Coherent Dependence Cluster

Syed Saiful Islam

Thesis submitted for the degree of

Doctor of Philosophy

of

UCL

Department of Computer Science

University College London

University of London

2014

I, Syed Saiful Islam, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

_____

_____

Abstract

This thesis introduces *coherent dependence clusters* and shows their relevance in areas of software engineering such as program comprehension and maintenance. All statements in a coherent dependence cluster depend upon the same set of statements and affect the same set of statements; a coherent cluster's statements have 'coherent' shared backward and forward dependence.

We introduce an approximation to efficiently locate coherent clusters and show that its precision significantly improves over previous approximations. Our empirical study also finds that, despite their tight coherence constraints, coherent dependence clusters are to be found in abundance in production code. Studying patterns of clustering in several open-source and industrial programs reveal that most contain multiple significant coherent clusters. A series of case studies reveal that large clusters map to logical functionality and program structure. Cluster visualisation also reveals subtle deficiencies of program structure and identify potential candidates for refactoring efforts. Supplementary studies of inter-cluster dependence is presented where identification of coherent clusters can help in deriving hierarchical system decomposition for reverse engineering purposes. Furthermore, studies of program faults find no link between existence of coherent clusters and software bugs. Rather, a longitudinal study of several systems find that coherent clusters represent the core architecture of programs during system evolution.

Due to the inherent conservativeness of static analysis, it is possible for unreachable code and code implementing cross-cutting concerns such as error-handling and debugging to link clusters together. This thesis studies their effect on dependence clusters by using coverage information to remove unexecuted and rarely executed code. Empirical evaluation reveals that code reduction yields smaller slices and clusters.

Acknowledgements

I would like to thank my supervisor, Dr. Jens Krinke, for his patience, guidance, encouragement, advise and support during my time as his research student. I have been extremely lucky to have a supervisor who cares about my research and has always been there to answer my questions.

I would also like to thank my second supervisor, Prof. Mark Harman for his keen interest in my research and his guidance, encouragement and advise throughout the research. I would also like to thank Prof. David Binkley, who has spent many hours reading and discussing my research, and providing invaluable feedback. I would also like to thank members of CREST who have provided support, encouragement and feedback on my research.

I also want to express my gratitude to Amrin, my wife, for her continued encouragement and unwavering support. I am eternally indebted to her for her patience while sharing the experiences of ups and downs of my research and helping me get through it. I would also like to thank my parents who have invested so much of their life and made it possible for me to be here. I also want to thank my friends who have provided escapes from my research and have helped keep me motivated in achieving my goals.

Finally, I would like to thank the Department of Computer Science at UCL, the Engineering and Physical Sciences Research Council and the Higher Education Funding Council for England for supporting my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Program dependence analysis is a foundation for many activities in software engineering such as testing, comprehension, and impact analysis [Binkley, 2007]. For example, it is essential to understand the relationships between different parts of a system when making changes and the impacts of these changes [Gallagher and Lyle, 1991]. This has led to static [Yau and Collofello, 1985, Black, 2001], dynamic [Korel and Laski, 1988, Law and Rothermel, 2003] and blended (static and dynamic) [Ren et al., 2006, 2005] dependence analyses of the relationships between dependence and impact.

One important property of dependence is the way in which it may cluster. This occurs when a set of statements all depend upon one another, forming a dependence cluster. Within such a cluster, any change to any element potentially affects every other member of the cluster. Binkley and Harman [2005b] introduced dependence clusters and later demonstrated in a large scale empirical validation that large dependence clusters were (perhaps surprisingly) common, both in industrial and in open source system [Harman et al., 2009]. Their study of a large corpus of C code found that 89% of the programs studied contained at least one dependence cluster that consumed at least 10% of the program's statements. The average size of the programs studied was 20KLoC, so these clusters of more than 10% denoted significant portions of code. They also found evidence of super-large clusters: 40% of the programs had a dependence cluster that consumed over half of the program.

More recently, dependence clusters have been identified in other languages and systems, both in open source and in industrial systems [Acharya and Robinson, 2011]. Large dependence clusters were also found in Java systems [Beszédes et al., 2007, Savernik, 2007, Szegedi et al., 2007] and in legacy Cobol

systems [Hajnal and Forgács, 2011].

Since its inception, dependence clusters have been thought of as potential problem points as large inter-twined dependence is thought to make the task of program comprehension and maintenance difficult [Binkley and Harman, 2005b]. Binkley et al. [2008b] have regarded dependence clusters as bad code smells and considered them to be dependence 'anti-patterns' because the high impact of changes that may lead to problems for on-going software maintenance and evolution [Savernik, 2007]. To this end there has been work that studies the link between impact analysis and dependence clusters [Acharya and Robinson, 2011, Jasz et al., 2012].

There has been interesting work that investigates the relationship between program faults, software metrics, and dependence clusters [Black et al., 2009]. A possible link between dependence clusters and program faults has also been suggested [Black et al., 2006]. As a result, approaches have been proposed that identify linchpin vertices (responsible for holding clusters together) in both traditional dependence clusters [Binkley and Harman, 2009, Binkley et al., 2013a] and SEA-based clusters [Beszédes et al., 2007].

Despite their potentially negative impact, dependence clusters are not well understood. Cluster analysis is complicated because inter-procedural dependence is *non-transitive*; thus, the definition of a dependence cluster is subtle, even surprising. One implication of this complexity is that past studies have focused on the internal aspects of dependence clusters and thus largely ignored the extra-cluster dependences (both in to and out of the cluster). Non-transitivity of dependence means that the statements in a traditional dependence cluster, though they all depend on each other, may not each depend on the same set of statements, nor need they necessarily affect the same set of statements.

This thesis introduces and empirically studies *coherent dependence clusters*. In a coherent dependence cluster all statements share identical intra-cluster and extra-cluster dependence. A coherent dependence cluster is thus more constrained than a general dependence cluster. A coherent dependence cluster retains the essential property that all statements within the cluster are mutually dependent, but adds the constraint that all incoming dependence must be identical and all outgoing dependence must also be identical. That is, all statements within a coherent cluster depend upon the same set of state-

ments outside the cluster and all statements within a coherent cluster affect the same set of statements outside the cluster.

This means that, when studying a coherent cluster, we need to understand only a single external dependence context in order to understand the behaviour of the entire cluster. For a dependence cluster that fails to meet the external constraint, each statement of the cluster may have a different external dependence context, because inter-procedural dependence is not transitive.

It might be thought that very few sets of statements would meet these additional coherence constraints or that, where such sets of statements do meet the constraints, there would be relatively few statements in the coherent cluster so-formed. Our empirical findings provide evidence that this is not the case, coherent dependence clusters are common and they can be very large. This finding provides a new way to investigate the dependence structure of a program and the way in which it clusters.

This thesis looks at dependence clusters in a new light. Unlike previous understanding, this thesis shows that coherent clusters are not necessarily problems hindering code maintenance and comprehension, instead clusters are found to depict logical structure of a program. The thesis shows that visualisation of coherent clusters can help reveal these logical structures and can also help identify potential structural problems and refactoring opportunities.

A study on the link between program faults and coherent clusters find no evidence to suggest that coherent clusters have a link to program faults. On the other hand, a study of system evolution finds that coherent clusters remain surprisingly stable during evolution of systems. These two studies provide further support for the central theme of the thesis where we show that coherent clusters are not potential problems but occur naturally in programs and depict the logical structure of the program.

Finally, the thesis presents a study where coverage information is used to reduce programs by removing unexecuted and rarely executed code, leaving code that is of most interest to developers and maintainers. This allows for improved static analysis by excluding many cross-cutting concerns. For example, the average slice size of the reduced programs drop by 30% when compared to the original versions. An evaluation of coherent clusters show that the clusters in the reduced version capture more fine-grained logical structures in programs.

1.1 Problem Statement

Analyse and visualize coherent dependence clusters and to study their impact on program comprehension and maintenance.

1.2 Research Questions

Formally, this thesis addresses the following fourteen research questions. This section gives a brief description of these questions and their relationship.

The first three research questions *RQ1.1–RQ1.3* form the core validation study of the thesis and are addressed in Chapter 4. *RQ1.1* validates the experimental setup and its methodology. *RQ1.2* asks whether coherent clusters are indeed common in production systems, making them worthy of further study. *RQ1.3* studies the conservatism introduced by using approximations for slice-based clusters.

RQ1.1 *How precise is hashing as a proxy for comparing slices?*

RQ1.2 *How large are coherent clusters that exist in production source code?*

RQ1.3 *How conservative is using the same-slice relationship as an approximation of slice-inclusion relationship?*

Research question *RQ2.1* uses graph-based cluster visualisations to study the cluster profile for the subject programs. Patterns of clustering visible from the cluster profile of the graph-based visualisation are identified and form the answer to this question. Many of the later research questions are based on identifying changes in the cluster profile making this research question of identifying patterns important. *RQ2.2* studies the impact of various pointer analysis algorithms on the size of slices and how this affects the coherent clusters found in programs. Both *RQ2.1* and *RQ2.2* are addressed in Chapter 5.

RQ2.1 *Which patterns of clustering can be identified using graph-based cluster visualisation?*

RQ2.2 *What is the effect of pointer analysis precision on coherent clusters?*

Research question *RQ3.1* forms a major contribution of this thesis. *RQ3.1* is answered using a series of four case studies, which together show that coherent clusters map to logical constructs of programs. This is achieved by

employing the *decluvi* cluster visualisation tool to analyse the mapping between clusters and functions of the programs. *RQ3.2* presents a quantitative study of how functions and coherent clusters map to each other. Both these questions are addressed in Chapter 6.

RQ3.1 *Which structures within a program can coherent cluster analysis reveal?*

RQ3.2 *How do functions and clusters overlap, and do overlap and size correlate?*

RQ4.1 looks explicitly at the relationship between coherent clusters and program faults. It finds no link between faults and coherent clusters. *RQ4.2* on the other hand studies the changes of coherent cluster profile during system evolution and finds them to be very stable. Answers to *RQ4.1* and *RQ4.2* provide additional support to the answer of *RQ3.1* and bolster the central claim of the thesis that dependence clusters map to logical program structure. *RQ4.3* studies the implications of inter-cluster dependence and shows how they may be leveraged to identify larger dependence structures. *RQ4.4* looks at the existence of coherent clusters in object-oriented paradigm by studying Java programs. *RQ4.1–RQ4.4* are addressed in Chapter 7.

RQ4.1 *How do program faults relate to coherent clusters?*

RQ4.2 *How stable are coherent clusters during system evolution?*

RQ4.3 *What are the implications of inter-cluster dependence between coherent clusters?*

RQ4.4 *Are coherent clusters prevalent in object-oriented programs?*

The final set of research questions *RQ5.1–RQ5.3* are related to the code reduction framework aimed at reducing the size of static slices and clusters. *RQ5.1* studies the impact of different test suites on static slicing and dependence clusters. *RQ5.2* analyses the existence and change in clustering profile following code reduction, whereas *RQ5.3* ascertains whether code reduction improves the quality of the clustering. *RQ5.1–RQ5.3* are addressed in Chapter 8.

RQ5.1 *What is the impact of different test suites on static program slices and dependence clusters in coverage-based reduced programs?*

RQ5.2 *How large are coherent clusters that exist in the coverage-based reduced programs and how do they compare to the original version?*

RQ5.3 *Which structures within a coverage-based reduced program can coherent cluster analysis reveal and how do they compare to the original version?*

1.3 Contributions

The primary contributions of the thesis are as follows:

1. Definition of *coherent dependence clusters*

2. An algorithm for efficient and accurate clustering

3. Empirical evaluation of the impact of pointer analysis precision on clustering

4. Empirical evaluation of the frequency and size of coherent dependence clusters in production grade software

5. A cluster visualisation tool for graph-based and interactive multi-level visualisation of dependence clusters

6. A series of case studies showing that coherent clusters map to logical program structure

7. Studies into the relationship between software bugs and software evolution with coherent dependence clusters

8. Identification of inter-cluster dependence and highlighting how coherent clusters form the building blocks of larger program dependence structures and can support reverse engineering

9. Introduction of a framework for coverage-based code reduction to eliminate unwanted cross-cutting concerns or other features in order to reduce size of static slices and clusters.

1.4 Publications

Publications on *Coherent Dependence Clusters*:

- Syed Islam, Jens Krinke, David Binkley, and Mark Harman. Coherent dependence clusters. In *PASTE '10: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 53–60. ACM Press, 2010.

- Syed Islam, Jens Krinke, and David Binkley. Dependence cluster visualization. In *SoftVis '10: 5th ACM/IEEE Symposium on Software Visualization*, pages 93–102. ACM Press, 2010.

- Syed Islam, Jens Krinke, David Binkley, and Mark Harman. Coherent clusters in source code. *Journal of Systems and Software*, 88(0):1 – 24, 2014.

Publications on *Dependence Clusters*:

- David Binkley, Mark Harman, Youssef Hassoun, Syed Islam, and Zheng Li. Assessing the impact of global variables on program dependence and dependence clusters. *Journal of Systems and Software*, 83(1):96–107, 2010.

- David Binkley, Nicholas Gold, Mark Harman, Syed Islam, Jens Krinke, and Zheng Li. Efficient identification of linchpin vertices in dependence clusters. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 35(2):1–35, July 2013.

1.5 Research Methodology

This thesis uses both Quantitative research and Qualitative research. The techniques are used to complement each other rather than compete as suggested by Wohiln et al. [2003]. The quantitative research broadly looks at ascertaining the presence and frequency of dependence clusters in real-world systems. As part of the quantitative research, experiments were conducted to replicate results of previous studies and validate current results against them. As this thesis is the first to present *Coherent Clusters*, it is important to replicate previous results from closely related work on dependence clusters for the purpose

of validation and comparison. An extension of the quantitative studies also looks into the effect of other dependence properties of the clusters.

The qualitative research looks in-depth into four of the subject programs to ascertain different properties and implications of the presence of coherent clusters in them. The qualitative research is done by considering the production systems as case studies where their clustering is studied and compared to logical program structure.

1.6 Thesis Structure

The remainder of this thesis is organised into eight chapters. Chapter 2 provides a literature review on clustering and work related to dependence clusters. Chapter 3 provides the necessary background on dependency graphs, program slicing and various instantiations of dependence clusters. Chapter 4 introduces coherent clusters and its various slice based instantiations. Chapter 5 introduces various graph-based visualisations and the *decluvi* cluster visualisation tool. Chapter 6 presents a series of four case studies for qualitative and quantitative studies of mapping between coherent clusters and functions. Chapter 7 presents studies on the link between clusters and bugs, clusters and system evolution, inter-cluster dependence and existence of clusters in object-oriented programs. Chapter 8 presents the coverage-based code reduction framework and its impact on static slicing and clustering. Finally, Chapter 9 presents threats to validity, highlights future work and draws conclusions of the thesis.

Chapter 2

Background

2.1 Software Clustering

Clustering is the process of grouping entities such that entities within a group are similar to one another and different from those in other groups. The similarity between entities is determined based on their features. Clustering is a very important activity widely researched and has a wide variety of applications in various different domains [Anderberg, 1973, Everitt, 1974, Romesburg, 1984], including software engineering.

According to Anquetil et al. [1999] and Maqbool et al. [2007] the following need to be considered before clustering can be performed:

Entities: *What is to be grouped?*

It is necessary to group together entities that result in a partition (clustering result) which is of interest to software engineers. The goal is to create subsystems that contain entities that are related based on a specific set of criteria. In case of software clustering these entities are files, functions, global variables, types, macros statement block, syntactic units etc. For large systems, the desired entities may be abstract, and modelled using architectural components such as subsystems and subsystem relations. For even larger systems, hundreds of subsystems themselves might be collected into other subsystems, resulting in a subsystem hierarchy [Lakhotia, 1997].

