Locating Dependence Clusters and Dependence Pollution

David Binkley
Loyola College
Baltimore MD
21210-2699, USA
binkley@cs.loyola.edu

Mark Harman
King's College London
Strand, London
WC2R 2LS, UK.
Mark@dcs.kcl.ac.uk

Keywords: ripple effect, dependence analysis, system dependence graph

Abstract

A dependence cluster is a set of program statements all of which are mutually inter–dependent. Such clusters can cause problems for maintenance, because a change to any statement in the cluster will have a potential impact on all statements in the cluster.

This paper introduces the concept of dependence clusters and dependence pollution and shows how a simple visualisation can be used to quickly and effectively locate them. The paper presents the results of two empirical studies and several case studies which evaluate the approach.

The results indicate the importance of dependence cluster analysis: for a set of 20 programs, ranging in size from 1,170 LoC to 179,623 LoC, 99.6% of clusters identified were within 1% tolerance of being identical, while dependence clusters were found to be surprisingly common: 80% of the programs studied contained clusters of 10% or more of the program.

1 Introduction

The impact of change is one of the most pressing problems facing software maintainers at the source code level. It is well known that a simple source code change can have far-reaching and unexpected consequences. The problem of tracking the impact of such source-code-level changes is one of the important cost drivers behind remedial maintenance activities such as Y2K remediation, Euro currency conversion, zip code, telephone and bank account numbering changes [22, 23].

From the maintainer's point of view, the less dependence there is in a system, the lower the chances of some unexpected knock-on effect (or 'ripple effect' [9]). Therefore, a set of statements in a program that are all mutually inter–dependent should be viewed with a certain amount of caution and concern; a change to one is a change to all.

In this paper, such sets of mutually inter–dependent statements are called 'dependence clusters'. Where a program contains a large dependence cluster, software modification may cause significant ripple effects and, as a result, problems for maintainers. Furthermore, it may turn out that the dependence that binds together the statements in the dependence cluster is avoidable. In this paper, this phenomenon is called dependence 'pollution' because it is dependence which has 'leaked out' from one part of the program to influence another part, with potentially harmful effects on maintainability.

Dependence clusters are formally defined as the solution to a reachability problem over a program's System Dependence Graph (SDG) [21]. However, as with other forms of pollution (like noise pollution [30]), what constitutes dependence pollution is an inherently subjective matter, determined by whether dependence is avoidable. The paper gives case study examples of what might (and might not) be deemed to be dependence pollution, illustrating the way in which dependence cluster analysis can be used to support and inform maintenance activities.

The paper introduces a simple approach for finding dependence clusters in terms of slice sizes and a visualisation (the Monotonic Slice-size Graph, MSG). The approach approximates whether statements (or, more precisely, nodes of the dependence graph) are in a dependence cluster, by checking to see if the size of their slice is 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.

Two empirical studies are presented. 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 import to gauge how often these false positives turn up in practice. If they are too common 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, theoretically at least, it seems likely that as slice size increases, two slices which happen to have the same size are likely to share a great deal of similar content. It is just too much of a co-incidence to find two large slices of the same size but entirely different content. The verification study bears out this claim, showing that for 99.6% of clusters, the slices in these clusters are within 1% of being completely 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 action). Of course, what constitutes a *large* dependence cluster depends upon the definition of ‘large’. The validation study adopts a cautious approach: a dependence cluster is deemed to be large if it contains 10% or more of the program’s slices. Even with this high threshold, the validation study reveals that large dependence clusters are surprisingly common: In real code, 14 out of 20 programs studied contained at least one dependence cluster of at least 10% of the total number of slices of the program.

The programs studied were 20 programs, all written in C, mostly open source, with some industrial programs from the European Space Agency. A summary of information about the programs studied is contained in Figure 1. In total, the set of programs represent just over 450,000 lines of code.

Overall, the findings of the paper suggest that dependence clusters and dependence pollution are worthy of further study. The paper shows that dependence clusters are easy to define, to locate and to investigate and provides evidence to suggest that the MSG visualisation is helpful in analysing them. It also shows that dependence clusters occur with surprising frequency and size in real programs. Despite inherent subjectivity, the paper demonstrates that it is possible to identify some of these clusters as pollutants, which can be removed by refactoring. This indicates indicating that the study of dependence pollution can act as a supporting mechanism for software maintenance.

