysis for search based software testing. The analysis allows the search space to be reduced with a consequent decrease in the effort required to generate test data. More work is needed to provide a full evaluation of this potential application, but the initial results are encouraging.

The paper also introduced two visualizations for human analysis and interpretation of dependent proportions: bubble charts and skyline diagrams. It was shown that the bubble chart visualization can help with comprehension, by directing attention to code features that cause interesting properties in the visualizations, such as 'gaps' in dependence and outlier bubbles in the bubble charts. It was also shown, for example, that bubble charts are useful in supporting the analysis of change between different releases of a system.

Future work is required to explore these possibilities fully and to consider other possible applications of the analysis techniques used. For instance, consider the situation where a function has an unusually large number of formals, with an unusually low dependence of predicates upon the values of these formals. Such a function would show up as an outlier on the formal parameter dependence bubble chart. It might be that this suggests a possibility for re-factoring; the function needs to be split, with subsets of the original set of formals in each sub–function. The dependence analysis might be a guide, both to the identification of candidates for re-factoring and to the determination of 'cut points'. Finally, future work will consider the correlations between the results of dependence analysis and other indicators of qualities of the software such as maintainability, reliability, testability.

11 Acknowledgements

All three referees provided detailed comments and suggestions that helped to improve the presentation and content of the paper. DaimlerChrysler AG Berlin kindly made available their evolutionary test data generation tool, and André Baresel of DaimlerChrysler performed the experiments, for which results were reported in Section 6.1. The authors also wish to thank GrammaTech Inc. (`http://www.grammatech.com`) from providing CodeSurfer, Gregg Rothermel for providing the `space` program, Chris Morel for his statistical guidance, and Erin Ptah who assisted with the visualizations. Mark Harman is supported, in part, by EPSRC Grants GR/R98938, GR/M58719, GR/M78083 and GR/R43150 and by two development grants from DaimlerChrysler, Berlin. Dave Binkley is supported by National Science Foundation grant CCR-0305330.

References

[1] Hiralal Agrawal. On slicing programs with jump statements. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 302–312, Orlando, Florida, June 20–24 1994. Proceedings in SIGPLAN Notices, 29(6), June 1994.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[3] Darren C. Atkinson and William G. Griswold. Effective whole–program analysis in the presence of pointers. In *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering (FSE-98)*, volume 23, 6 of *Software Engineering Notes*, pages 46–55, New York, November 3–5 1998. ACM Press.

[4] Thomas Ball and S. G. Eick. Visualizing program slices. In Allen L. Ambler and Takayuki Dan Kimura, editors, *Proceedings of the Symposium on Visual Languages*, pages 288–295, Los Alamitos, CA, USA, October 1994. IEEE Computer Society Press.

[5] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control–flow. In Peter Fritzson, editor, 1^{st} *Conference on Automated Algorithmic Debugging*, pages 206–222, Linköping, Sweden, 1993. Springer. Also available as University of Wisconsin–Madison, technical report (in extended form), TR-1128, December, 1992.

[6] Françoise Balmas. Using dependence graphs as a support to document programs. In 2^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 145–

154, Montreal, Canada, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[7] James M. Bieman and Linda M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, August 1994.

[8] David Wendell Binkley and Keith Brian Gallagher. Program slicing. In Marvin Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[9] David Wendell Binkley and Mark Harman. An empirical study of predicate dependence levels and trends. In 25^{th} *IEEE International Conference and Software Engineering (ICSE 2003)*, pages 330–339, Portland, Oregon, USA, May 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

[10] David Wendell Binkley and Mark Harman. Results from a large–scale study of performance optimization techniques for source code analyses based on graph reachability algorithms. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 203–212, Amsterdam, Netherlands, September 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

[11] David Wendell Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[12] David Wendell Binkley, Mark Harman, L. Ross Raszewski, and Christopher Smith. An empirical study of amorphous slicing as a program comprehension support tool. In 8^{th} *IEEE International Workshop on Program Comprehension (IWPC 2000)*, pages 161–170, Limerick, Ireland, June 2000. IEEE Computer Society Press, Los Alamitos, California, USA.