Selection of features: *What characteristics will be considered?*

The entities selected for clustering will have a lot of features that they exhibit. A set of features, which can be used as a basis for judging similarity of entities, must be selected for the clustering process. In case

of software clustering, both formal (function calls, file inclusion, variable referencing, type referencing etc.) and informal features (comments, identifier names, directory path, LOC, last modification time, developer, change requests) have been widely used as basis of clustering [Andritsos and Tzerpos, 2005, Lakhotia, 1997]. More recently, the use of dynamic information (for example, the number of function calls made at run-time) has also been explored [Tzerpos, 2005].

Entities' coupling: *How will the relationship between entities be represented?*
Once the entities and the features have been selected, the next step is to decide how to group them into cohesive units. There are two approaches, 'direct link' and 'sibling link'. The first puts together entities based on direct relationship between the entities, while the second puts together entities that exhibit the same behaviour. The approaches are discussed in further details in Section 2.2.

Clustering algorithm *How will the clusters be grouped?*
There are many different algorithms that are successful for clustering, however all of these may not be applicable to software clustering and may not produce results that are of interest from a software engineering point of view. An example is the grid method, where entities are clustered based on their geographical position within a grid structure [Asif et al., 2009]. This will not be useful as software artefacts do not have any spacial properties.

Software clustering can thus be described as the process of gathering software entities (such as files, functions, etc.) that compose the system into meaningful (cohesive) and independent (loosely coupled) subsystems.

There have been many surveys of the vast amount of techniques applied in the field of software clustering. Many frameworks have been proposed for classifying these techniques. It is beyond the scope of the review presented here to consider all the various approaches and classifications in detail. For our review we consider in detail the two most popular approaches in software clustering: hierarchical clustering (Section 2.3) and search-based clustering (Section 2.4). Other approaches are briefly outlined in Section 2.5. Below, we briefly note major surveys, frameworks and classifications of the area.

From the rich literature on clustering [Xu and Wunsch, 2005, Ander-

berg, 1973, Everitt, 1974, Romesburg, 1984], a set of suitable clustering algorithms that may be applied to software clustering process was surveyed by Wiggerts [1997], Mitchell [2004] and Tzerpos et al. [1998]. Maqbool and Babri [2007], Anquetil and Lethbridge [1999] provide a detailed survey of hierarchical techniques.

Lakhotia [1997] gives a survey on subsystem classification techniques and provides a unified framework for entity description and clustering methods in order to facilitate comparison between various subsystem classification techniques. Koschke [2002] presents a classification in his PhD thesis where he considers 23 different approaches and classifies them based on relationships between entities. Other studies [Armstrong and Trudeau, 1998, Storey et al., 2000] look at reverse engineering applications and provide evaluation and comparison of tools. Ducasse and Pollet [2009] provide a comprehensive taxonomy of the software architecture recovery field and evaluate 34 techniques/tools based on various capabilities.

2.2 Direct and Sibling link approaches

Before proceeding further we must discuss the representation of the relationships between the various entities. This is of particular importance to us as they will lead to different clustering approaches that we will review, namely hierarchical clustering and search-based clustering. Tzerpos [1998] distinguishes the following kinds of relationships:

Direct relationship: *Entities depend on each other.*

In this case, entities and relationships are represented as a graph, where the nodes are the entities and the edges are the relations. Where multiple relations exist, the graph will have multiple kinds of edges, also weighted features may be represented in a weighted graph. The direct link approach has an appealing simplicity, for example, if a function calls another function or a file includes another file they are related to each other. However, this often brings us back to the graph partitioning problem which is known to be NP-Hard [Garey and Johnson, 1990]. This direct link approach is thus mostly and commonly used by graph-theoretic clustering and search-based techniques [Hutchens and Basili, 1985, Lung et al., 2004, Mancoridis et al., 1998, Mitchell and Mancoridis, 2001a, Muller et al., 1993] (Section 2.4).

Sibling link approach: *Features that entities share.*

This approach is based on representing the commonality of features of the different entities rather than a direct relationship between the entities themselves [Anquetil and Lethbridge, 2003, Anquetil et al., 1999, Schwanke, 1991]. Whereas in the previous approach entities could not be clustered unless there was a link between them, using this approach two entities that have no link between them may also be clustered based on some feature that they have in common. Sibling link approach is also known to produce better clustering results than direct link approach [Anquetil and Lethbridge, 1999, Kunz, 1993, Ricca et al., 2004]. We describe the similarity measure that can be applied to this approach in Section 2.3.1.

2.3 Hierarchical Clustering

Hierarchical clustering algorithms produce a nested decomposition of the system into subsystems, which in turn may be broken down into smaller subsystems and entities. At one end of the hierarchy is the partition where each entity is in a different cluster and at the other end the partition where all the entities are in the same cluster. The multi-level architectural views facilitates both architectural and implementation understanding as they provide both detailed view and abstractions at various levels that intuitively match the structural decomposition of software [Shtern and Tzerpos, 2004]. Hierarchical clustering methods can be divided into two major categories on the basis of the strategy it adapts to cluster the entities, divisive (top down) or agglomerative (bottom up).

Divisive (Top-Down): Divisive hierarchical clustering algorithms start with one cluster containing all the entities representing the systems and use the top down strategy to iteratively divide the cluster into smaller ones, until each entity form a cluster on its own or satisfy certain termination condition. Such algorithms however suffer from excessive computational complexity and are unpopular, as it has to consider an exponential number of partition possibilities at every step ($2^{n-1} - 1$ possibilities in the first step) [Wiggerts, 1997].

Agglomerative (Bottom-up): Agglomerative hierarchical clustering starts by placing each entity in its own cluster and then iteratively merges the

most similar clusters based on some criteria to get larger and larger clusters until certain termination condition is satisfied or all the entities are in one cluster. At each level of the merging, a clustering result (partition) of the system is obtained and represents a solution. The partitions achieved at initial stages contain more clusters and provide a detailed view whereas the ones at later stages contain fewer clusters and provide a more high-level view. Hierarchical agglomerative clustering algorithms are by far the most popular choice when it comes to software architecture recovery [Anquetil et al., 1999, Anquetil and Lethbridge, 2003, Hutchens and Basili, 1985, Schwanke, 1991]. There are many agglomerative hierarchical algorithms that have been applied to the problem of software architecture recovery and vary on the strategy used to decide on which clusters to merge based on similarity of entities contained within the clusters. The similarity measures are described in Section 2.3.1 and the various cluster merging strategies are described in Section 2.3.2.

2.3.1 Sibling link similarity measures

The sibling link approach takes into consideration features of the entities, and two entities with the most features in common are considered to be most similar. To be able to do this we must be able to decide that "entity a is more similar to entity b than entity c", based on some kind of a measure of commonality. Similarity metrics thus compute a coupling value between two entities and the choice of the measure is important as the choice of the similarity measure has more influence on the result than the clustering algorithm [Jackson et al., 1989]. There is a large number of similarity measures/metrics that are found in the literature and can be grouped into the four categories [Anquetil and Lethbridge, 1999, Maqbool and Babri, 2007]: association coefficients, distance measures, correlation coefficients and probabilistic coefficients.

Association coefficients

These compare the features that two entities have in common considering whether features are present or not. The idea behind association metrics is very intuitive: the more relevant feature matches there are between two entities under comparison, the more similar the two entities are. Association coefficients for two entities i and j are expressed in terms of the number of features which match and mismatch in the entities, as shown in Table 2.1. In the matrix, a represents the number of features i and j have in common, b rep-

resents the number of features unique to j, c represents the number of features unique to i, and finally d represents the number of features missing from both i and j. As the features are represented in binary a, b, c and d are also known as $1-1$, $1-0$, $0-1$ and $0-0$ matches.

	entity j **1**	entity j **0**
entity i **1**	a	b
entity i **0**	c	d

Table 2.1: Association matrix

Association coefficients based similarity metrics vary mainly in two places:

1. *The handling of* $0-0$ *matches.* Some measures do not take into account $0-0$ matches and some assign it lower weights than the other three matches. In the case of software, features are considered to be asymmetric, that is their presence may indicate similarity, but their absence may not tell us anything. For example, it is significant whether two functions use the same global variable (a $1-1$ match), since this indicates that they may be similar. But, the fact that two functions do not use a global variable (a $0-0$ match) does not indicate a similarity between them; hence, a $1-1$ match is more significant than a $0-0$ match.

2. The weight that is applied to matches $1-1$ and mismatches $0-1$ or $1-0$. In other words, should features unique to mismatches play a bigger role in deciding the dissimilarity between entities or the other way around.

Some well-known association based similarity measures are shown in Table 2.2. For example the Jaccard coefficient is the ratio of $1-1$ matches, without considering $0-0$ matches. The simple matching coefficient counts both $1-1$ and $0-0$ matches as relevant. The Sorenson coefficient is similar to the Jaccard coefficient, but the number of $1-1$ matches, a, is given twice the weight.

Table 2.3 illustrates an example with three entities with eight attributes. A 1 entry indicates that the attribute is present in the corresponding entity, while a 0 indicates the absence of the feature. Entity x in Table 2.3, consists of attributes 1, 3, 4, and 8; entity y is positive to attributes 1, 2, 3, and 7. Entity x and y share two common attributes 1 and 3, or these entities have two $1-1$ (a) matches. Similarly, there are $1-0$ (b), $0-1$ (c), and $0-0$ (d) attribute matches

Similarity measure (sim)	*Formula*
Simple matching coefficient	$\frac{a+d}{a+b+c+d}$
Jaccard coefficient	$\frac{a}{a+b+c}$
Sorenson-Dice	$\frac{2a}{2a+b+c}$
Rogers and Tanimoto	$\frac{a+d}{a+2(b+c)+d}$
Sokal and Sneath	$\frac{a}{a+2(b+c)}$
Gower and Legendre	$\frac{a+d}{a+1/2(b+c)+d}$
Russel and Rao Coefficient	$\frac{a}{a+b+c+d}$
Yule coefficient	$\frac{ad-bc}{ad+bc}$

Table 2.2: Well-known similarity measures (association coefficients) [Romesburg, 1984]

Entity	Attribute							
	1	2	3	4	5	6	7	8
x	1	0	1	1	0	0	0	1
y	1	1	1	0	0	0	1	0
z	0	1	1	0	1	0	1	0

Table 2.3: Data matrix

between the two entities. Therefore, the association coefficients for entities x and y are $a = 2$, $b = 2$, $c = 2$, and $d = 2$. Similarly, for entities x and z, we obtain $a = 1$, $b = 3$, $c = 3$, and $d = 1$; for entities y and z, $a = 3$, $b = 1$, $c = 1$, and $d = 3$. By applying the Sorenson matching coefficient to the example in Table 2.3, we get $sim_{xy} = (2 * 2)/(2 * 2 + 2 + 2) = 1/2$. Likewise, $sim_{xz} = (2*1)/(2*1+3+3) = 1/4$ and $sim_{yz} = (2*3)/(2*3+1+1) = 3/4$. For this particular data representation, the higher a coefficient, the more similar the two corresponding entities represented. Hence, entity y and z are the most similar pair, since the resemblance coefficient sim_{yz} is the largest.

Results in the literature [Anquetil et al., 1999, Maqbool and Babri, 2007] show that for software clustering it is best not to consider zero-dimensions, Jaccard association coefficient and the Sorensen-Dice (which do not consider zero-dimensions) both achieve good results.

Distance coefficient

Instead of measuring features in binary (qualitatively), they can also be measured quantitatively on a ordinal scale. For example, $x = (3, 5, 10, 2)$ indicates that entity x contains four attributes with values of 3, 5, 10, and 2, respectively. To calculate the resemblance coefficients based on the quantitative input data,

distance measures are commonly used. Distance metrics measure the dissimilarity of entities as opposed to association coefficient that measure similarity. The greater the outcome the more dissimilar the entities are. The distance between two entities is zero iff the entities have the same score on all features. Common distance measures are given in Table 2.4, where x and y represent points in the Euclidean space R^s.

Table 2.4: Common Distance Measures [Maqbool and Babri, 2007]

Correlation coefficients

Correlation coefficients are originally used to correlate features. The most popular coefficient of this sort is the Pearson product-moment correlation coefficient. The value of a correlation coefficient lies in the range from -1 to 1. A value of 0 means not related at all, 1 means completely related and negative implies a inverse trend. They are not commonly found in the literature except one reference where in an experiment Anquetil and Lethbridge [1999] report them to yield similar results to Jaccard measures.

Probabilistic coefficients

Probabilistic measures are based on the idea that agreement on rare features contribute more to the similarity between two entities than agreement on features which are frequently present. So probabilistic coefficients take into account the distribution of the frequencies of the features present over the set of entities [Wiggerts, 1997]. However, although they were listed in several literature as an option, they were not studied or used to perform software clustering.

By far the most popular choice amongst researchers is the use of associative coefficients for similarity measures in the form of Jaccard coefficients and Sorsen-Dice coefficients. For dissimilarity measures the Euclidean Distance is preferred [Maqbool and Babri, 2007].

2.3.2 Agglomerative Hierarchical Clustering algorithms

An agglomerative hierarchical clustering method is a sequence of operations that incrementally groups similar entities into clusters. The sequence begins with each entity in a separate cluster. At each step, the two clusters that are closest to each other are merged and the number of clusters is reduced by one. So, in the beginning there are n clusters with each of the n entities in one of the clusters. At the end of the process there is only one cluster which contains all the n entities. The generic algorithm for agglomerative hierarchical clustering is listed below as Algorithm 1.

Algorithm 1: Agglomerative hierarchical clustering algorithm

```
Put each entity of the system into its own cluster;
Calculate similarity between every pair of clusters within the system;
repeat
    Merge the two most similar clusters to form one cluster;
    Re-calculate similarity between this newly formed cluster and all
    other clusters;
until more than one cluster left;
```

The similarity/dissimilarity measures that we looked at in the last section allows for similarity between entities to be calculated. However, when it comes to calculating similarity between clusters, the metrics cannot be applied directly as clusters contain multiple entities. The various agglomerative hierarchical algorithms in the literature use different strategies for this purpose. The strategy used by the four basic hierarchical algorithms (also called group

measures [Maqbool and Babri, 2007]) are presented next, where E_i, E_m, and E_o represent entities, and E_{mo} represents the cluster formed by merging entities E_m and E_o. Also, E_i is considered to be in a singleton cluster.

Single linkage (*SL*)

Using the single linkage or nearest neighbour strategy the similarity between two clusters are calculated as:

$$SL(E_i, E_{mo}) = Max(sim(E_i, E_m), sim(E_i, E_o))$$

It defines the similarity measure between two clusters as the maximum similarity among all pair entities in the two clusters.

Complete linkage (*CL*)

Using the complete linkage or furthest neighbour strategy the similarity between two clusters are calculated as:

$$CL(E_i, E_{mo}) = Min(sim(E_i, E_m), sim(E_i, E_o))$$

It defines the similarity measure between two clusters as the minimum similarity among all pair entities in the two clusters.

Weighted average linkage (*WAL*)

Clusters may not have the same number of entities and thus to achieve a more uniform result, weighted average linkage assigns different weights to entities depending on which cluster they belong to. The similarity between two clusters using WAL is calculated as:

$$WAL(E_i, E_{mo}) = 1/2(sim(E_i, E_m)) + 1/2(sim(E_i, E_o))$$

The similarity measure between two clusters is calculated as the simple arithmetic average of similarity among all pair of entities in the two clusters.

Unweighed average linkage (*UWAL*)

Unweighed average linkage similarity measure is also an average link measure but uses the size of both the clusters, therefore all entities have the same weight (i.e. they are not weighted). UWAL similarity is measured as:

$$UWAL(E_i, E_{mo}) = \frac{sim(E_i, E_m) * size(E_m) + sim(E_i, E_o) * size(E_o)}{size(E_m) + size(E_o)}$$

Other algorithms

The four algorithms presented above lead to a large number of arbitrary decisions, especially toward the latter half of the clustering process. The arbitrary decision arises from the algorithm being unable to decide which clusters to combine when more than one cluster have the same similarity measure. Arbitrary decisions in hierarchical algorithms adversely affect the quality of clustering results [Maqbool and Babri, 2007]. This is because in hierarchical algorithms once an entity is assigned to a cluster, it cannot be reassigned to a new cluster.