The primary contributions of the paper are as follows:

Program	LoC	Brief Description
`acct`	10,182	Process monitoring utilities
`barcode`	5,926	Barcode generator
`bc`	16,763	Calculator
`byacc`	6,626	Berkeley `yacc`
`cadp`	12,930	Protocol toolbox
`copia`	1,170	ESA signal processing code
`diffutils`	19,811	File comparison utilities
`ed`	13,579	Test editor
`epwic`	9,597	Image compressor
`espresso`	22,050	Logic simplifier
`findutils`	18,558	File finding utilities
`go`	29,246	Strategy game
`indent`	6,724	Text Formatter
`ntpd`	47,936	Network Daemon
`oracolo2`	14,864	Antennae array set-up
`prepro`	14,814	ESA array pre-processing
`space`	9,564	ESA space program
`spice`	179,623	Digital circuit simulator
`userv`	8,009	Access control utility
`which`	5,407	Unix utility
Total	453,379	

Figure 1. The subject programs studied.

1. The concept of a dependence cluster is introduced together with a simple visualisation (the Monotone Slice–size Graph) to help with identification of large dependence clusters.
2. The paper contains two empirical studies, which give evidence to support the central claims that:
    - The MSG represents a precise approximation to allow identification of dependence clusters.
    - Large dependence clusters are prevalent.
3. Case study evidence is presented to demonstrate that dependence clusters can be used to identify and help remove possible sources of dependence pollution.

The remainder of the paper is organized as follows: Sections 2 and 3 introduce dependence clusters and Monotone Slice-size Graphs. Sections 4 and 5 present the results of the two empirical studies concerned with verification and validation. Section 6 presents case studies which illustrate the use of dependence pollution as part of the maintenance process. Section 7 considers the threats to validity of the results. Section 8 briefly describes related work on dependence and clustering and Section 9 concludes.

2 Dependence Clusters

A dependence cluster is a set of program points which mutually depend upon one another. In this paper ‘program points’ will be taken to mean nodes of the Control Flow Graph (CFG). Any change to the computation represented at one point in a dependence cluster potentially affects the computations represented all other points. Definition 1 below, defines the concept of a dependence cluster more formally.

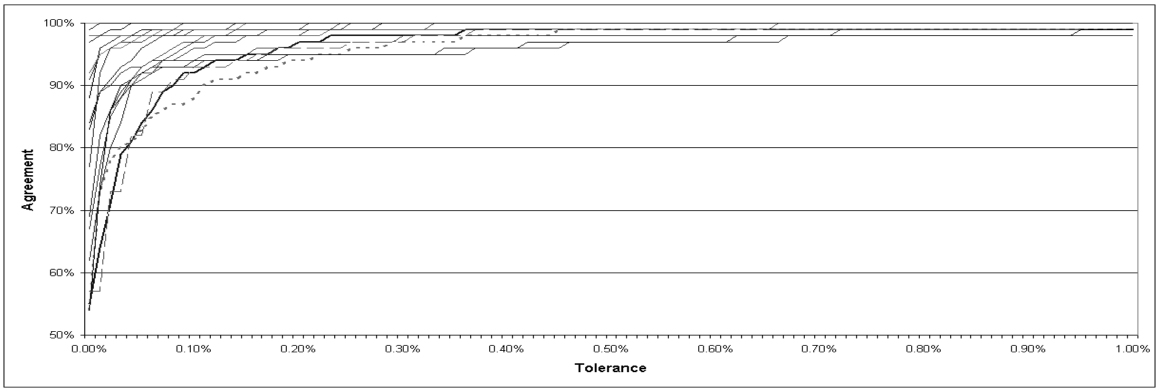


Figure 2. Agreement Levels for up to 1% of Tolerance

Definition 1 *Dependence Cluster*
A *dependence cluster* is a set of nodes, $\{N_1, \ldots, N_m\}$ ($m > 1$), of the Control Flow Graph (CFG) such that for all i, $1 \leq i \leq m$ and for all j, $1 \leq j \leq m$ N_i depends on N_j.

Dependence clusters can be identified using program slicing. A static backward program slice is a semantically meaningful portion of a program that captures a subset of the program's computation [34]. A slice is computed from a 'slicing criterion' (a program point and variable of interest) and contains those parts of the program which potentially affect the slicing criterion. The results presented in this paper use the System Dependence Graph (SDG) [21] to compute program slices

Two nodes which depend upon each other must have the same slice. Furthermore, since a slice contains the node for which it is constructed, where two nodes have identical slices, then each node must be in the slice of the other and therefore, each node must depend upon the other. Consequently, it is possible to use the fact that two nodes have the same slice as a way of determining whether they depend upon each other. That is, according to Definition 1, two nodes are in a dependence cluster iff they have the same slice.