[13] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

[14] Jong–Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[15] Andrea De Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, pages 9–18, Berlin, Germany, March 1996. IEEE Computer Society Press, Los Alamitos, California, USA.

[16] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, January 1996.

[17] Keith Brian Gallagher. Evaluating the surgeon's assistant: Results of a pilot study. In *Proceedings of the International Conference on Software Maintenance*, pages 236–244. IEEE Computer Society Press, Los Alamitos, California, USA, November 1992.

[18] Grammatech Inc. The codesurfer slicing system, 2002.

[19] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5(3):143–162, September 1995.

[20] Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance and Evolution*, 10(6):415–441, 1998.

[21] Mark Harman and Robert Mark Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[22] Mark Harman, Lin Hu, Robert Mark Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, January 2004.

[23] Mark Harman, Margaret Okunlawon, Bala Sivagurunathan, and Sebastian Danicic. Slice-based measurement of coupling. In Rachel Harrison, editor, 19^{th} *ICSE, Workshop on Process Modelling and Empirical Studies of Software Evolution*, Boston, Massachusetts, USA, May 1997.

[24] Mary Jean Harrold and Ning Ci. Reuse-driven interprocedural slicing. In *Proceedings of the 20th International Conference on Software Engineering*, pages 74–83. IEEE Computer Society Press, April 1998.

[25] Robert Mark Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[26] Micheal Hind. Pointer analysis — haven't we solved this problem yet? In *Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, Utah, USA, June 2001. ACM.

[27] John H. Holland. *Adaption in Natural and Artificial Systems.* MIT Press, Ann Arbor, 1975.

[28] Susan Horwitz, Thomas Reps, and David Wendell Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language*

Design and Implementation, pages 25–46, Atlanta, Georgia, June 1988. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[29] Susan Horwitz, Thomas Reps, and David Wendell Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[30] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand Interprocedural Dataflow Analysis. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 95)*, pages 104–115, October 1995.

[31] B.F. Jones, H.-H. Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms. *The Software Engineering Journal*, 11:299–306, 1996.

[32] Bryan F. Jones, David E. Eyres, and Harmen H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal*, 41(2):98–107, 1998.

[33] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 155–169, N.Y., January 19–21 2000. ACM Press.

[34] Raghavan Komondoor and Susan Horwitz. Effective automatic procedure extraction. In *11th IEEE International Workshop on Program Comprehension*, page To appear., Portland, Oregon, USA, May 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

[35] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

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

[37] Jens Krinke. Evaluating context-sensitive slicing and chopping. In *IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 22–31, Montreal, Canada, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[38] Sumit Kumar and Susan Horwitz. Better slicing of programs with jumps and switches. In *Proceedings of the* 5^{th} *International Conference on Fundamental Approaches to Software Engineering (FASE 2002)*, volume 2306 of *Lecture Notes in Computer Science*, pages 96–112. Springer, 2002.

[39] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7:49–76, 2002.

[40] Arun Lakhotia and Jean-Christophe Deprez. Restructuring programs by tucking statements into functions. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 677–689. Elsevier, 1998.

[41] Arun Lakhotia and P. Singh. Challenges in getting formal with viruses. *virus bulletin*, September 2003.

[42] Loren D. Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, pages 495–505, Berlin, 1996.

[43] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In Oscar Nierstrasz and Michel Lemoine, editors, *ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 199–215. Springer-Verlag / ACM Press, 1999.

[44] James R. Lyle, Dolores R. Wallace, James R. Graham, Keith B. Gallagher, Joseph P. Poole, and David Wendell Binkley. Unravel: A CASE tool to assist evaluation of high integrity software, Volume 1: Requirements and design. Technical Report NISTIR 5691, US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899., 1995.

[45] James R. Lyle and Mark Weiser. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications*, pages 877–882, Peking, 1987. IEEE Computer Society Press, Los Alamitos, California, USA.

[46] Spiros Mancoridis, Brian S. Mitchell, Yih-Farn Chen, and Emden R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 50–59. IEEE Computer Society Press, 1999.