The Combined algorithm (CA) [Saeed et al., 2003], Weighted combined algorithm (WCA) [Maqbool and Babri, 2004], and LIMBO [Andritsos and Tzerpos, 2005], three recently proposed hierarchical clustering algorithms adopt a two-step approach to determine the similarity of a cluster with existing clusters. As the first step, they associate a new feature values with the newly formed cluster. This new feature values are based on the feature values of the constituent entities. At the second step, similarity between the cluster and existing clusters is recomputed. This approach allows useful feature-related information to be retained, thus reducing the number of arbitrary decisions and improving clustering results. Adnan et al. [2008] presents an adaptive clustering algorithm, which is based on the same idea of minimising arbitrary decisions but does not implement a two step process. Instead it is a hybrid approach that switches between the various algorithms trying to minimise the number of arbitrary decisions.

2.3.3 Summary

It is relevant to note that although the aim of clustering methods is to extract natural clusters in the data, it is quite possible that a method imposes a structure where no such structure exists [Choi and Scacchi, 1990, Muller and Uhl, 1990]. Different algorithms thus produce different clusters. For example, SL is known to favour non-compact but more isolated (less coupled) clusters whereas, CL usually results in more compact (cohesive) but less isolated clusters.

There has been numerous studies into the quality of the clustering produced by the algorithms which report, WCA to be the best and CA second best [Maqbool and Babri, 2007, Lung et al., 2006]. The rest are graded from CLA, WLA, UWLA down to SLA [Anquetil et al., 1999, Davey and Burd, 2000] in order of their performance.

2.4 Search-based clustering

As opposed to hierarchical algorithms that work on the sibling linking approach, this class of algorithms are applied to situations where the direct link approach is used. This approach employs a graph of entities as nodes and edges linking the entities representing direct relationships between the entities, such as, function calls, file inclusion etc. The optimal algorithm to perform the partitioning would investigate every possible partition of the system and choose the best one based on some criteria. However, this approach faces a combinatorial explosion as the number of possible partitions is extremely large, rendering the algorithm impractical. To overcome this problem, optimisation algorithms are employed. These algorithms start with an initial partition and try to modify it in an attempt to optimise a criterion that represents the quality of a given partition. We will describe two such algorithms as implemented by the Bunch tool [Mancoridis et al., 1999, Mitchell and Mancoridis, 2007].

2.4.1 Module Dependence Graph

The first step of the clustering process is representing the system entities and their inter-relationships as a module dependency graph (MDG) [Mancoridis et al., 1999]. Formally $MDG = (M, R)$ is a graph where E is a set of entities of a software system, and $R \subseteq M \times M$ is the set of ordered pairs $\langle u, v \rangle$ that represent the source-level dependencies (e.g. procedural invocation, variable access, file inclusion) between entities u and v of the system. Such MDGs in the literature are automatically constructed using source analysis tools such as CIA [Chen et al., 1990] for C programs and ACACIA [Chen et al., 1995] for C++.

2.4.2 Fitness function

The fitness function defines the quality of a partition (clustering result). Search-based algorithms try to improve this value in order to achieve a better result. The fitness functions used in the literature aim to improve cohesion and reduce coupling as good software design dictates that subsystems should exhibit high cohesion and have low coupling with other subsystems [Mancoridis et al., 1998]. Cohesion and coupling in MDGs are defined in terms of intra-connectivity and inter-connectivity of clusters.

Intra-connectivity

Intra-connectivity (A) is a measure of the connectivity between the entities that are grouped together in the same cluster. A high degree of intra-connectivity indicates good subsystem partitioning because the entities grouped within a common subsystem are inter-dependent. A low degree of intra-connectivity indicates poor subsystem partitioning because the entities assigned to a particular subsystem share few dependencies (limited cohesion). Intra-connectivity measurement A_i of cluster i consisting of n_i entities and μ_i intra-edge dependencies is defined as:

$$A_i = \frac{\mu_i}{{n_i}^2}$$

This measure is a fraction of the maximum number of intra-edge dependencies that can exist for cluster i, which is ${n_i}^2$. The value of A_i is bounded between the values of 0 and 1. A_i is 0 when entities in a cluster do not have any dependency between them; A_i is 1 when every entity in a cluster is dependent on all other entities within the cluster.

Inter-connectivity

Inter-Connectivity (E) is a measure of the connectivity between two distinct clusters. A high degree of inter-connectivity is an indication of poor subsystem partitioning. A low degree of inter-connectivity indicates that the individual clusters of the system are largely independent of each other. Inter-connectivity $E_{i,j}$ between clusters i and j consisting of n_i and n_j entities, respectively, with $\varepsilon_{i,j}$ inter-edge dependencies is defined as:

$$E_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \dfrac{\varepsilon_{i,j}}{2n_i n_j} & \text{if } i \neq j \end{cases}$$

The inter-connectivity measurement is a fraction of the maximum number of inter-edge dependencies between clusters i and j ($2n_i n_j$). This measurement is bound between the values of 0 and 1. $E_{i,j}$ is 0 when there are no cluster-level dependencies between subsystem i and subsystem j; $E_{i,j}$ is 1 when each entity in subsystem i depends on all of the entities in subsystem j and vice-versa.

Modularisation quality

The fitness function is defined as modularization quality (MQ). It establishes a trade-off between inter-connectivity and intra-connectivity that rewards the

creating of highly cohesive clusters and penalises the creation of too many inter-cluster dependencies. This trade-off is achieved by subtracting the average inter-connectivity from the average intra-connectivity. The MQ for a partition with k clusters, where A_i is the inter-connectivity of the i^{th} cluster, and $E_{i,j}$ is the interconnectivity between the i^{th} and j^{th} clusters, is defined as:

$$MQ = \begin{cases} \dfrac{\sum_{i=1}^{k} A_i}{k} - \dfrac{\sum_{i,j=1}^{k} E_{i,j}}{\frac{k(k-1)}{2}} & k > 1 \\ A_i & k = 1 \end{cases}$$

MQ is thus bound between -1 (no cohesion within the clusters) and 1 (no coupling between the clusters). Other module clustering quality measure such as EVM has also been experimented with and were found to be similar in nature to MQ [Harman et al., 2005]

2.4.3 Hill-Climbing clustering algorithm

The Hill Climbing clustering algorithm starts with a random partition and relies on moving entities between the clusters of the partition to improve the MQ of the result. The moving of entities between the cluster is accomplished by generating a set of neighbouring partitions. Two partitions are said to be neighbours when there is only one difference between them, that is, a single entity is placed in different clusters within the two partitions. The generic hill-climbing algorithm is listed below as Algorithm 2.

Algorithm 2: Hill-climbing algorithm

Generate an initial random partition of the system;
repeat
| Replace current partition with a better neighbouring partition;
until *no further "improved" neighbouring partitions can be found*;

A better neighbouring partition is discovered by going through the set of neighbouring partitions of the current partition, iteratively, until a partition with a higher MQ is found. During discovery of a better neighbour there are two strategies commonly employed, choosing the first neighbour partition with better MQ as the partition of the next iteration or examine all neighbouring partitions and pick the one with the largest MQ as a base of the next itera-

tion. The first approach is called next ascent hill-climbing (NAHC) [Mancoridis et al., 1999], runs slow but often provides good results. The second approach is called steepest ascent hill-climbing (SAHC) [Mancoridis et al., 1998], is much faster but runs the risk of obtaining sub-optimal results that are not useful.

A well-known problem with hill-climbing algorithms is that certain initial starting points may converge to poor/unacceptable solutions (local maximum). One approach (multiple hill-climbing [Mahdavi et al., 2003]) to solving this problem is to repeat the experiment many times using different initial random partitions. The experiment that results in the largest MQ is presented as the sub-optimal solution. As the number of experiments increases, the probability of finding the globally optimal partition also increases at the expense of more computation time. The other approach (simulated annealing [Mitchell and Mancoridis, 2007]) enables the search algorithm to accept, with some probability, a worse variation as the new solution of the current iteration. The idea is that by accepting a worse neighbour, occasionally the algorithm will "jump" to a new area in the search space, hence avoid getting stuck at the local maximum.

2.4.4 Genetic clustering algorithm

Genetic algorithms which have been successfully applied to many problems that involve exploring large search spaces can also be used for software clustering. GAs have been found to overcome some of the problems of traditional search methods such as hill-climbing; the most notable problem being "getting stuck" at local optimum, and therefore missing the global optimum (best solution) [Doval et al., 1999, Kazem, Ali Asghar Pourhaji And Lotfi, 2006, Mitchell, 1998].

Discovering an acceptable sub-optimal solution based on genetic algorithms involves starting with a population of randomly generated initial partitions and systematically improving them until all of the initial samples converge. In this approach, the resultant partition with the largest MQ is used as also a sub-optimal solution.

GAs operate on a set (population) of strings (individuals), where each string is an encoding of the problems input data. In the case of software clustering each individual item on the string represents the cluster association of entities. Each strings fitness (quality value) is calculated using MQ. In GA terminology, each iteration of the search is called a generation. In each

generation, a new population is created by keeping a certain portion of the fittest individuals of the previous generation and merging the rest to form new solutions. GAs use three basic operators to produce new generation of solutions from the existing ones:

Selection and Reproduction: Individuals are randomly selected from the current population, who are then combined to form the new population. However, the selection is not completely random and are biased to favour fittest members. Also, in some approaches the fittest member of the current population is always retained in the next generation, ensuring that results do not degrade over successive iterations.

Crossover: The crossover operator is used to combine the pairs of selected individuals (parents) to create new members, that potentially have a higher fitness than either of their parents. This is the core step of the algorithm that concentrates on improving the results at each iteration.

Mutation: The mutation operator is applied to every individual created from the crossover process. Mutation has a fixed probability and changes the members construction arbitrarily, thus avoiding getting stuck at local optimum.

The genetic algorithm used by Bunch [Doval et al., 1999] for partitioning of software systems is listed as Algorithm 3.

Algorithm 3: Genetic algorithm

Generate the initial population, creating random strings of fixed size;
repeat
  Create a new population by applying the selection and reproduction operator to select pairs of strings;
  Apply the crossover operator to the pairs of strings of the new population;
  Apply the mutation operator to each string in the new population;
  Replace the old population with the newly created population;
until *number of iterations is more than the maximum*;

GA algorithms generally iterate for a fixed number of times as the functions' upper bound (the maximum fitness value possible) is often not found.

The number of generations must be limited to guarantee termination of the search process. Mancoridis et al. [2001] developed a web-based portal (Reportal), which used genetic algorithms to perform architecture recovery for C, C++ and Java programs.

2.5 Other clustering approaches

This section briefly outlines several other approaches where software clustering is targeted at software architecture recovery.

2.5.1 Graph theoretic clustering

Graph theoretic partitioning can also be applied to graphs depicting direct relationships between entities just as in the previous section. Graph partitioning algorithms do not start from individual nodes (entities), but try to find sub graphs like connected components, maximal complete sub graphs or spanning trees to derive clusters [Xu and Wunsch, 2005]. However, this technique is not so common in the literature as graph partitioning also suffers from the same problem as optimal clustering algorithms.

Choi and Scacchi [1990] presented an approach to finding subsystem hierarchies based on graphs that represent resource exchanges between functions. They used graph-theoretic algorithms to remove articulation points so that strongly connected sub graphs are separated into clusters. Muller et al. [1993] presented an extension of this work. Lakhotia [1997] looked at four graph-theoretic based software clustering techniques and found that with exception of Choi and Sacchi's work [1990] the rest [Livadas and Johnson, 1994, Ong, 1994] produced flat results. Mancoridis et al. [1996] have also presented an approach where "Tube Graph" interconnection clustering is done to reduce the number of interconnection between the clusters detected by other techniques. This system complements other clustering techniques but on its own is not capable of doing clustering.

2.5.2 Knowledge based approaches

Another approach to the problem of understanding a software system and recovering its design is the knowledge-based approach. This technique involves reverse engineering smaller subsystems separately using domain knowledge. Finally, the understanding of the subsystems is combined to gain an overall understanding of the system. This approach has been shown to work well with

small systems, but fails to perform effectively on large systems [Tzerpos and Holt, 1998].

2.5.3 Pattern based approach

ACDC (Algorithm for Comprehension-Driven Clustering) [Tzerpos and Holt, 2000a] uses a pattern driven approach, where the common patterns (source file pattern, directory structure pattern, body header pattern, library pattern etc.) are located and are used for the clustering. The number and size of the clusters are also closely controlled to ensure that results are comprehensible and useful.

2.5.4 Formal methods based approaches

Formal methods have been also used for reverse engineering. Due to the mathematical nature of formal specification languages, formal methods are time consuming and tedious. When applied as a standalone technique the complexity of this approach was reported to be unmanageable [Gannod and Cheng, 1997]. Such techniques are cost effective only when studying safety critical systems.

2.5.5 Structure based approaches

The structure based approaches either use formal features or informal features of source code to perform clustering.

Formal features

A feature is considered to be formal if it consists of information that has direct impact on the software system's behaviour. For example, describing an entity with the "functions it calls" is a formal descriptive feature because it is an information source that has direct impact on the system's behaviour. Changing a function call in the code will result in a change in the system behaviour [Anquetil et al., 1999]. Lakhotia [1997] in his framework defines 21 such formal relationships, amongst which are: function assigns to global variable, function calls and file inclusion. The following techniques use formal features:

Concept analysis: Concept Analysis [van Deursen and Kuipers, 1999, Lindig and Snelting, 1997, Stiff and Reps, 1997] was also used for software clustering and has been shown to work well in certain scenarios, such as identification of objects.

Metric based: Belady and Evangelisti's [1981] approach groups related entities using a similarity metric based on data bindings. A data binding is a

potential data exchange via a global variable. Hutchens and Basili [1985] extend Belady and Evangelisti's work by using a hierarchical clustering technique to identify related entities and subsystems.

Schwanke presented his tool called ARCH [1991, 1993] where he pioneered the "classic" low-coupling and high-cohesion heuristics as a basis of software clustering. Also, his maverick analysis enabled refinement of a partition by identifying entities that were put in the wrong cluster to counter shortcoming of hierarchical algorithms.

Muller et al. [1988, 1993] introduced semi-automatic approaches in the form of the tool RIGI. They introduced the principles of small interfaces (the number of elements of a subsystem that interact with other subsystems should be small compared to the total number of elements in the subsystem) and of few interfaces (a given subsystem should interact only with a small number of the other subsystems).

Informal features

Many researchers have also used informal features to perform software clustering. Informal features are those features which do not have a direct influence on the system's behaviour. An example is the name of a function, changing the name of function has no impact on the system's behaviour. Informal features are also independent of programming languages and are regarded to provide better quality information than formal ones, as they are intended for human readers [Anquetil and Lethbridge, 1997]. Some commonly used informal features are file names, identifier names, function names, comments, physical organisation, developer, change request etc.

File name: Many organisations have well established rules for naming files and functions which means programmers name related files using meaningful and related suffixes or prefixes which can be easily extracted. A lot of work on clustering thus uses file names [Anquetil and Lethbridge, 1999, 1998]. Some researchers have also used several other heuristics based on naming conventions [Cimitile et al., 1997, Burd et al., 1996].

Developer: The organisation of system developers into teams can help reverse engineer a software's architecture as each member is likely to be dealing with one or related subsystems [Vanya et al., 2008].

Evolution history: Entities of the system that evolve together represent a relationship. This relationships can be mined from repositories and has also

been used as a basis of clustering [Beck, 2009, Hutchens and Basili, 1985].

Fault based approach: Similar to the previous approach, entities of the system that were modified to fix a bug/fault is also related and has been used as a basis of clustering [Selby and Basili, 1991].