3 Monotone Slice-size Graphs

It would be possible to locate dependence clusters by finding slices and checking to see which slices are identical. However, in this paper, an approximation is used for 'same slice' which is not only more efficient, but also leads to a useful visualisation for identifying clusters: the Monotone Slice-size Graph (MSG). Rather than checking to see if two nodes of the SDG yield identical slices, the approach is to simply check whether the two nodes yield slices of the same size.

The conjecture which underpins this approximation is that two slices which are the same 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 in this way. 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 approximation quality.

Definition 2 *Monotone Slice-size Graph (MSG)*
A *Monotone Slice-size Graph (MSG)* is a graph of the function of slice size, plotted for monotonically increasing size. That is, slices are sorted according to increasing slice size and the sizes are plotted on the vertical axis against slice number in order on the horizontal axis.

The MSG visualisation plots a landscape of monotonically increasing slice sizes, in which dependence clusters correspond to sheer–drop cliff faces. 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?). For example, looking ahead to Figure 6 which contains the MSGs of 10 programs. Consider the MSG of the program `userv` at the bottom of the figure. The graph shows (reading along the horizontal axis) that the first 38% or so of slices are extremely small, but then reveals a sharp increase at about 44%, where the remaining slices are all comfortably over 50% of the program in size.

The slice size approach is inherently more efficient than comparing slice content: Computing and comparing all slices' sizes for a given program is $O(n^2)$, while computing and comparing all slices' content is $O(n^3)$, where n is the number of edges in the SDG.

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 which contains it. As an example, consider the MSG of the program `userv` in Figure 6 once more. The sharp increase in slice size, which 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.

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 close approximation to 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 nearly the same vertices. Of course, the answer will depend upon the interpretation given to 'nearly the same'. This will be referred to as the 'tolerance'; the degree to which two slices can differ in content while being deemed to be essentially the same.

Figure 2 plots the tolerance (on the horizontal axis) against the agreement between slice size and slice contents (on the vertical axis) for the 20 programs studied. Both axes are represented as percentages. A tolerance of $x\%$ means that, of the total number of nodes in the two slices, the percentage of nodes upon which they differ is $x\%$, thus it is possible to speak of slices being 'identical within a certain tolerance'. For a given value of tolerance, $x\%$, an agreement of $y\%$ means that $y\%$ of the total number of slices are identical (within $x\%$ tolerance).

As can be seen from the figure, almost total agreement is reached for most programs with a very small tolerance. The horizontal axis is cut at 1% tolerance, so all the data shown in Figure 2 concern slices which are within 1% of containing identical sets of nodes. In total, 99.6% of the clusters are represented on this graph. That is, 99.6% of clusters are within 1% tolerance of total agreement. If the figure were to be redrawn, with a horizontal axis extended out to 100%, then the detail would be completely lost, because almost all programs would appear to almost immediately reach 100% agreement on the vertical axis.

Of course, there are a few programs where there are some slices that simply *happen* to be the same size, but which contain completely different sets of nodes. This should be expected to occur in a suitably large data set. Since the data presented in this paper comes from the analysis of almost half a million lines of code, it is likely that this may occur.

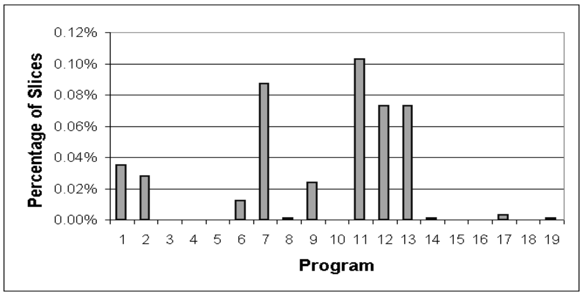


Figure 3. Sparsity of high tolerance

To get a sense for how common this occurrence is, consider the data presented in Figure 3. This figure shows all the data for which a tolerance of more than 1% is required to reach 100% agreement. The horizontal axis shows each program studied. The vertical axis shows the percentage of same-size slices which require more than 1% tolerance in order to agree 100%. As the figure shows, for almost half of these programs, there are simply *no* slices of the same size which require more than 1% tolerance in order to agree 100%. However, there are a few which do; these constitute false positives. For the 20 programs studied, only 0.4% of the clusters required a tolerance of more than 1% in order to achieve 100% agreement.

Furthermore, even this low figure of 0.4% is perhaps unduly pessimistic because it records the number of *clusters* which require more than 1% tolerance in order to fully agree. However, even in a cluster which requires more than 1% tolerance for full agreement, many of the individual slices in the cluster may fully agree, with only a few disagreeing. The figure for the number of pairwise slice comparisons which fail to agree within 1% tolerance is only 0.00533%. These results provide strong evidence for the claim that 'size agreement' is a good approximation for 'slice content agreement' and thus for locating dependence clusters using MSGs.