[47] C.C. Michael, G. McGraw, and M.A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, (12):1085–1110, December 2001.

[48] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[49] Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. Improving program slicing with dynamic points-to data. In William G. Griswold, editor, *Proceed-*

ings of the 10^{th} *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-02)*, pages 71–80, New York, November 2002. ACM Press.

[50] F. Mueller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*, pages 144–154, Washington - Brussels - Tokyo, June 1998. IEEE.

[51] Jim Ning, Andre Engberts, and Voytek Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50–57, 1994.

[52] A. Jefferson Offutt. An integrated system for automatically generating test data. In Raymond T. Ng, Peter A.; Ramamoorthy, C.V.; Seifert, Laurence C.; Yeh, editor, *Proceedings of the First International Conference on Systems Integration*, pages 694–701, Morristown, NJ, April 1990. IEEE Computer Society Press.

[53] A. Jefferson Offutt, Z. Jin, and Jie Pan. The dynamic domain reduction approach to test data generation. *Software Practice and Experience*, 29(2):167–193, January 1999.

[54] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Effects of pointers on data dependences. In 9^{th} *IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 39–49, Toronto, Canada, May 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[55] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Incremental slicing based on data-dependences types. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*, pages 158–167, Florence, Italy, November 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[56] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability*, 9:263–282, 1999.

[57] Hartmut Pohlheim and Joachim Wegener. Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1795, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[58] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *ACM Foundations of Software Engineering (FSE'94)*, pages 11–20, New Orleans, LA, December 1994. ACM SIGSOFT Software Engineering Notes 19, 5 (December 1994).

[59] Jurgen Rilling and S. P. Mudur. 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*, pages 42–52, Richmond, Virginia, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[60] A. Schultz, J. Grefenstette, and K. Jong. Test and evaluation by genetic algorithms. *IEEE Expert*, 8(5):9–14, 1993.

[61] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. *Lecture Notes in Computer Science*, 1302:16–34, 1997.

[62] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control-flow. In *Proceedings of the 21st International Conference on Software Engineering*, pages 432–441. ACM Press, May 1999.

[63] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 32–41, St. Petersburg, Florida, January 1996. ACM Press.

[64] Robert Endre Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9(3):355–365, December 1974.

[65] Frank Tip. A survey of program slicing techniques. Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, Amsterdam, 1994.

[66] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[67] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis*, pages 73–81. ACM/SIGSOFT, March 1998.

[68] Nigel Tracey, John Clark, and Keith Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)*, pages 169–180. IFIP, January 1998.

[69] G. A. Venkatesh. Experimental results from dynamic slicing of C programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–216, March 1995.

[70] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms*, 43(14):841–854, 2001.

[71] Joachim Wegener, K. Grimm, M. Grochtmann, Harmen Sthamer, and Bryan F. Jones. Systematic testing of real-time systems. In *4th International Conference on Software Testing Analysis and Review (EuroSTAR 96)*, 1996.

[72] Joachim Wegener, Harmen Sthamer, Bryan F. Jones, and David E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality*, 6:127–135, 1997.

[73] Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[74] Mark Weiser. Programmers use slicing when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[75] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[76] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. *In Proceedings of the SIGPLAN 99 Conference on Programming Language Design and Implementation (Atlanta, GA)*, pages 91–103, May 1999.

[77] Jyh-shiarn Yur, Barbara G. Ryder, and William A. Landi. An incremental flow-and context-sensitive pointer aliasing analysis. In *Proceedings of the 21st International Conference on Software Engineering*, pages 442–452. IEEE Computer Society Press, Los Alamitos, California, USA, May 1999.

[78] X. Zhang and R. Gupta. Hiding program slices for software security. In *First Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 325–336, San Francisco, CA, USA, March 2003.

[79] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In 25^{th} *IEEE International Conference and Software Engineering (ICSE 2003)*, pages 319–329, Portland, Oregon, USA, May 2003. IEEE Computer Society Press, Los Alamitos, California, USA.