2.5.6 Association rule mining

Maletic and Marcus applied information retrieval techniques called Latent Semantic Indexing (LSI) for architecture recovery [Maletic and Marcus, 2001, Kuhn et al., 2005]. This allowed them to extract various keywords from different entities and perform clustering based on similarity and frequency of keywords found.

2.5.7 Reflexion

Another approach in software clustering is the use of reflexion, where the developer creates a partitioning of the system that he thinks is correct. This is then compared to the partitioning extracted from the actual system using a tool and the difference between the tool's result and the programmer's views are compared to iteratively refine the clustering [Murphy et al., 1995].

2.5.8 Task driven approach

Tonella [2003] presents a task driven approach to software clustering where entities are clustered based on the support that they provide for understanding a set of modification tasks.

2.6 Clustering evaluation

This section of the review considers the techniques used to evaluate the clustering results (partitions) produced by various clustering techniques. The results can be evaluated qualitatively for suitability of use or can be measured quantitatively against set criteria and values.

2.6.1 Qualitative assessors

The primary purpose of software clustering is to aid in program comprehension, thus Anquetil and Lethbridge [1999] proposes that clustering results should exhibit the following high-level traits:

- *Actually represent the system.*
  The view achieved by the clustering should give a view of the current

state of the system rather than impose an ideal view based on domain understanding.

- *Make sense to the software designers.*
  The results should be presented in such a way that it is understandable and usable by designers and engineers.

- *Be adaptable to different needs.*
  The clustering process should provide various (hierarchical) views of the system, so that both the implementation details and overall design structure can be easily ascertained.

- *Be general.*
  The clustering process itself should not be tailored to fit one particular problem or language. It should be general so that it may be applied to different systems.

2.6.2 Quantitative assessors

Although the qualitative assessment is important in ensuring that the results obtained are useful, in most cases researchers want to assess their results against an expert decomposition of the system. By expert decomposition we refer to a decomposition that is carried out by the designer/developer who is also an expert on the system. Anquetil [1999] proposes to compare two partitions (results) by considering pairs of entities, where entities are in the same cluster (intra-pair). The quality is then measured in terms of Precision and Recall:

Precision: Percentage of intra-pairs proposed by the clustering method which are also intra-pairs in the expert partition.

Recall: Percentage of intra-pairs in the expert partition which were also found by the clustering method.

Although precision and recall are good indicators of clustering matches, they have issues where a bad result ranks high on one of the measures. A result partition containing only singleton clusters will have excellent precision and poor recall. On the other hand a partition with one huge cluster containing all the system entities will have excellent recall and poor precision.

To alleviate this problem of balancing the two measures, a single measure called MoJo [Tzerpos and Holt, 1999] was introduced. It counts the minimum

number of operations (such as moving an entity from one cluster to another and joining two clusters) needed to transform one partition to the other. A smaller MoJo value denotes higher similarity between the proposed clustering and expert decomposition whereas, a large value denotes the opposite.

Researchers have also proposed other metrics for comparing two partitions, for example, the Measure of Congruence [Lakhotia and Gravley, 1995], MoJoFM [Wen and Tzerpos, 2004], EdgeSim and MeCl [Mitchell and Mancoridis, 2001a], and the Koschke-Eisenbarth (KE) measure [Koschke and Eisenbarth, 2000]. All the six measures mentioned above compare flat software decompositions, where all subsystems are presented at a single level. Shtern and Tzerpos [2004] present a framework for comparing hierarchical decompositions.

2.6.3 External assessment

Clustering results are validated using three different approaches: external, internal, and relative assessment [Jain et al., 1999, Maqbool and Babri, 2007]. The external assessment involves comparing the results obtained from the clustering technique to an expert or external decomposition.

Expert decomposition

Expert decomposition, also known as the gold standard, is obtained by a manual inspection of the system by the designer/architect or an expert. According to the literature review, this technique is by far the most common evaluation technique applied by researchers [Tzerpos and Holt, 2000a]. However, Maqbool [2007] notes in his work that experts often do not agree on a decomposition. They have different points of view from which they look at the system resulting in different outcomes. It is thus noted in the literature that "a classification is neither true or false" [Everitt, 1974] and there is no single "best" approach [Anderberg, 1973] to clustering.

Physical structure

Obtaining expert decomposition is not always possible. An alternative is the use of the physical/directory structure of the system [Anquetil et al., 1999]. The structure of well-known open systems (such as Linux) are organised in several directories which form a reasonable decomposition of the systems and may be used in the place of an actual expert decomposition. However, this technique should be employed with caution as it relies on the fact that developers would responsibly organise the system physically.

2.6.4 Internal assessment

The internal assessment is where instead of comparing the results to a decomposition not produced by the system, properties and characteristics of the system are used to validate the results.

Arbitrary decisions

During the clustering process, when computing which two clusters to combine, there are instances where multiple clusters have the same similarity value, forcing the algorithm to pick one of the pairs arbitrarily. Arbitrary decisions may turn out to be problematic [Maqbool and Babri, 2004], especially in the case of hierarchical clustering algorithms, where once a decision is made it cannot be reversed. Maqbool [2007] in his study used this as a criteria for deciding upon the quality of the clustering.

Redundancy

Anquetila et al. [1999] argue that formal features often provide redundant information. Considering file inclusion will also automatically consider variable and function inclusion. Thus the result of a clustering process can be assessed using the number of redundant features used.

Number and size of clusters

The number of clusters and cluster size have also been used for evaluating clusters internally [Anquetil and Lethbridge, 1999, Davey and Burd, 2000, Wu et al., 2005]. The number of clusters obtained at each step of the clustering process can be used as an indicator of the quality of a clustering approach [Davey and Burd, 2000]. Similarly, the number of singleton clusters can also be an indicator of the quality. Thus, algorithms that tend to start attracting all the entities into one cluster, those that tend to have a lot of singleton clusters or ones where all clusters suddenly seem to merge into one large cluster are all indicators of poor quality.

Design criterion

Anquetil and Lethbridge [1999] use the measure of cohesion and coupling between subsystems within a partition to evaluate its quality. Well-designed clusters are more likely to be of interest to the software engineers which should also exhibit high cohesion and low coupling [Sommerville, 1995].

2.6.5 Relative Assessment

Relative assessment is used for comparing results of two different clustering algorithms or results of the same algorithm.

Stability

An algorithm is said to be stable if the clustering results that it produces do not change drastically when small changes are made to the system. This approach has been widely used [Tzerpos and Holt, 2000b,a, Wu et al., 2005, Hutchens and Basili, 1985] to assess the quality of clustering approaches.

CRAFT [Mitchell and Mancoridis, 2001b] is a tool that supports relative evaluation of clustering results. When supplied with a clustering algorithm and a system, the tool compares the result of the algorithm with results produced by a common set of algorithms already built into the tool. This tool can be the starting point of assessing quality of results produced by new techniques against already existing ones.

2.7 Other issues

The literature also identifies a few issues which needs to be considered when proposing new clustering approaches.

2.7.1 Cluster labelling

Providing meaningful names to subsystems detected during clustering will allow maintainers to understand the results faster. One of the earliest works on cluster labelling is by Schwanke and Platoff [1993], who use a summary of features to suggest labels, which are then assigned manually. Ricca et al. [2004] and Kuhn et al. [2005] perform clustering and cluster labelling on the basis of keywords. Maqbool and Babri [2006] use function identifiers as representative keywords for labelling entities.

2.7.2 Omni-present entities

In software systems there are entities, such as utility functions, which act as suppliers to other entities of the system. It is often the case that these entities are frequently called by others and have have far more inter-relationships than most other entities. The results of attempting to group such entities are thus unpredictable [Lung et al., 2004]. Most of the work in the literature detect omnipresent entities based on their high number of interconnections [Mancoridis et al., 1999, Hutchens and Basili, 1985, Mitchell and Mancoridis, 2006, Muller and Uhl, 1990, Wen and Tzerpos, 2005] and deal with them separately.

2.8 Dependence-based Clustering

The chapter has thus far discussed various clustering techniques based on properties shared by entities or dependency relationships between the entities such as file or variable access. This section discusses some of the recent work on software clustering which uses data and control flow information in programs to ascertain dependencies and uses that for clustering.

The notion of using data and control dependence to perform software clustering was first introduced by Binkley and Harman [2005b]. They defined a dependence cluster, which is a set of mutually dependent program elements. They looked at 20 subject programs and found that a significant number of programs have large dependence clusters. They also introduced approximations where static program slicing could be used efficiently to locate dependence clusters in programs. Harman et al. [2009] later extended this initial study with a large-scale empirical study of 45 programs. They again found slice-based dependence clusters to be common in the programs and defined separate clustering techniques depending on the dependency flow direction considered.

Jiang et al. [2008] proposed the use of search-based program slicing to identify dependence structures in programs using evolutionary algorithms (Hill Climbing and Genetic). Rather then using traditional program slicing to capture dependence clusters, Static Execute After/Before (SEA/SEB) was proposed as an alternative [Jasz et al., 2008]. SEA-based clustering is more efficient as it considers functions of the program as entities and clusters based on execution traces, however this efficiency comes at the price of precision.

Although dependence clusters were found to be quiet common in programs, they were not well-understood because of their intricate interweaving of dependencies and subtleties of dependency in programs. As such, they were regarded as problems that may make it difficult for developers and maintainers in understanding programs. Black et al. [2006] have indicated that there may even be the potential for a link between program faults and dependence clusters. To this end, Binkley et al. [2008b] regarded dependence clusters as anti-patterns that developers should be aware of. Binkley et al. [2005b] identified that global variables as one of the causes for the formation of dependence clusters and found that over half of the programs have a global variable that was causing a significant cluster. This has also motivated other work [Binkley, 2008, Binkley and Harman, 2009] where low-level causes of dependence clusters

are identified and their removal is attempted. Recent extension to this idea has proposed an improved algorithm [Binkley et al., 2013a] for identifying linchpin vertices (vertices that are responsible for dependence clusters). The search for linchpin vertices that cause dependence clusters have also been extended for the SEA-based clusters, where linchpins are functions of the program [Schrettner et al., 2012, Jasz et al., 2012].

Lehnert [2011] has considered the relationship between dependence clusters and impact analysis and found that clustering can be used to determine the ripple-effect during software maintenance. Beszédes [2007] looks at the relationship between SEA-based clusters and software maintenance and found that SEA can be used to identify hidden dependencies helping in many maintenance tasks, including change propagation and regression testing.

2.9 Our Clustering Technique

The work in this thesis will use dependence-based clustering. The use of dependence-based clustering is expected to yield clusters consisting of program elements that share dependency properties. Similar to previous work by Binkley and Harman [2005b] and Harman et al. [2009], we will be performing our clustering on the vertices of the System Dependence Graph. As such, the clustering process will treat the vertices of the SDG as entities. This not only ensures that dependence relationship between various program points are correctly identified but will also alleviate issues of source code layout and various programming styles.

Vertices of the SDG that have the exact same relationship will be grouped together in the same cluster. The dependence relationships will be extracted in the form of program slice, vertices that yield the same slice will be grouped together into the same cluster. We therefore use the sibling link approach where we consider the similarity of the slices produced by the entities. For the similarity measure we rely on a binary measure of whether vertices yield the exact same slice or not.

Finally, we use a variation of the expert decomposition technique for evaluation, where we study whether the clusters represent the high-level logical structure of the programs as done by similar work [Beszedes et al., 2013, Hamilton and Danicic, 2012] in the area.

2.10 Chapter Summary

This chapter gives a literature survey on software clustering and the various techniques used in such clustering. The literature survey shows that there is no single best approach to clustering. Different clustering algorithms may provide different results or views of the same system. As architecture is not explicitly represented at the source code level, clustering infers it from the entire system based on some criteria. Different researchers propose different criteria and often developers working on the same system cannot fully agree on how a system should be decomposed.

In our clustering approach we will be using vertices of the system dependence graph as entities and cluster them based on dependencies that they have in common. The full details of the clustering is presented in Chapters 3 and 4.

Chapter 3

Dependence Clusters

3.1 Overview

This chapter provides detailed background information on dependence clusters and its various instantiations which is later extended to define coherent dependence clusters (in Chapter 4). The chapter also provides the necessary background on program slicing, dependence analysis and existing dependence cluster visualisation techniques.

Previous work [Binkley and Harman, 2005b, Harman et al., 2009] has used the term *dependence cluster* for a particular kind of cluster, termed a *mutually-dependent cluster* herein to emphasise that such clusters consider *only* mutual dependence internal to the cluster. This later allows the definition to be extended to incorporate external dependencies.

3.2 Mutually-dependent clusters

Informally, *mutually-dependent clusters* are maximal sets of program statements that mutually depend upon one another. Harman et al. [2009] present the following formalisation of mutually dependent sets and clusters.

Definition 1 (Mutually-Dependent Set and Cluster)
A *mutually-dependent set (MDS)* is a set of statements, S, such that

$\forall x, y \in S : x$ depends on y.

A *mutually-dependent cluster* is a maximal MDS; thus, it is an MDS not properly contained within another MDS.

The definition of an MDS is parametrised by an underlying *depends-on* relation. Ideally, such a relation would precisely capture the impact, influence, and dependence between statements. Unfortunately, such a relation is

not computable. A well known approximation is based on Weiser's *program slice* [Weiser, 1981]: a slice is the set of program statements that affect the values computed at a particular statement of interest (referred to as a slicing criterion). While its computation is undecidable, a minimal (or precise) slice includes exactly those program elements that affect the criterion and thus can be used to define an MDS in which statement t depends on statement s iff s is in the minimal slice taken with respect to slicing criterion t.

The slice-based definition is useful because algorithms to compute (approximations to minimal) slices can be used to define and compute approximations to mutually-dependent clusters. A slice can be computed as the solution to a reachability problem over a program's *System Dependence Graph* (SDG) [Horwitz et al., 1990]. The following subsections will give the necessary background information on SDGs and program slicing before continuing with the various definitions that instantiate mutually-dependent clusters.

3.3 System Dependence Graphs

An SDG [Horwitz et al., 1990] is an inter-connected collection of Procedure Dependence Graphs (PDGs) [Ferrante et al., 1987]. A PDG is a directed graph comprised of vertices, which essentially represent the statements of the program, and two kinds of edges: data dependence edges and control dependence edges.

A data dependence edge is essentially a data flow edge that connects a definition of a variable with each use of the variable reached by the definition [Ferrante et al., 1987]. For example, in Figure 3.1, there is a data dependence between the point i=1 and the point while (i < 11) indicating that the value of i flows between those two points.

A control dependence connects a predicate p to a vertex v when p has at least two control-flow-graph successors, one of which can lead to the exit vertex without encountering v and the other always leads eventually to v [Ball and Horwitz, 1993]. That is, v must not post-dominate p and there must exist a vertex n in the path from p to v post-dominated by v (Ferrante et al. [1987]). Thus p controls the possible future execution of v. For example, in Figure 3.1, there is a control-dependence edge between the vertex representing the statement while (i < 11) and the statements sum = sum + i and i = i + 1, whose execution depends on whether the former evaluates to true. For structured

code, control dependence reflects the nesting structure of the program.

```
1:    int main() {
2:      int sum = 0;
3:      int i = 1;
4:      while (i < 11) {
5:        sum = sum + i;
6:        i = i + 1;
7:      }
8:      printf("%d\n", sum);
9:      printf("%d\n", i);
10:   }
```

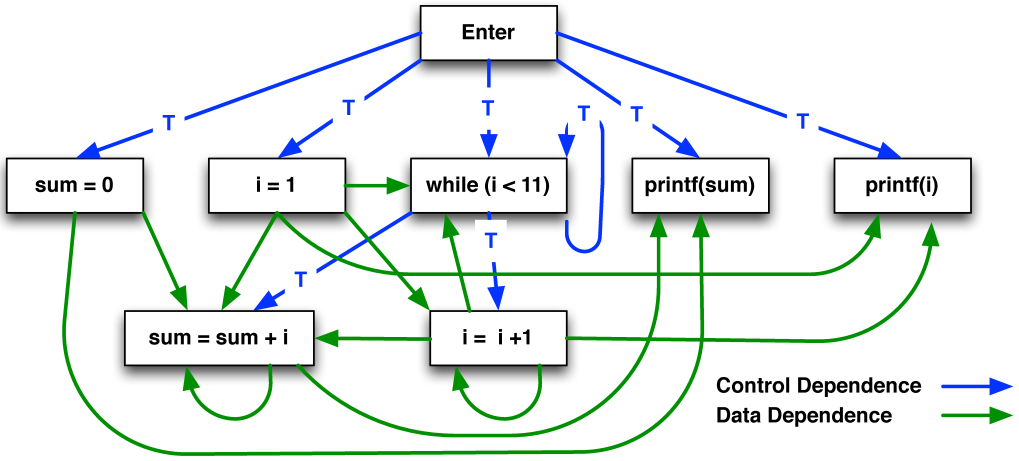


Figure 3.1: Program Dependence Graph (PDG)

An SDG is a directed graph consisting of interconnected PDGs, one per function in the program. Interprocedural control-dependence (dashed blue lines in Figure 3.2) edges connect procedure call sites to the entry points of the called procedure. Interprocedural data-dependence edges (broken green lines in Figure 3.2) represent the flow of data between actual parameters and formal parameters (and return values).