5 Empirical Validation: Do Dependence Clusters Occur in Practice?

In the empirical study reported in this section, the concern is to validate the approach. In determining whether or not a program has a large dependence cluster or not, there is a value judgment to be made concerning the size of a cluster. Clearly, most programs will have small clusters of slices. For the empirical study, the threshold above which a cluster is considered to be 'large' was set to 10%. In other words, a cluster is large if 10% or more of its slices are in the cluster. This relatively large threshold was chosen in order to provide a conservative answer to the validation question. That is, in a medium size program of (perhaps)

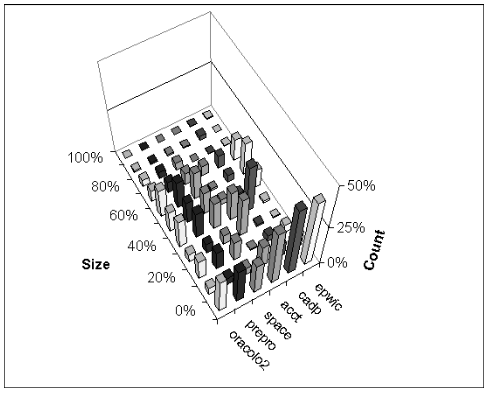
50,000 slices, a cluster of 5,000 slices of identical size is important. It would not be likely to arise by chance. However, smaller clusters (of a few thousand slices) may also be interesting and worthy of further investigation. Therefore, the results presented in this section can be thought of as a lower bound estimate of the frequency of large dependence clusters in the programs studied.

In total, only 6 of the 20 programs were found to contain no large clusters according to the 10% threshold. Of the remaining programs, 4 were found to have spectacularly large clusters which encompass most of the program. These 4 are explored in more detail in Section 6. The 20 programs were thus divided onto three categories: those which contained no large clusters, those which contained large clusters and those with spectacularly large clusters.

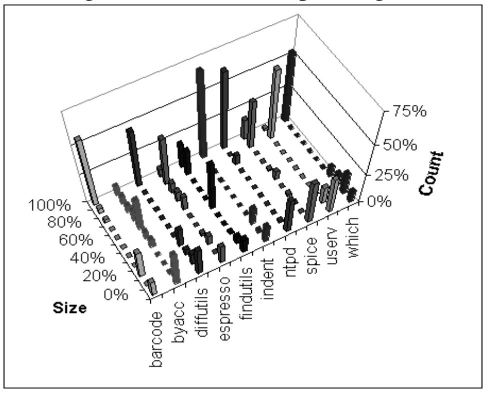
Figure 4 shows the data for each of these three categories as a distribution of slice sizes for each program. Figure 4 contains one three dimensional plot for each of the three categories. One of the two horizontal axes is used to set out the data for each (named) program. The other refers to the size of the slice (expressed as a percentage of the program which contains it). To fit the data into a single figure, the sizes of slices are grouped into ranges for each 10%. The vertical axis shows the proportion of slices having that size. As can be seen, for Figure 4(a) slice sizes fall fairly evenly over the 10% ranges when compared to Figure 4(c), which shows how dramatically the presence of exceptionally large clusters biases the distribution of slice sizes.

Figure 5 shows the MSGs of the category of 6 programs for which there were no large clusters. These programs show only small 'cliff drops' in the landscape of their MSG. However, 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 maintenance. Figure 6 shows the MSGs for the category of 10 programs which were found to contain evidence for the presence of large clusters (more than 10% of the slices in a single cluster). Figure 7 shows the MSGs of the category of 4 programs where these clusters were particularly pronounced.

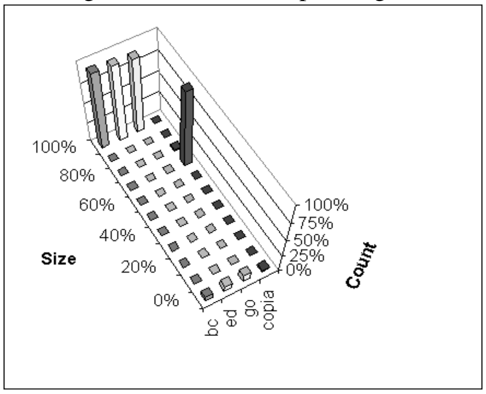
Visually, the MSGs clearly help identify large dependence clusters: compare the MSGs in Figures 6 and 7 (which clearly show large, tell-tale, cliff faces) with those in Figure 5 (which have comparatively smooth landscapes).



(a) Absence of Clusters
See Figure 5 for the corresponding MSGs.