In an SDG, in addition to the vertices representing statements and predicates, each PDG explicitly contains entry and exit vertices, vertices representing parameters and return values. A PDG contains *formal-in* vertices representing the parameters to the function and *formal-out* vertices representing the variables returned by the function. A function call is represented by a *call-site* vertex and there is an interprocedural control-dependence edge from

```
1:   int main() {
2:     int sum = 0;
3:     int i = 1;
4:     while (i < 11) {
5:       sum = add(sum, i);
6:       i = add(i, 1);
7:     }
8:     printf("%d\n", sum);
9:     printf("%d\n", i);
10:  }
```

```
11:  int add (int x, int y) {
12:    x = x + y;
13:    return x;
14:  }
```

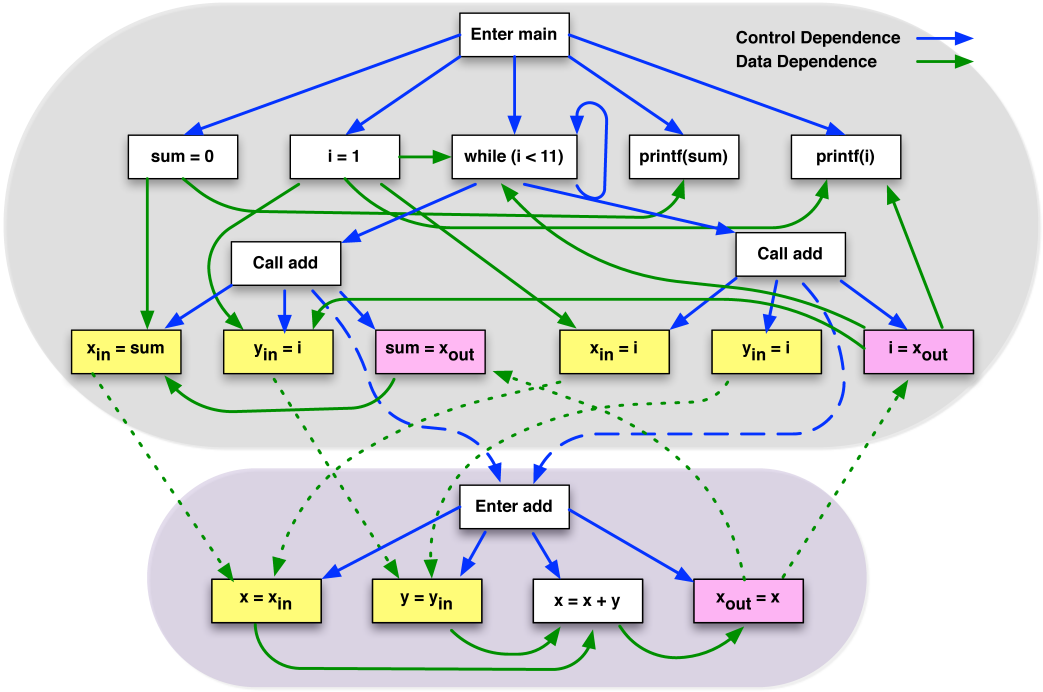


Figure 3.2: System Dependence Graph (SDG)

each call site to the corresponding entry point of the callee. There is also a control-dependence edge from a procedure's entry vertex to each of the top-level statements in that procedure, as any of the top-level statements are only reachable by the execution of the function. Finally there is also an inter-procedural data-dependence edge between the *actual-in* parameters associated with a *call-site* and the *formal-in* parameters of a procedure. There is an inter-procedural data-dependence edge between PDGs *formal-out* vertices and the associated *actual-out* vertices in the calling procedure.

Non-local variables such as globals, file statics, and variables accessed indirectly through pointers are handled by modelling the program as if it used only local variables. Each non-local variable used in a function, either directly or indirectly, is treated as a "hidden" input parameter, and thus gives rise to additional program points. For each global used or modified by a procedure there is a *global-formal-in* vertex and for each variable modified by a procedure there is a *global-formal-out* vertex. There is an inter-procedural data-dependence edge between the *global-actual-in* parameters associated with a call-site and the *global-formal-in* parameters of a procedure. There is an inter-procedural data-dependence edge between PDGs *global-formal-out* vertices and the associated *global-actual-out* vertices in the calling procedure. The PDG/SDG-based representation subsumes the notion of call graphs and data flow graphs [Anderson and Teitelbaum, 2001].

3.4 Program Slicing

Since its advent, program slicing has come a long way and has been extended in many directions and specialised to individual programming languages and language independent ones [Binkley et al., 2013b]. The huge number of program slicing related publications have resulted in several survey papers [Harman and Hierons, 2001, Xu et al., 2005, Tip, 1995, Binkley and Harman, 2004]. The various slicing techniques are summarised in Figure 3.3 by a recent survey on program slicing by Silva [2012]. It is beyond the scope of this thesis to provide a comprehensive survey of the slicing techniques and we only include background that is relevant to this thesis.

Program slicing is a technique which computes a set of program statements, known as a slice, that may affect a point of interest known as the slicing criterion. In other words program slicing reveals all the program points that the slice *criterion* depends on. Mark Weiser [1979] first introduced program slicing in 1979. A slice S is a set of statements calculated for program P with respect to a *slicing criterion* represented as a pair (V, s), where V is a set of variables and s is a program point of interest. The slice S of P is obtained by removing all statements and predicates of P which cannot affect V at s.

A program can be traversed forwards or backwards from the slicing criterion. As defined by Weiser when traversed backwards, all those statements that could influence the slicing criterion are found and is hence termed back-

Figure 3.3: Slicing Survey [Silva, 2012]

ward slicing. In contrast, when traversed forwards, all those statements that could be influenced by the slicing criterion can be found and is referred to as forward slicing [Reps and Bricker, 1989].

The original definition of program slicing defined by Weiser was also static [Weiser, 1984] in the sense that it did not consider any particular input for the program being sliced. The slice for a particular criterion did not consider any particular execution, and was created for all possible inputs. Korel and Laski [1988] introduced the concept of dynamic slicing where a slice is computed using information from a trace of an execution of the program for a specific set of inputs. In general, dynamic slices are much smaller (precise) than static ones because they contain the statements of the program that affect the slice criterion for a particular execution (in contrast to any possible execution in case of static slicing). More recently, Binkley et al. [2013b] have introduced observation-based slicing which uses program execution and observation of the output to perform program slicing for multi-language systems.

In order to extract a slice from a program, the dependencies between its statements must be computed first. Ottenstein and Ottenstein [1984] noted that the program dependence graph (PDG) was the ideal data structure for

program slicing because it allows for slicing to be done in linear time on the number of vertices of the PDG. An intraprocedural slice can be computed as a simple graph reachability problem. For example in Figure 3.4, slicing on the statement printf(i) produces a slice that includes all the nodes of the graph that are reachable by traversing the directed edges. Edges and vertices of the PDG included in the slice are shown in bold.

```
1:    int main() {
2:      int sum = 0;
3:      int i = 1;
4:      while (i < 11 ){
5:        sum = sum + i;
6:        i = i + 1;
7:      }
8:      printf("%d\n", sum);
9:      printf("%d\n", i);
10:   }
```

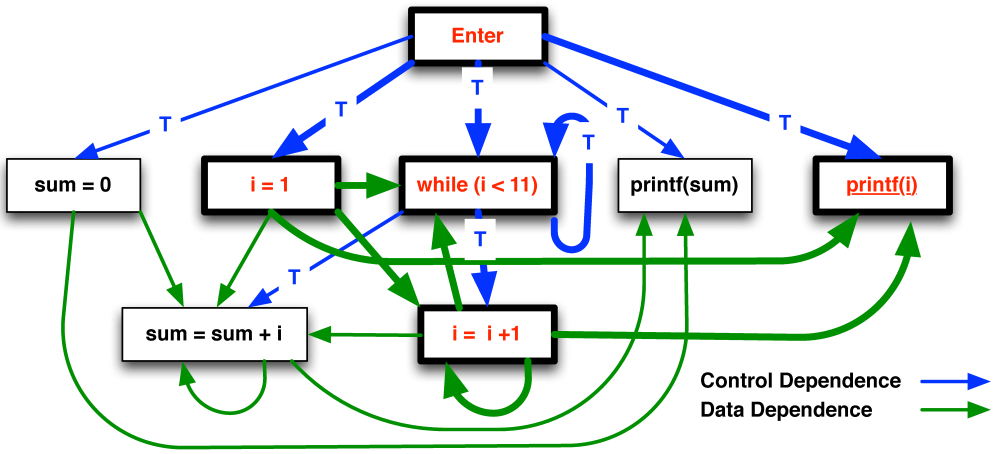


Figure 3.4: Intraprocedural program slicing

The original definition of program slicing has been later classified as intraprocedural slicing, because the original algorithm does not take into account information related to the fact that slices can cross the boundaries of procedure calls. In such cases, simple traversal of the graph leads to an imprecise slice with edges traversed that are not feasible in the control flow of the program, failing to respect *calling context* [Gallagher, 2004, Binkley and Harman, 2005b, Horwitz et al., 1990, Krinke, 2002, Binkley and Harman, 2003, Krinke, 2003].

For example, in Figure 3.2 backward (transitive) traversal of all directed edges from the statement printf(i) would include both calls to add. However, it is clear that the call to add from line 5: sum = add(sum,i); cannot influence printf(i) and should not be included in the slice. If dependence were transitive and slicing could be done as a transitive closure, both calls to add would be included in the slice.

Horwitz et al. [1988] proposed to address this problem by introducing calling context during the graph traversal. When slicing an SDG, a slicing criterion is a vertex from the SDG. Horwitz et al. [1990] introduced an algorithm that makes two passes over the system dependence graph to compute a valid slice. Each pass traverses only certain kinds of edges. To calculate a backward slice taken with respect to SDG vertex v, denoted BSlice(v), the traversal in Pass 1 starts from v and goes backwards (from target to source) along flow edges, control edges, call edges, summary edges, and parameter-in edges, but not along parameter-out edges. The traversal in Pass 2 starts from all actual-out vertices reached in Pass 1 and goes backwards along flow edges, control edges, summary edges, and parameter-out edges, but *not* along call or parameter-in edges. The result is an inter-procedural backward slice consisting of the set of vertices encountered during Pass 1 and Pass 2, and the edges induced by those vertices. Symmetrically, for a forward slice with respect to SDG vertex v, denoted FSlice(v), the traversal in Pass 1 starts from v and follows only edges *up* into calling procedures and Pass 2 traverses edges *down* into called procedures.

The valid backward slice for statement printf(i) is shown in Figure 3.5 which respects the calling context. Here the backward slice on printf(i) no longer includes the additional call to function add.

We employ both kinds of context-sensitive static SDG slices in this thesis, backward slice and forward slice.

Definition 2 (Backward Slice)
The backward slice taken with respect to vertex v of an SDG, denoted BSlice(v), is the set of vertices reaching v via a path of control and data dependence edges of the SDG, where the path respects context.

Definition 3 (Forward Slice)
The forward slice, taken with respect to vertex v of an SDG, denoted FSlice(v), is the set of vertices reachable from v via a path of control and data dependence edges of the SDG, where the path respects context.

```
1:   int main() {
2:     int sum = 0;
3:     int i = 1;
4:     while (i < 11) {
5:       sum = add(sum, i);
6:       i = add(i, 1);
7:     }
8:     printf("%d\n", sum);
9:     printf("%d\n", i);
10:  }
```

```
11:  int add (int x, int y) {
12:    x= x + y;
13:    return x;
14:  }
```

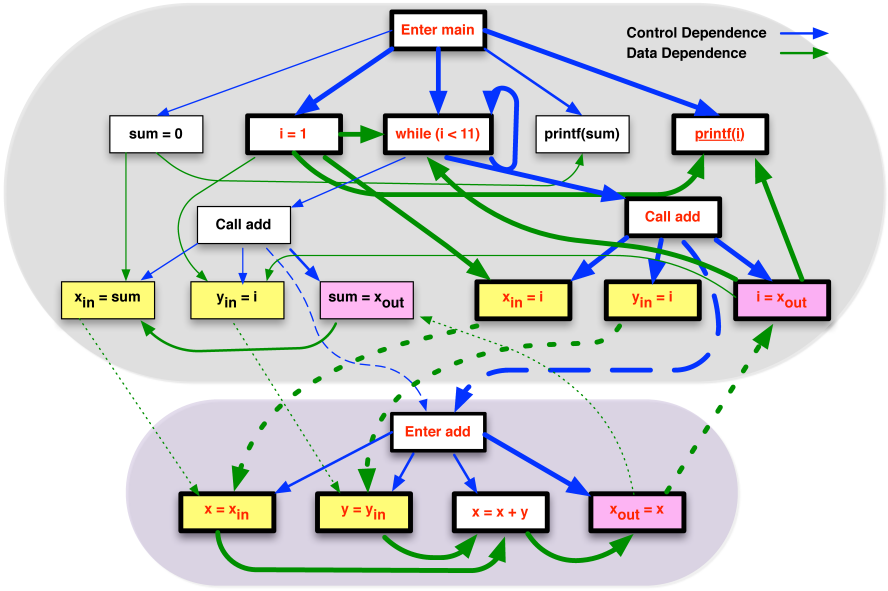


Figure 3.5: Interprocedural program slicing

3.5 Slice-based Clusters

A *slice-based cluster* is a maximal set of vertices included in each others slice. The following definition essentially instantiates Definition 1 using BSlice. Because $x \in \mathsf{BSlice}(y) \Leftrightarrow y \in \mathsf{FSlice}(x)$ the dual of this definition using FSlice is equivalent. Where such a duality does not hold, both definitions are given. When it is important to differentiate between the two, the terms *backward* and *forward* will be added to the definition's name as is done in this chapter. Harman et al. [2009] provide the following definition:

Definition 4 (Backward-Slice MDS and Cluster)
A *backward-slice MDS* is a set of SDG vertices, V, such that

$\forall x, y \in V : x \in \mathsf{BSlice}(y)$.

A *backward-slice cluster* is a backward-slice MDS contained within no other backward-slice MDS.

Note that as x and y are interchangeable, this is equivalent to $\forall x, y \in V : x \in \mathsf{BSlice}(y) \wedge y \in \mathsf{BSlice}(x)$. Thus, any unordered pair (x, y) with $x \in \mathsf{BSlice}(y) \wedge y \in \mathsf{BSlice}(x)$ creates an edge (x, y) in an undirected graph in which a complete subgraph is equivalent to a backward-slice MDS and a backward-slice cluster is equivalent to a maximal clique. Therefore, the clustering problem is the NP-Hard *maximal cliques* problem [Bomze et al., 1999] making Definition 4 prohibitively expensive to implement. An efficient and practical approximation of slice-based clusters are discussed in subsequent sections of this chapter.

3.6 Identifying causes of Dependence Clusters

There have been multiple studies into the causes of dependence clusters. The first of these studies by Binkley et al. [2010] explicitly looked at the impact of global variables on dependence clusters. Other studies include identification of linchpin vertices [Binkley and Harman, 2009], which are responsible for holding clusters together. Dependence clusters essentially group together program statements that have mutual dependency, as such there are three primary constructs that are responsible for formation of dependence clusters.

3.6.1 Loops

A loop is a sequence of instruction(s) that is continually repeated until a certain condition is reached. Loop carried dependence can lead to mutual dependence between the statements forming the body of the loop. An example of such a cluster is seen in Figure 3.6. The statement on line 5 and the two predicates all depend upon each other as the data dependence on i is carried forward during iterations of the loop. As loops have inherent dependence within, the body of the loop and consequently the termination condition become mutually dependent on each other, leading to formation of dependence clusters.

Although one could assume that a program whose execution is controlled via an infinite loop would have all of its statements in a large dependence cluster, it would be wrong. Because interprocedural dependence is not transitive (discussed in Section 3.7) loops on its own will rarely lead to large dependence clusters in real-world production code. However, intraprocedural dependence is transitive and the formation of an intraprocedural dependence cluster *always*

P backward slice on i at line 5		P
	1:	
	2:	...
|	3:	while (i < 10)
|	4:	if (a[i]) > 0)
|	5:	**i** = i + 2;
	6:	...

Figure 3.6: Depenence Cluster caused by loop

requires a loop construct, albeit not all statements of a loop need to be part of the same dependence cluster.

3.6.2 Global Variables

The use of global variables is another cause of dependence cluster formation. When various components of programs interact with each other using globals, in other words when several components read and write to a global, it is possible for the components to become mutually dependent with the variable and in-turn mutually dependent upon each other. An example of this is seen in Figure 3.7 where both functions f1 and f2 read and write to the global x. This causes each of the functions bodies to have mutual dependence with the global x and subsequently on each other.

Global variables and pointers that refer to globals have a scope that covers the entire code. The definitions and use of global variables can 'glue together' statements to form a large cluster. This is even more evident in places where the global variables link various smaller clusters together into one very large cluster of mutual dependence. For example, where a global variable is responsible for mutual data flow between two large and otherwise unconnected clusters. The variable acts as a small 'capillary vessel' along which the dependence 'flows' linking two unconnected sub-clusters to create one larger cluster. A study by Binkley et al. [2010] found that a quarter of programs have a global variable that is solely responsible for large dependence clusters.

3.6.3 Mutually Recursive Calls

Mutual recursion of function calls can also lead to formation of dependence clusters because each function (transitively) calls all the others, making the outcome of each function dependent upon the outcome of some call to the

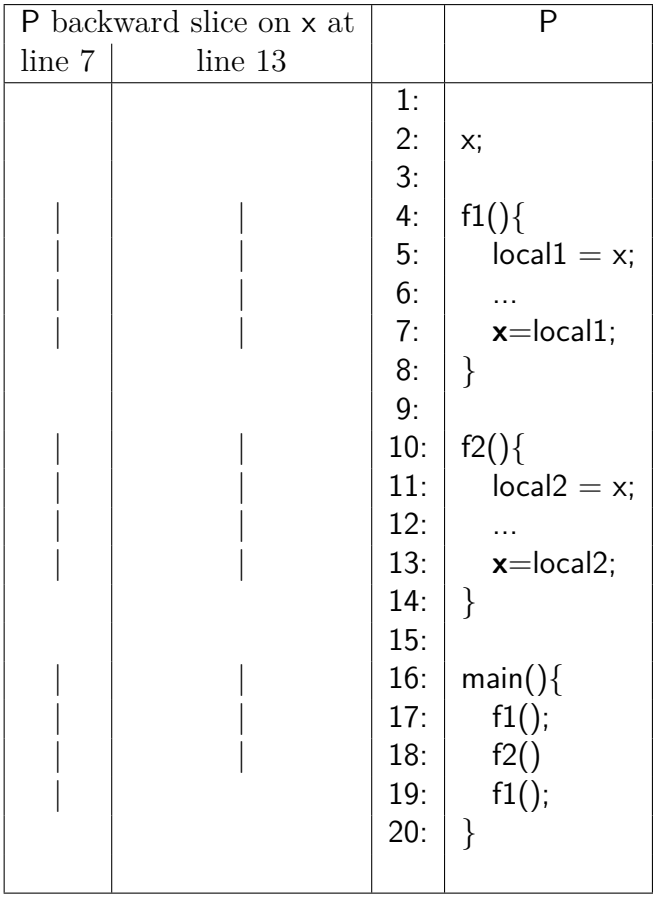


Figure 3.7: Dependence Cluster caused by global variable

others. Figure 3.8 gives an example of a cluster formed due to mutual recursion. The functions even and odd call each other recursively causing mutual interdependence and a slice-based cluster.

The clusters projected due to mutually recursive calls are sometimes aggravated and deemed to be larger because of the nature of safe (conservative) approximation employed by static analysis (discussed further in section 6.2.4).

3.7 Dependence Intransitivity

A naïve definition of a dependence cluster would be based on transitive closure of the dependence relation and thus would define a cluster to be a strongly connected component in the SDG. Unfortunately, for certain language features, dependence is not transitive. Examples of such features include procedures [Horwitz et al., 1990] and threads [Krinke, 1998]. Thus, in the presence of these features, strongly connected components overstate the size and number

P backward slice on r at			P
line 8	line 16		
		1:	
|	|	2:	even (i){
|	|	4:	if (i == 0)
|	|	5:	r =1 ;
|	|	6:	else
|	|	7:	r = odd(abs(i)-1);
|	|	8:	return r;
		9:	}
		10:	
|	|	11:	odd (i){
|	|	12:	if (i == 0)
|	|	13:	r =0 ;
|	|	14:	else
|	|	15:	r = even(abs(i)-1);
|	|	16:	return r;
		17:	}
		18:	

Figure 3.8: Dependence Cluster caused by mutual recursion

of dependence clusters. Fortunately, context-sensitive slicing captures the necessary context information [Binkley and Harman, 2005b, Horwitz et al., 1990, Krinke, 2002, Binkley and Harman, 2003, Krinke, 2003].

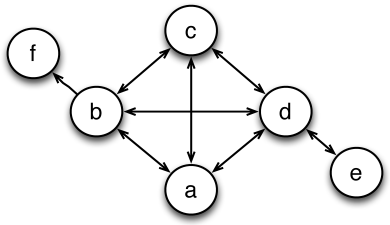
The program P shown in Figure 3.9 illustrates the non-transitivity of slice inclusion. The program has six statements (a, b, c, d, e and f) whose dependencies are shown in columns 1–6 using backward slice inclusion. The dependency relationship between these variables are also extracted and shown in Figure 3.10 using a directed graph where the nodes of the graph represent the statements and the edges represent the backward slice inclusion relationship in Figure 3.9. In the diagram, a depends on b ($b \in BSlice(a)$) is represented by $b \rightarrow a$. The diagram firstly shows two instances of dependence intransitivity in P. Although b depends on nodes a,c and d, node f that depends on b does not depend on a, c or d. Similarly node d depends on node e but nodes a, b and c that depend on d do not depend on node e.

Because dependence is not transitive, calculating slice-based clusters is equivalent to the maximal clique problem which is NP-Hard [Bomze et al., 1999]. Other than the high cost of calculating maximal cliques, the problem is further compounded by the fact that even when maximal cliques can be

backward slice on assignment to						P
a	b	c	d	e	f	
						1:
|	|	|	|			2: f1(x) {
|	|	|	|			3: **a** = f2(x, 1) + f3(x);
|	|	|	|			4: return f2(a, 2) + f4(a);
						5: }
						6:
|	|	|	|		|	7: f2(x, y) {
|	|	|	|		|	8: **b** = x + y;
|	|	|	|		|	9: return b;
						10: }
						11:
|	|	|	|			12: f3(x) {
|	|	|	|			13: if (x>0) {
|	|	|	|			14: **c** = f2(x, 3) + f1(x);
|	|	|	|			15: return c;
						16: }
|	|	|	|			17: return 0;
						18: }
						19:
|	|	|	|	|		20: f4(x) {
|	|	|	|	|		21: **d** = x;
|	|	|	|	|		22: return d;
						23: }
						24:
			|	|		25: f5(x) {
			|	|		26: **e** = f4(5);
			|			27: return f4(e);
						28: }
						29:
	|				|	30: f6(x){
						|	31: **f** = f2(42, 4);
						32: return f;
						33: }
						34:

Figure 3.9: Dependence intransitivity and clusters

calculated at higher costs it may lead to undecideable situations where dependencies have to be arbitrarily ignored. For example in Figure 3.11 we see an example where vertices i, j, k are mutually dependent and vertices i, j, l are



Slice Criterion	Backward Slice	Forward Slice
a	{a, b, c, d}	{a, b, c, d}
b	{a, b, c, d}	{a, b, c, d, f}
c	{a, b, c, d}	{a, b, c, d}
d	{a, b, c, d, e}	{a, b, c, d, e}
e	{d, e}	{d, e}
f	{b, f}	{f}

Figure 3.10: Backward slice inclusion relationship for Figure 3.9

also mutually dependent. It should be noted that vertices k and l do not have any dependencies between them because of dependence intransitivity. In such a case it is not clear whether a dependence cluster (or maximal) clique should be formed from {i, j, k} leaving {l} in its own cluster or forming {i, j, l} into a cluster leaving {k} in its own cluster. To overcome this partitioning problem, an approximation of the same-slice clusters were introduced by Binkley and Harman [2005b].

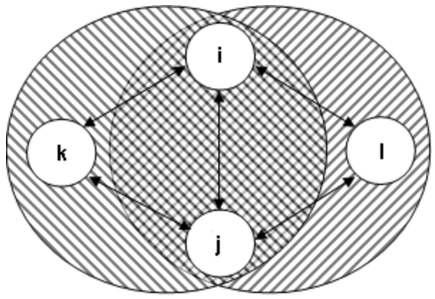


Figure 3.11: Overlapping dependence clusters

3.8 Same-Slice Clusters

An alternative definition of slice-based clusters uses the *same-slice* relation in place of slice inclusion [Binkley and Harman, 2005b]. This relation replaces the need to check if two vertices are in each others slice with checking if two vertices have the *same* slice. The result is captured in the following definitions for *same-slice cluster* [Harman et al., 2009]. The first uses backward slices and the second uses forward slices.

Definition 5 (Same-Slice MDS and Cluster)
A *same-backward-slice MDS* is a set of SDG vertices, V, such that

$\forall x, y \in V : \mathsf{BSlice}(x) = \mathsf{BSlice}(y)$.

A *same-backward-slice cluster* is a same-backward-slice MDS contained within no other same-backward-slice MDS.

A *same-forward-slice MDS* is a set of SDG vertices, V, such that
$\forall x, y \in V : \mathsf{FSlice}(x) = \mathsf{FSlice}(y)$.
A *same-forward-slice cluster* is a same-forward-slice MDS contained within no other same-forward-slice MDS.

Because $x \in \mathsf{BSlice}(x)$ and $x \in \mathsf{FSlice}(x)$, two vertices that have the same slice will always be in each other's slice. If slice inclusion were transitive, a backward-slice MDS (Definition 4) would be identical to a same-backward-slice MDS (Definition 5). However, as illustrated by the example in Figure 3.9, slice inclusion is not transitive; thus, the relation is one of containment where every same-backward-slice MDS is also a backward-slice MDS but not necessarily a maximal one.

For example, in Figure 3.10 the set of vertices {a, b, c} form a same-backward-slice cluster because each vertex of the set yields the same backward slice. Whereas the set of vertices {a, c} form a same-forward-slice cluster as they have the same forward slice. Although vertex d is mutually dependent with all vertices of either set, it doesn't form a same-slice cluster with either set because it has additional dependence relationship with vertex e.

Although the introduction of same-slice clusters was motivated by the need for efficiency, the definition inadvertently introduced an *external* requirement on the cluster. Comparing the definitions for slice-based clusters (Definition 4) and same-slice clusters (Definition 5), a slice-based cluster includes only the *internal* requirement that the vertices of a cluster depend upon one another. However, a same-backward-slice cluster (inadvertently) adds to this internal requirement the *external* requirement that all vertices in the cluster are affected by the same vertices external to the cluster. Symmetrically, a same-forward-slice cluster adds the *external* requirement that all vertices in the cluster affect the same vertices external to the cluster.

3.9 Same-Slice-Size Cluster

Even calculating same-slice clusters is expensive. In practice it requires tens of gigabytes of memory for even modest sized programs. Thus, a second approximation was also employed by Binkley and Harman [2009]. This approximation

replaces 'same-slice' with 'same-slice-size': rather than checking if two vertices yield identical slices, the approach simply checks if the two vertices yield slices of the same size. The resulting *same-slice-size* approach is formalised by Harman et al. [2009] as follows:

Definition 6 (Same-Slice-Size Backward MDS/Cluster)
A *Same-Slice-Size Backward MDS* is a set of statements, S, such that

$$\forall x, y \in S : |\mathsf{BSlice}(x)| = |\mathsf{BSlice}(y)|.$$

A *Same-Slice-Size Backward Cluster* is a Same-Slice-Size Backward MDS contained within no other Same-Slice-Size Backward MDS.

Definition 7 (Same-Slice-Size Forward MDS/Cluster)
A *Same-Slice-Size Forward MDS* is a set of statements, S, such that

$$\forall x, y \in S : |\mathsf{FSlice}(x)| = |\mathsf{FSlice}(y)|.$$

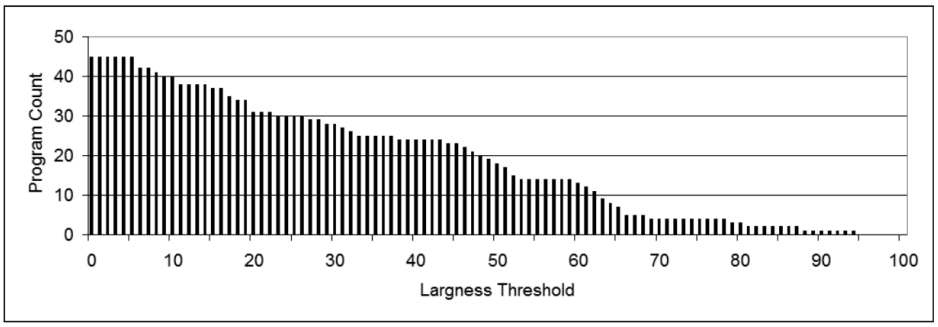
A *Same-Slice-Size Forward Cluster* is a Same-Slice-Size Forward MDS contained within no other Same-Slice-Size Forward MDS.

The observation motivating this approximation is that two slices of the same (large) size are likely to be the same slice. In practice, this approximation is very accurate if a small tolerance for difference is allowed. With a tolerance of 1% the approximation is 99% accurate. However, in the strict case of *zero* tolerance the accuracy falls to 78.3% [Binkley and Harman, 2005b].

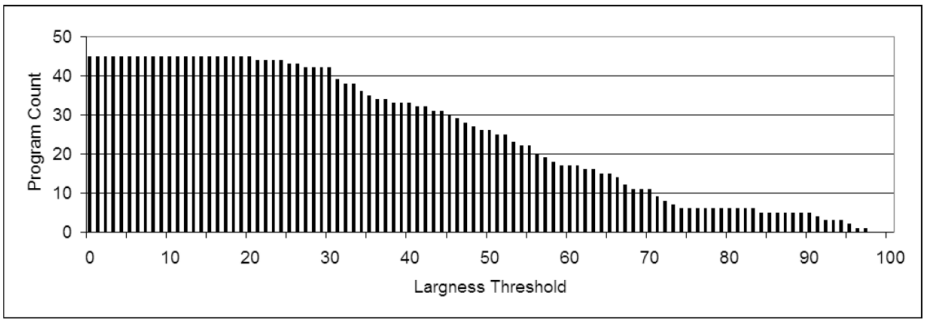
3.10 Existence of dependence clusters in production code

Harman et al. [2009] addressed the question of whether same-slice-size dependence clusters are common in production code. They studied 45 production programs and found that such clusters are indeed prevalent.