(b) Clusters Present
See Figure 6 for the corresponding MSGs.



(c) Enormous Clusters Present
See Figure 7 for the corresponding MSGs.

Figure 4. Size/Frequency for each of the three categories

6 Case Study Results

This section defines dependence pollution and uses this concept to investigate, in more detail, the four programs where extremely large dependence clusters were found (see Figure 7). These are indicative examples only; any of the programs for which the MSGs are depicted in Figures 6 and 7 could have been chosen, as they all show signs of large dependence clusters. Unfortunately, space does not allow a detailed treatment of all four, so two (the industrial program, `copia` and the open source program `bc`) are described in more detail as case studies, while only brief (but indicative) observations are made about the remaining two

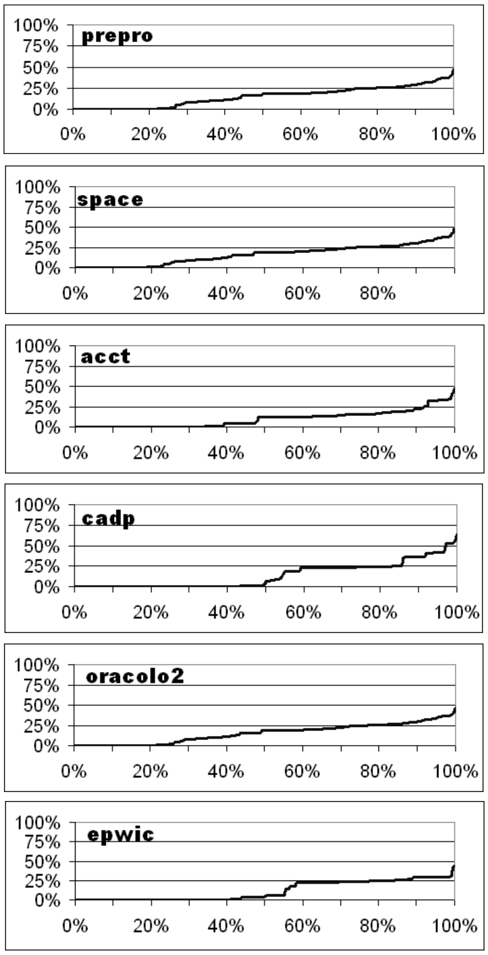


Figure 5. MSGs for Programs with an Absence of Large Dependence Clusters (No Cliff Faces in the MSGs)

(the editor, `ed` and the board game `go`). The purpose of this section is to give a flavour for the kind of maintenance investigation which could take place supported by dependence cluster analysis.

6.1 Dependence Pollution

Large dependence clusters are inhibitors to successful software maintenance because of the way in which a change to one element of the cluster can ripple to all other members. Therefore, this paper adopts the term 'pollution' for unwanted and avoidable dependence clusters. As with other forms of pollution, such as 'noise pollution' [30], what constitutes pollution is inherently subjective. In the case of dependence pollution, the judgment is determined by whether dependence is large and whether it is deemed to arise as the result of avoidable coding styles. In practice, it may well be a matter of degree: how problematic a dependence clus-