Figure 3.12 shows the statistics for the largest same-slice-size dependence clusters present in 45 production programs. The y-axis of the graphs shows the number of programs that have a large cluster which is at least of a certain size. This threshold size is shown on the x-axis. For example, at the threshold of 0%, both graphs show that all 45 programs meet the threshold. At a largeness threshold of 50%, 19 of the 45 programs have a backwards same-slice-size cluster (Figure 3.12a). Similarly at the 50% threshold, 25 programs have a large forward same-slice-size cluster (Figure 3.12b). Even considering very high thresholds such as 75% the graphs show that around 5 programs have a large same-slice-size cluster.



(a) Same-backward-slice clusters



(b) Same-forward-slice cluster

Figure 3.12: Existence of same-slice clusters

3.11 Chapter Summary

This chapter introduces the notion of dependence clusters and the necessary background information. It presents various instantiations of dependence clusters using program slicing. In particular it discusses the same-slice clusters which was introduced for efficient partitioning of slices and subsequently to make it implementable in practice. A further approximation was introduced that replaces the need to compare slice content with slice size. The chapter also discusses dependence intransitivity and illustrates how this gives same-slice clusters internal and external properties. Finally, the chapter shows that same-slice clusters are prevalent in production programs.

Chapter 4

Coherent Clusters

4.1 Overview

This chapter introduces the notion of *coherent dependence clusters* which is a stricter form of dependence clusters. The chapter first formalises coherent dependence clusters and then presents a slice-based instantiation of the definition. It then presents an approximation of coherent clusters which is efficient and accurate making the clustering implementable in practice. This is followed by the introduction of experimental subjects and setup. The chapter then goes on to answer the validation questions about the precision of the efficient approximation for coherent clusters. Finally, a study into the prevalence of coherent clusters in production code is presented. More formally, the chapter addresses the three following research questions:

RQ1.1 How precise is hashing as a proxy for comparing slices?

RQ1.2 How large are coherent clusters that exist in production source code?

RQ1.3 How conservative is using the same-slice relationship as an approximation of slice-inclusion relationship?

4.2 Coherent Dependence Clusters

Coherent clusters are dependence clusters that include not only an internal dependence requirement (each statement of a cluster depends on all the other statements of the cluster) but also an external dependence requirement. The external dependence requirement includes both that each statement of a cluster depends on the same statements external to the cluster and also that it influences the same set of statements external to the cluster. In other words, a

coherent cluster is a set of statements that are mutually dependent and share identical extra-cluster dependence. Coherent clusters are defined in terms of the coherent MDS:

Definition 8 (Coherent MDS and Cluster)
A *coherent MDS* is a MDS V, such that

$\forall x, y \in V : x$ depends on a implies y depends on a and a depends on x implies a depends on y.

A *coherent cluster* is a coherent MDS contained within no other coherent MDS.

The slice-based instantiation of coherent cluster employs both backward *and* forward slices. The combination has the advantage that the entire cluster is both affected by the same set of vertices (as in the case of same-backward-slice clusters) and also affects the same set of vertices (as in the case of same-forward-slice clusters). In the slice-based instantiation, a set of vertices V forms a coherent MDS if

$$\begin{array}{ll} \forall x, y \in V : x \in \mathsf{BSlice}(y) & \text{the internal requirement of an MDS} \\ \quad \wedge\ a \in \mathsf{BSlice}(x) \implies a \in \mathsf{BSlice}(y) & x \text{ and } y \text{ depend on same external } a \\ \quad \wedge\ a \in \mathsf{FSlice}(x) \implies a \in \mathsf{FSlice}(y) & x \text{ and } y \text{ impact on same external } a \end{array}$$

Because x and y are interchangeable

$$\begin{aligned} \forall x, y \in V : \quad & x \in \mathsf{BSlice}(y) \\ & \wedge\ a \in \mathsf{BSlice}(x) \implies a \in \mathsf{BSlice}(y) \\ & \wedge\ a \in \mathsf{FSlice}(x) \implies a \in \mathsf{FSlice}(y) \\ & \wedge\ y \in \mathsf{BSlice}(x) \\ & \wedge\ a \in \mathsf{BSlice}(y) \implies a \in \mathsf{BSlice}(x) \\ & \wedge\ a \in \mathsf{FSlice}(y) \implies a \in \mathsf{FSlice}(x) \end{aligned}$$

This is equivalent to

$$\begin{aligned} \forall x, y \in V : \quad & x \in \mathsf{BSlice}(y) \wedge y \in \mathsf{BSlice}(x) \\ & \wedge\ (a \in \mathsf{BSlice}(x) \Leftrightarrow a \in \mathsf{BSlice}(y)) \\ & \wedge\ (a \in \mathsf{FSlice}(x) \Leftrightarrow a \in \mathsf{FSlice}(y)) \end{aligned}$$

which simplifies to

$$\forall x, y \in V : \mathsf{BSlice}(x) = \mathsf{BSlice}(y) \wedge \mathsf{FSlice}(x) = \mathsf{FSlice}(y)$$

and can be used to define coherent-slice MDS and clusters:

Definition 9 (Coherent-Slice MDS and Cluster)
A *coherent-slice MDS* is a set of SDG vertices, V, such that

$$\forall x, y \in V : \mathsf{BSlice}(x) = \mathsf{BSlice}(y) \wedge \mathsf{FSlice}(x) = \mathsf{FSlice}(y)$$

A *coherent-slice cluster* is a coherent-slice MDS contained within no other coherent-slice MDS.

At first glance the use of both backward and forward slices might seem redundant because $x \in \mathsf{BSlice}(y) \Leftrightarrow y \in \mathsf{FSlice}(x)$. This is true up to a point: for the internal requirement of a coherent-slice cluster, the use of either BSlice or FSlice would suffice. However, the two are not redundant when it comes to the external requirements of a coherent-slice cluster. With a mutually-dependent cluster (Definition 1), it is possible for two vertices within the cluster to influence or be affected by different vertices *external* to the cluster. Neither is allowed with a coherent-slice cluster. To ensure that both external effects are captured, both backward and forward slices are required for coherent-slice clusters.

In Figure 3.10 the set of vertices {a, c} form a coherent cluster as both these vertices have exactly the same backward and forward slices. That is, they share the identical intra- and extra-cluster dependencies. Coherent clusters are therefore a stricter from of same-slice clusters, all coherent clusters are also same-slice MDS but not necessarily maximal. It is worth noting that same-slice clusters partially share extra-cluster dependency. For example, each of the vertices in the same-backward-slice cluster {a, b, c} is dependent on the same set of external statements, but do not influence the same set of external statements.

Coherent slice-clusters have an important property: If a slice contains a vertex of a coherent slice-cluster V, it will contain all vertices of the cluster:

$$\mathsf{BSlice}(x) \cap V \neq \emptyset \implies \mathsf{BSlice}(x) \cap V = V \tag{4.1}$$

This holds because:

$$\forall y, y' \in V \;:\; y \in \mathsf{BSlice}(x) \implies x \in \mathsf{FSlice}(y) \implies x \in \mathsf{FSlice}(y') \implies y' \in \mathsf{BSlice}(x)$$

The same argument clearly holds for forward slices. However, the same is not true for non-coherent clusters. For example, in the case of a same-backward-slice cluster, a vertex contained within the forward slice of any vertex of the cluster is not guaranteed to be in the forward slice of other vertices of the same cluster.

4.3 Hash-based Coherent Slice Clusters

The computation of coherent-slice clusters (Definition 9) grows prohibitively expensive even for mid-sized programs where tens of gigabytes of memory are required to store the set of all possible backward and forward slices. The computation is cubic in time and quadratic in space. An approximation is employed to reduce the computation time and memory requirement. This approximation replaces comparison of slices with comparison of hash values, where hash values are used to summarise slice content. The result is the following approximation to coherent-slice clusters in which H denotes a hash function.

Definition 10 (Hash-Based Coherent-Slice MDS and Cluster)
A *hash-based coherent-slice MDS* is a set of SDG vertices, V, such that

$$\forall x, y \in V : \mathsf{H}(\mathsf{BSlice}(x)) = \mathsf{H}(\mathsf{BSlice}(y)) \wedge \mathsf{H}(\mathsf{FSlice}(x)) = \mathsf{H}(\mathsf{FSlice}(y))$$

A *hash-based coherent-slice cluster* is a hash-based coherent-slice MDS contained within no other hash-based coherent-slice MDS.

The precision of this approximation is empirically evaluated in Section 4.6. From here on, this thesis considers only hash-based coherent-slice clusters unless explicitly stated otherwise. Thus, for ease of reading, hash-based coherent-slice cluster is referred to simply as *coherent cluster*.

4.4 Hash Algorithm

The use of hash values to represent slices reduces both the memory requirement and runtime, as it is no longer necessary to store or compare entire slices. The hash function, denoted H in Definition 10, uses XOR operations iteratively on the unique vertex IDs (of the SDG) which are included in a slice to generate

a hash for the entire slice. We chose XOR as the hash operator because we do not have duplicate vertices in a slice and the order of the vertices in the slice does not matter.

A slice S is a set of SDG vertices $\{v_1, .., v_n\}$ $(n \geq 1)$ and $\mathrm{id}(v_i)$ represents the unique vertex ID assigned by CodeSurfer to vertex v_i, where $1 \leq i \leq n$. The hash function H for S is defined as $\mathsf{H_S}$, where

$$\mathsf{H_S} = \bigoplus_{i=1}^{n} \mathrm{id}(v_i) \tag{4.2}$$

Hence, our hashing algorithm uses XOR operator iteratively on the unique vertex IDs (of the SDG) which are included in a slice to generate a hash for the entire slice. The precision achieved by this hash function in terms of both slicing and clustering are examined and presented in Section 4.6.

4.5 Experimental Subjects and Setup

The slices along with the mapping between the SDG vertices and the actual source code are extracted from the mature and widely used slicing tool *CodeSurfer* [Anderson and Teitelbaum, 2001] (version 2.1). The cluster visualisations were generated by *decluvi* [Islam et al., 2010a] using data extracted from CodeSurfer. The data is generated from slices taken with respect to *source-code representing* SDG vertices. This excludes pseudo vertices introduced into the SDG, e.g., to represent global variables which are modelled as additional pseudo parameters by CodeSurfer. Cluster sizes are also measured in terms of source-code representing SDG vertices, which is more consistent than using lines of code as it is not influenced by blank lines, comments, statements spanning multiple lines, multiple statements on one line, or compound statements. The *decluvi* system along with scheme scripts for data acquisition and pre-compiled datasets for several open-source programs can be downloaded from `http://www.cs.ucl.ac.uk/staff/s.islam/decluvi.html`. Chapter 5 gives further details about *decluvi*.

The study considers the 30 C programs shown in Table 4.1, which provides a brief description of each program alongside seven measures: number of files containing executable C code, LoC – lines of code (as counted by the Unix utility wc), SLoC – the non-comment non-blank lines of code (as counted by the utility sloccount [Wheeler, 2004]), ELoC – the number of source code lines that CodeSurfer considers to contain executable code, the number of SDG vertices,

Program	C Files	LoC	SLoC	ELoC	SDG vertex count	SDG edge count	Total Slices	Largest Coherent Cluster Size	SDG Build Time	Clustering Time	Description
a2ps	79	46,620	22,117	18,799	224,413	2,025,613	97,170	8%	1m52.048s	583m40.758s	ASCII to Postscript
acct	7	2,600	1,558	642	7,618	22,061	2,834	11%	0m15.658s	0m12.545s	Process monitoring
acm	114	32,231	21,715	15,022	159,830	718,683	63,014	43%	1m57.652s	230m18.418s	Flight simulator
anubis	35	18,049	11,994	6,947	112,282	561,160	34,618	13%	0m41.253s	70m54.322s	SMTP messenger
archimedes	1	787	575	454	20,136	91,728	2,176	4%	0m3.658s	0m12.701s	Semiconductor device simulator
barcode	13	3,968	2,685	2,177	16,721	65,367	9,602	58%	0m10.234s	2m56.026s	Barcode generator
bc	9	9,438	5,450	4,535	36,981	355,942	15,076	32%	0m15.359s	13m14.221s	Calculator
byacc	12	6,373	5,312	4,688	45,338	203675	16,590	7%	0m9.820s	10m15.746s	Parser generator
cflow	25	12,542	7,121	5,762	68,782	304,615	24,638	8%	0m23.312s	31m25.104s	Control flow analyser
combine	14	8,202	6,624	5,279	49,288	247,464	29,118	15%	0m14.577s	26m11.625s	File combinator
copia	1	1,168	1,111	1,070	42,435	145,562	6,654	48%	0m2.046s	2m35.680s	ESA signal processing code
cppi	13	6,261	1,950	2,554	17,771	67,217	10,280	13%	0m10.514s	2m33.213s	C preprocessor formatter
ctags	33	14,663	11,345	7,383	152,825	630,189	31,860	48%	0m27.094s	96m0.948s	C tagging
diction	5	2,218	1,613	427	5,919	17,158	2,444	16%	0m5.339s	0m8.189s	Grammar checker
diffutils	23	8,801	6,035	3,638	30,023	113,824	16,122	44%	0m23.384s	9m11.509s	File differencing
ed	8	2,860	2,261	1,788	35,475	142,192	11,376	55%	0m6.602s	6m38.521s	Line text editor
enscript	22	14,182	10,681	9,135	67,405	423,349	33,780	19%	0m44.690s	54m0.652s	File converter
findutils	59	24,102	13,940	9,431	102,910	177,822	41,462	22%	0m36.250s	21m20.795s	Line text editor
flex	21	23,173	12,792	13,537	89,806	860,859	37,748	16%	0m31.249s	77m28.885s	Lexical Analyser
garpd	1	669	509	300	5,452	14,908	1,496	14%	0m1.681s	0m3.670s	Address resolved
gcal	30	62,345	46,827	37,497	860,476	4,565,570	286,000	62%	3m3.946s	5d4h18m35s	Calendar program
gnuedma	1	643	463	306	5,223	14,075	1,488	44%	1m12.888s	0m4.365s	Development environment
gnushogi	16	16,301	11,664	7,175	64,482	277,648	31,298	40%	0m25.907s	47m40.268s	Japanese chess
indent	8	6,978	5,090	4,285	24,109	143,821	7,543	52%	0m10.000s	10m3.012s	Text formatter
less	33	22,661	15,207	9,759	451,870	2,156,420	33,558	35%	1m56.968s	339m48.985s	Text reader
spell	1	741	539	391	6,232	17,574	1,740	20%	0m1.663s	0m4.905s	Spell checker
time	6	2,030	1,229	433	4,946	12,971	3,352	4%	0m3.120s	0m3.683s	CPU resource measure
userv	2	1,378	1,112	1,022	15,418	54,258	5,362	9%	0m13.787s	0m53.332s	Access control
wdiff	4	1,652	1,108	694	10,077	30,085	2,722	6%	0m9.154s	0m39.241s	Diff front end
which	6	3,003	1,996	753	8,830	29,377	3,804	35%	0m4.528s	0m24.215s	Unix utility
Sum	602	356,639	232,623	175,883	2,743,073	14,491,187	864,925	-	16m3.891s	-	
Average	20	11,888	7,754	5,863	91,436	483,040	28,831	27	0m32.130s	-	

Table 4.1: Subject programs

the number of SDG edges, the number of slices produced, and finally the size (as a percentage of the program's SDG vertex count) of the largest coherent cluster. All LoC metrics are calculated over source files that CodeSurfer considers to contain executable code and, for example, do not include header files.