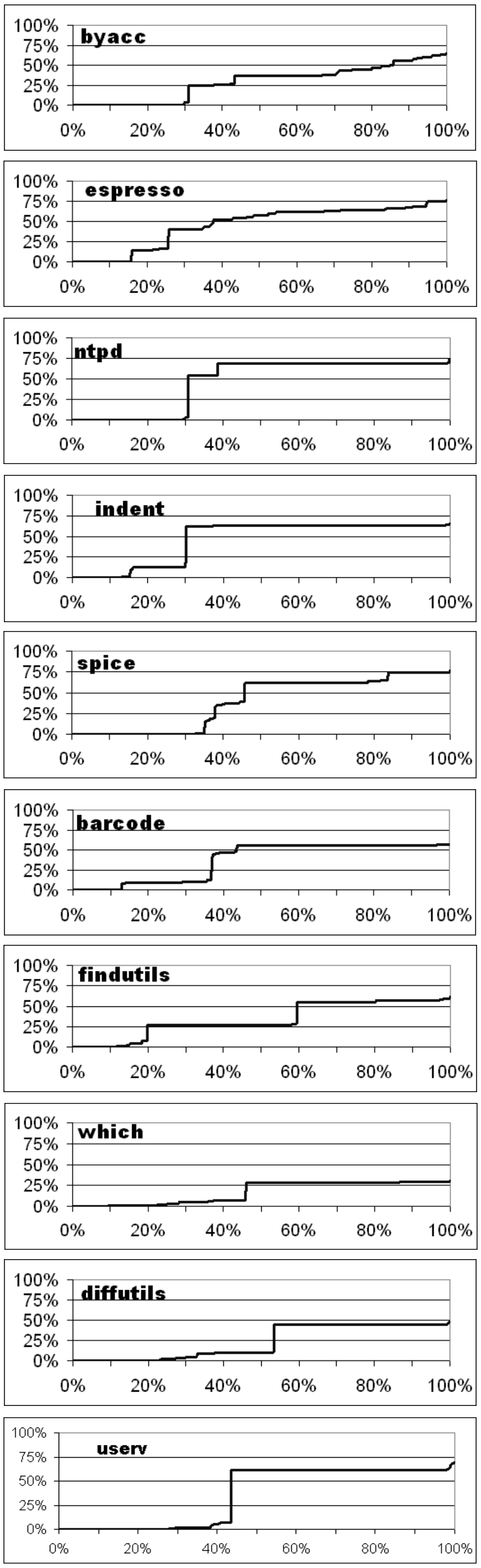


Figure 6. MSGs of Programs with Large Dependence Clusters (Denoted by Cliff Faces in the MSGs)

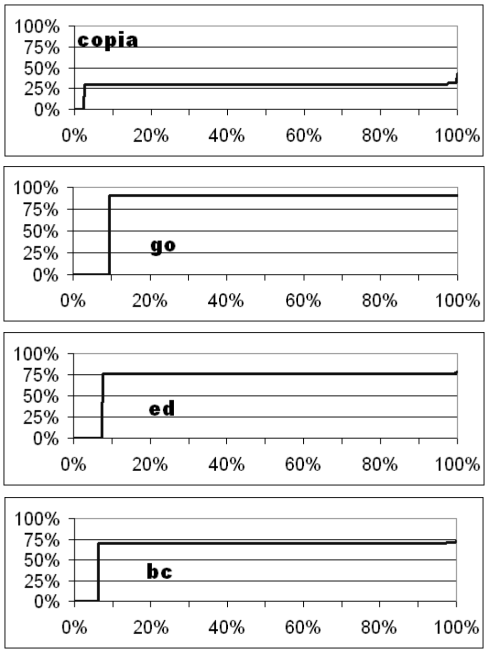
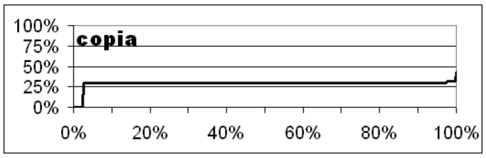
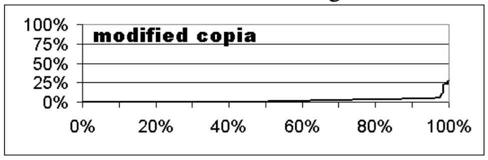


Figure 7. MSGs of Programs with Potentially Severe Dependence Pollution



Before refactoring



After Refactoring

Figure 8. Removing Dependence Pollution

ter is, weighed against the ease with which its dependence structure can be broken up and refactored.

Definition 3 *Dependence Pollution*
Dependence pollution occurs when a program contains a large dependence cluster which arises because of the use of some avoidable programming construction or feature.

Two possible sources of dependence pollution are Mutually Recursive Clusters (MRCs) and Capillary Data Flows (CDFs). Both of these are likely to lead to large dependence clusters and might be avoided by refactoring or otherwise transforming the program in which they occur.

Mutual recursion naturally leads to large dependence clusters because each function calls all the others, making the outcome of each function dependent upon the outcome of some call to each of the others. A Capillary Data Flow is a data flow which occurs between two large and otherwise unconnected clusters through a single variable. The variable acts as a small 'capillary vessel' along which the dependence 'flows' creating one large cluster from two or more, otherwise unconnected, sub-clusters.

6.2 Case Study: Dependence Pollution Found

The program `copia` implements a collection of analyses on an input table. As can be seen from Figure 7, the MSG contains a large plateau of slices which appear to have the same size; certainly a large dependence cluster and a candidate for dependence pollution. 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.

Inspection of the code reveals that `copia` has 234 small functions that call one large function, `seleziona`, which in turn, calls the smaller functions. This mutual recursion produces the large dependence cluster. Therefore, this is an example of a Mutually Recursive Cluster (MRC). The large function calls the smaller functions using a 234-case `switch` statement which selects the function to be called based upon the value of an integer, derived from the parameter passed to it. This use of a numeric function index plays a role very similar to that of a function pointer (function pointers are known to cause large static dependence [6, 31]).

This is a clear case of dependence pollution, because the use of a numeric function index and mutual recursion was entirely avoidable in this case. A simple refactoring was performed (by hand) to remove the need for the large `switch` statement and the associated calls through the numeric function index. Following this refactoring, the average size of slices of the program dropped from 13,033 nodes to 149 nodes, indicating a massive drop in overall dependence levels. Figure 8 shows the two MSGs before and after the refactoring. These two MSGs reveal the extent of dependence pollution in the original program. Not only is dependence significantly reduced by refactoring, but a more detailed landscape of dependence emerges, free from the 'tell-tale' cliff drop dependence cluster.

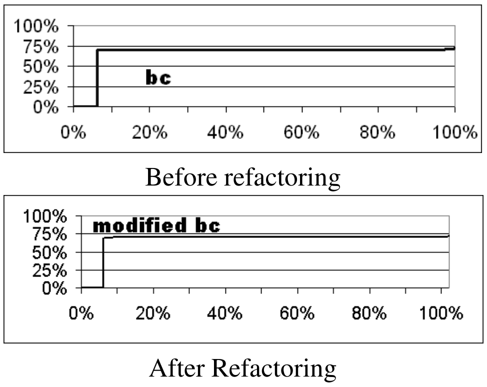


Figure 9. Dogged Dependence Pollution

6.3 Case Study: No Pollution Found

The program `bc` is a calculator program in which most of the code depends on the accumulator and the accumulator's value depends on most of the code. While this is clearly a property which emerges because of the nature of the application, it points to significant problems for maintenance because of the large-scale impact of almost any change. However, there will be little hope of refactoring to avoid the large dependence cluster as it is the nature of an accumulator–based calculator that much of the code depends upon one single variable. It is an example of a 'Capillary Data Flow (CDF)' with the accumulator and associated variables acting as the capillary variable that binds together the computations.

An attempt was made to refactor the program by finding the variable which contributed most to an increase in dependence. This analysis was performed by forming versions of the program, each of which was missing one global variable, to see which contributed most to dependence. The analysis revealed a global variable, `cmp_res`, which caused the biggest jump in dependence and so the program was refactored to remove this variable. Figure 9 shows the two MSGs before and after the refactoring. The refactoring attempt has provided almost no change in the profile depicted by the MSG. While the program may contain dependence pollution, it cannot be attributed to `cmp_res`.

6.4 Observations about `ed` and `go`

The program `ed` is a text editor. Many of the operations in the editor depend upon and affect the contents of the current document state. The program contains a set of operations, such as cut and paste, insert and delete, mark, etc. which all affect a common data structure. The data structure could be refactored to separate out the cursor location, the currently marked block and the text itself. This would allow commands to be grouped according to the parts of the (previously homogeneous) data structure which they affect and upon which they depend. This would have the effect of breaking the dependence cluster into several smaller clusters. As such, this observation suggests that there is dependence pollution due to Capillary Data Flow (CDF).

The game program, `go` is a strategy game in which most of the code is concerned with the state of the board. A related study found that only 13% of the slices were unique [15], confirming the finding here that this program contains a large dependence cluster. Like the program `ed`, the program `go` has a large dependence cluster as a result of CDF. In this case, the board state plays the role of the channel for capillary data flow, whereas in the editor it is the state of the document. However, unlike the text editor, this state cannot easily be refactored into sub-structures, so the large dependence cluster in the program `go` may be unavoidable and not, therefore, a case of dependence pollution.

7 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, but it applies to any study of properties of programs. Future work will be required to see if these results are replicated. Fortunately, however, the study did consider a large code base of twenty programs, covering a wide variety of different tasks including, applications, utilities, games, operating system code etc. 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.

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, the possible threats come from the potential for faults in the slicer and the values chosen of acceptable tolerance (which affects the verification study) and cluster size (which affects the validation study).

A mature and widely used slicing tool (CodeSurfer) was used to mitigate the first concern. For tolerance, the results showed that an overwhelming proportion (99.6%) of clusters of same size slices are within a tolerance of 1% of having the same content. For the applications of dependence clustering, this value of tolerance is well within acceptable limits. For cluster size, the value of 10% of the slices was chosen. That is, a cluster was deemed to be 'large' if it contained more than 10% of the slices of the program. Once again, this was a conservative choice of threshold, well within that which would be considered important in the application of dependence cluster analysis to maintenance problems. Furthermore, four of the programs studied were found to have clusters of well above this threshold (see Figure 7), suggesting that the evidence for the existence of large dependence clusters is extremely strong.

8 Related work

There has been much work concerned with clustering of one form or another [14, 28, 29]. However, the focus of this work has typically been 'clustering in the large', where the elements which occur in clusters are functions or entire modules. By comparison, the clusters of interest in the present paper are more fine-grained because the elements of interest are the nodes of the System Dependence Graph. This represents clustering at the statement level.

The present paper uses slicing to identify dependence clusters. There have been several surveys of slicing techniques, applications, and variations [5, 8, 12, 19, 33]. In particular, slicing has been applied to many problems related to software maintenance, for example re-engineering [10], program comprehension [13], debugging [27], testing [4, 17, 18], cohesion measurement [3], and impact analysis [16]. However, the present paper is the first to introduce dependence clusters and to suggest slicing as a mechanism for locating these clusters of dependence.

The work reported in the present paper follows the authors' more recent approach [7, 20] to using slicing as a *means to an end* rather than an end in itself. As such, the work draws from a flourishing well spring of research in program slicing, but seeks to use slicing as an enabling technology to inform other analyses. The slices, themselves, are not of interest, it is the combinations of slices and the way in which they can be used to identify dependence clusters that is of interest.

In adopting this approach, the paper follows other work on slicing, in which the slices are not the final product, but merely form a part of the overall analysis. For example, Bieman and Ott [3, 32] use slicing as part of a technique to measure the level of functional cohesion in a system as the proportion of a function which occurs in the overlap of all its primary slices. Dependence has also been proposed as the basis for measurement. For example, Black [9], uses dependence analysis to measure the ripple effect. The ripple effect is closely related to the work reported in the present paper, because the assertion that dependence clusters are harmful to maintenance is derived from their increased potential for intra–cluster ripple effects.

Other researchers have also used slicing as a means to an end, rather than an end in itself. For example Canfora, De Lucia and Munro [11] use dependence analysis as part of a wider approach to re-engineering, Korel and Rilling [24, 25] use dynamic slicing as a support for program comprehension, while Beszédes et al. [2] use it for maintenance of large programs and Krinke and Snelting [26] use a variant of static slicing for validation with respect to absence of inappropriate influences.

Another aspect of the work reported here is the use of slicing and dependence analysis as part of a visualisation for assisting in maintenance. This follows work by Balmas [1], who uses dependence analysis as part of a visualisation to assist with comprehension and maintenance activity. In her work, Balmas is concerned with managing the complexity of dependence graphs, by presenting them in a nested manner. In the present paper, the complexity dependence is addressed by focusing only upon the size of a slice as an approximation. This allows a single simple graph, the MSG, to summarise the whole dependence structure of a program for the purpose of identifying dependence clusters.

9 Summary and Future Work

This paper introduced the concept of a dependence cluster and shows how dependence clusters can be identified using a slice–based visualisation called the Monotone Slice-size Graph (MSG). Case study examples illustrate how MSGs can be used to locate dependence pollution. 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 (perhaps surprising) prevalence of these clusters in a large code base of 20 real-world programs. Future work will consider a wider class of programs to extend the empirical results presented in the paper and to attempt to categorise programs according to MSGs.

References

[1] F. 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, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

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

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

[4] D. W. Binkley. The application of program slicing to regression testing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 583–594. Elsevier, 1998.

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

[6] D. W. Binkley and M. Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *IEEE International Conference on Software Maintenance (ICSM 2003)*, pages 44–53, Amsterdam, Netherlands, Sept. 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

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

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

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

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

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

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

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

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

[15] K. Gallagher and D. Binkley. An empirical study of computation equivalence as determined by decomposition slice equivalence. In 10^{th} *IEEE Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, British Columbia, Canada, Nov. 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

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

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

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

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

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

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

[22] C. Jones. Mass-updates and software project management in is organizations- *http://www.artemis.it/artemis/artemis/lang_en/libreria/mass-updates.htm*, 1999. Accessed on Aug, 25 2003.

[23] S. Klusener and C. Verhoef. 9210: The zip code of another IT-soap. *Software Quality Journal*, 12(4):297 – 309, Dec. 2004.

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

[25] B. Korel and J. 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.

[26] J. Krinke and G. Snelting. Validation of measurement software as an application of slicing and constraint solving. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 661–675. Elsevier, 1998.

[27] J. R. Lyle and M. 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.

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

[29] S. Mancoridis, B. S. Mitchell, Y.-F. Chen, and E. 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.

[30] F. Miyara. how much noise is too much noise? *http://www.quiet.org/readings/howmuch.htm*, Aug. 1997. accessed 3rd February 2005.

[31] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing with dynamic points-to data. In W. G. Griswold, editor, *Proceedings of the* 10^{th} *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-02)*, pages 71–80, New York, Nov. 2002. ACM Press.

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

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

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