Columns 10 and 11 provide the runtimes recorded during the empirical study. The runtimes reported are wall clock times captured by the Unix time utility while running the experiments on a 64-bit Linux machine (CentOS 5) with eight Intel(R) Xeon(R) CPU E5450 @ 3.00GHz processors and 32GB of RAM. It should be noted that this machine acts as a group server and is accessed by multiple users. There were other CPU intensive processes intermittently running on the machine while these runtimes were collected, and thus the runtimes are only indicative.

Column 10 shows the time needed to build the SDG and the CodeSurfer project that is subsequently used for slicing. The build time for the projects were quite small and the longest build time (2m33.456s) was required for gcal with 46,827 SLoC. Column 11 shows the time needed for the clustering algorithm to perform the clustering and create all the data dumps for *decluvi* to create cluster visualisations. The process completes in minutes for small programs and can take hours and longer for larger programs. It should be noted that the runtime include both the slicing phase which runs in $O(ne)$, where n is the number of SDG vertices and e is the number of edges, and the hashing and clustering algorithm which runs in $O(n^2)$. Therefore the overall complexity is $O(ne)$. The long runtime is mainly due to the current research prototype (which performs slicing, clustering and extraction of the data) using the Scheme interface of CodeSurfer in a pipeline architecture. In future we plan to upgrade the tooling with optimisations for fast and massive slicing [Binkley et al., 2007] and merging the clustering phase into the slicing to reduce the runtime significantly.

Although the clustering and building the visualisation data can take a long time for large projects, it is still useful because the clustering only needs to be done once (for example during a nightly build) and can then be visualised and reused as many times as needed. During further study of the visualisation and the clustering we have also found that small changes to the system do not show a change in the clustering, therefore once the clustering is created it still remains viable through small code changes as the clustering is found to rep-

resent the core program architecture (discussed in Section 7.3). Furthermore, the number of SDG vertices and edges are quite large, in fact even for very small programs the number of SDG vertices are in the thousands with edge counts in tens of thousands. Moreover, the analysis produces a is-in-the-slice-of relation and graph with even more edges. We have tried several clustering and visualisation tools to cluster the is-in-the-slice-of graph for comparison, but most of the tools (such as Gephi [Bastian et al., 2009]) failed due to the large dataset. Other tools such as CCVisu [Beyer, 2008] which were able to handle the large data set simply produced a blob as a visualisation which was not at all useful. The underlying problem is that the is-in-the-slice-of graph is dense and no traditional clustering can handle such dense graphs.

4.6 Validity of the Hash Function

This section addresses research question *RQ1.1 How precise is hashing as a proxy for comparing slices?* The section validates the use of comparing slice hash values in lieu of comparing actual slice content. The use of hash values to represent slices reduce both the memory requirement and runtime, as it is no longer necessary to store or compare entire slices. The hash function, denoted H in Definition 10, determines a hash value for a slice based on the unique vertex ids assigned by CodeSurfer. Validation of this approach is needed to confirm that the hash values provide a sufficiently accurate summary of slices to support the correct partitioning of SDG vertices into coherent clusters. Ideally, the hash function would produce a unique hash value for each distinct slice. The validation study aims to find the number of unique slices for which the hash function successfully produces an unique hash value.

For the validation study we chose 16 programs from the set of 30 subject programs. The largest programs were not included in the validation study to make the study time-manageable. Results are based on both the backward and forward slices for every vertex of these 16 programs. To present the notion of precision we introduce the following formalisation. Let V be the set of all source-code representing SDG vertices for a given program P and US denote the number of *unique slices*:

$$US = \ |\{\mathsf{BSlice}(x) : x \in V\}| + |\{\mathsf{FSlice}(x) : x \in V\}|$$

Note that if all vertices have the same backward slice then $\{\mathsf{BSlice}(x) : x \in V\}$

is a singleton set. Finally, let UH be the number of *unique hash-values*,

$$UH = |\{\mathsf{H}(\mathsf{BSlice}(x)) : x \in V\}| + |\{\mathsf{H}(\mathsf{FSlice}(x)) : x \in V\}|$$

The accuracy of hash function H is given as Hashed Slice Precision, HSP:

$$HSP = \frac{UH}{US}$$

A precision of 1.00 ($US = UH$) means the hash function is 100% accurate (i.e., it produces a unique hash value for every distinct slice) whereas a precision of $1/US$ means that the hash function produces the same hash value for every slice leaving $UH = 1$.

Table 4.2 summarises the results. The first column lists the programs. The second and the third columns report the values of US and UH respectively. The fourth column reports HSP, the precision attained using hash values to compare slices. Considering all 78,587 unique slices the hash function produced unique hash values for 74,575 of them, resulting in an average precision of 94.97%. In other words, the hash function fails to produce unique hash values for just over 5% of the slices. Considering the precision of individual programs, five of the programs have a precision greater than 97%, while the lowest precision, for findutils, is 92.37%. This is, however, a significant improvement over previous use of slice size as the hash value, which is only 78.3% accurate in the strict case of *zero* tolerance for variation in slice contents [Binkley and Harman, 2005b].

Coherent cluster identification uses two hash values for each vertex (one for the backward slice and other for the forward slice) and the slice sizes. Slice size matching filters out some instances where the hash values happen to be the same by coincidence but the slices are different. The likelihood of both hash values matching those from another vertex with different slices is less than that of a single hash matching. Extending US and UH to clusters, Columns 5 and 6 (Table 4.2) report CC, the number of coherent clusters in a program and HCC, the number of coherent clusters found using hashing. The final column shows the precision attained using hashing to identify clusters, HCP.

$$HCP = \frac{HCC}{CC}$$

Program	Unique Slices (*US*)	Unique Hash values (*UH*)	Hashed Slice Precision (*HSP*)	Cluster Count (*CC*)	Hash Cluster Count (*HCC*)	Hash Precision Clusters (*HCP*)
acct	1,558	1,521	97.63%	811	811	100.00%
barcode	2,966	2,792	94.13%	1,504	1,504	100.00%
bc	3,787	3,671	96.94%	1,955	1,942	99.34%
byacc	10,659	10,111	94.86%	5,377	5,377	100.00%
cflow	16,584	15,749	94.97%	8,457	8,452	99.94%
copia	3,496	3,398	97.20%	1,785	1,784	99.94%
ctags	8,739	8,573	98.10%	4,471	4,470	99.98%
diffutils	5,811	5,415	93.19%	2,980	2,978	99.93%
ed	2,719	2,581	94.92%	1,392	1,390	99.86%
findutils	9,455	8,734	92.37%	4,816	4,802	99.71%
garpd	808	769	95.17%	413	411	99.52%
indent	3,639	3,491	95.93%	1,871	1,868	99.84%
time	1,453	1,363	93.81%	760	758	99.74%
userv	3,510	3,275	93.30%	1,827	1,786	97.76%
wdiff	2,190	2,148	98.08%	1,131	1,131	100.00%
which	1,213	1,184	97.61%	619	619	100.00%
Sum	78,587	74,575	–	40,169	40,083	–
Average	4,912	4,661	94.97%	2,511	2,505	99.72%

Table 4.2: Hash function validation

The results show that of the 40,169 coherent clusters, 40,083 are uniquely identified using hashing, which yields a precision of 99.72%. Five of the programs show total agreement, furthermore for each program *HCP* is over 99%, except for userv, which has the lowest precision of 97.76%. This can be attributed to the large percentage (96%) of single vertex clusters in userv. The hash values for slices taken with respect to these single-vertex clusters have a higher potential for collision leading to a reduction in overall precision. In summary, as an answer to *RQ1.1*, the hash-based approximation is found to be sufficiently accurate at 94.97% for slices and at 99.72% for clusters (for the studied programs). Thus, comparing hash values can replace the need to compare actual slices.

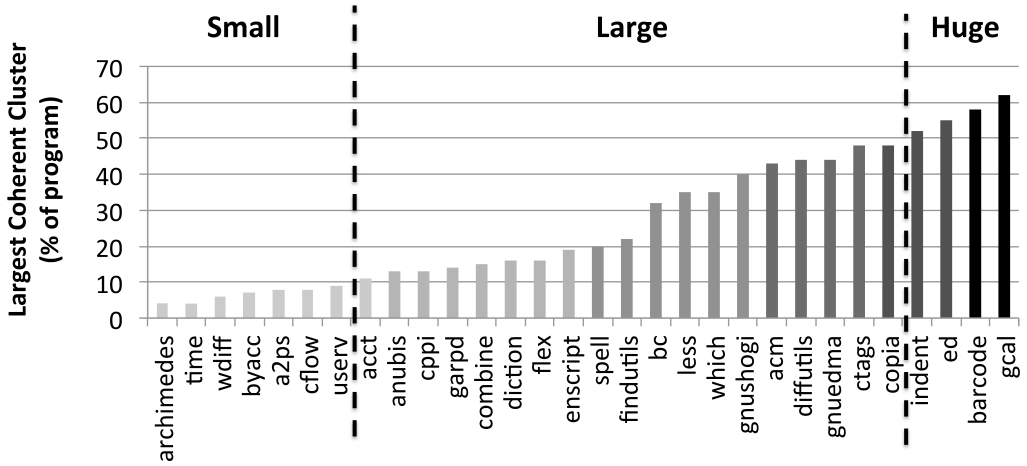


Figure 4.1: Size of largest coherent cluster

4.7 Do large coherent clusters occur in practice?

Having demonstrated that hash function H can be used to effectively approximate slice contents, this section considers the validation research question, *RQ1.2 How large are coherent clusters that exist in production source code?* The question is answered quantitatively using the size of the largest coherent cluster in each program.

To assess if a program includes a *large* coherent cluster requires making a judgement concerning what threshold constitutes large. Following prior empirical work [Binkley and Harman, 2005b, Harman et al., 2009, Islam et al.,

2010a,b], a threshold of 10% is used to classify a large cluster. In other words, a program is said to contain a large coherent cluster if 10% of the program's SDG vertices produce the same backward slice as well as the same forward slice. It should be noted that the other classification names and threshold differ from previous studies as we report on coherent clusters which have stricter properties.

Figure 4.1 shows the size of the largest coherent cluster found in each of the 30 subject programs. The programs are divided into 3 groups based on the size of the largest cluster present in the program.

Small: *Small* consists of seven programs none of which have a coherent cluster constituting over 10% of the program vertices. These programs are archimedes, time, wdiff, byacc, a2ps, cflow and userv. Although it may be interesting to study why large clusters are not present in these programs, this thesis focuses on studying the existence and implications of large coherent clusters.

Large: This group consists of programs that have at least one cluster with size 10% or larger. As there are programs containing much larger coherent clusters, a program is placed in this group if it has a large cluster between the size 10% and 50%. Over two-thirds of the programs studied fall in this category.

The program at the bottom of this group (acct) has a coherent cluster of size 11% and the largest program in this group (copia) has a coherent cluster of size 48%. We present both these programs as case studies and discuss their clustering in detail in Sections 6.2.1 and 6.2.4, respectively. The program bc which has multiple large clusters with the largest of size 32% falls in the middle of this group and is also presented as a case study in Section 6.2.3.

Huge: The final group consists of programs that have a large coherent cluster whose size is over 50%. Out of the 30 programs 4 fall in this group. These programs are indent, ed, barcode and gcal. From this group, we present indent as a case study in Section 6.2.2.

In summary all but 7 of the 30 subject programs contain a large coherent cluster. Therefore, over 75% of the subject programs contain a coherent cluster

of size 10% or more. Furthermore, half the programs contain a coherent cluster of at least 20% in size. It is also interesting to note that although this grouping is based only on the largest cluster, many of the programs contain multiple large coherent clusters. For example, ed, ctags, nano, less, bc, findutils, flex and garpd all have multiple large coherent clusters. It is also interesting to note that there is no correlation between a program's size (measured in SLoC) and the size of its largest coherent cluster. For example, in Table 4.1 two programs of very different sizes, cflow and userv, have similar largest-cluster sizes of 8% and 9%, respectively. Whereas programs acct and ed, of similar size, have very different largest coherent clusters of sizes 11% and 55%.

Therefore as an answer to *RQ1.2*, the study finds that 23 of the 30 programs studied have a large coherent cluster. Some programs also have a huge cluster covering over 50% of the program vertices. Furthermore, the choice of 10% as a threshold for classifying a cluster as large is a relatively conservative choice. Thus, the results presented in this section can be thought of as a lower bound to the existence question.

4.8 Slice inclusion relation vs Same-Slice relation

This thesis has so far dealt with questions regarding same-slice clusters and the specialised version, coherent dependence clusters. Both, coherent and same-slice clusters are built from equivalent slices and only capture a portion of the slice-inclusion relationship. This section presents a preliminary study which assesses the conservatism introduced in using this approximation. This section addresses research question *RQ1.3 How conservative is using the same-slice relationship as an approximation of slice-inclusion relationship?*

As equivalence slices capture only a portion of slice-inclusion relationship they also yield smaller clusters, and thus a conservative result. A more liberal approach would require using mutual-inclusion to perform clustering which is the NP-Hard *maximal cliques* problem [Bomze et al., 1999]. A preliminary experiment was designed to gain a better understanding of how conservative it is to use slice equivalence. The experiment compares the number of pairs of SDG vertices that are in each other's slices to the number of pairs where both vertices have the same slice, to find their ratio R.

$$R = \frac{|\{(x,y) : \mathsf{BSlice}(x) = \mathsf{BSlice}(y)\}|}{|\{(x,y) : x \in \mathsf{BSlice}(x) \wedge y \in \mathsf{BSlice}(y)\}|}$$

A large difference would give cause for further research into developing better detection algorithms for dependence clusters. Due to the large runtimes for this experiment (bc required around 30 days) the study was limited to a subset of the test subjects presented in Section 4.5. The test subjects considered and the results are shown in Table 4.3. The table shows various statistics about the relationship study, Column 1 lists the program, Column 2 gives the vertex count, Column 3 show the number of slice comparisons that were done, Column 4 gives the number of vertex pairs that were in each other's slice, Column 5 gives the number of vertex that yield the same slice. Finally, the last column gives the percentage of mutually-interdependent vertices that also have exactly the same slice.

Program	**Vertex Count**	**Slice Comparisons**	**Slice-inclusion Count**	**Same-Slice Count**	**Percentage**
acct	1,417	2,007,889	335,477	70,205	21%
barcode	4,801	23,049,601	8,291,421	7,894,155	95%
bc	7,538	56,821,444	30,935,773	18,354,919	59%
copia	3,327	11,068,929	2,587,437	2,584,221	99%
diffutils	8,061	64,979,721	19,671,054	13,413,596	68%
ed	5,688	32,353,344	15,082,352	14,703,994	97%
time	838	2,808,976	4,540	2,320	51%
wdiff	1,361	1,852,321	16,985	9,131	54%
which	1,902	3,617,604	1,213,936	947,870	78%
Average	3,881	22,062,203	8,682,108	6,442,268	69%

Table 4.3: Slice inclusion vs Same-slice Study

The results of the experiment shows the percentage can vary from 21% for acct to 99% for copia. The sizes of the program (vertex count) does not seem to have a correlation with the percentage of mutually-dependent vertices that produce the same slice. For example, smaller programs like acct and wdiff with similar vertex count have significantly different percentages at 21% and 54%. Whereas, bc which is almost 6 times larger than wdiff has similar percentage value to wdiff at 59%. The highest agreement is seen for copia at 99%. One possible reason for this could be that copia is a very tightly knit program where all the program logic is bound to a central mutually recursive structure. The

program Copia is discussed in detail in Section 6.2.4

As an answer to *RQ1.3*, we find that in the programs studied, on average 69% of the vertices that are mutually-dependent also have the same slice. This finding is used to motivate the study (Section 7.4) of inter-cluster dependence based on cluster inclusion relationship which results in larger dependence structures than any same-slice cluster. Although this result suggests the need for further research into new slice-inclusion based cluster identification techniques, further experiments need to be conducted on the remaining test subjects to gain a more generalised answer.

4.9 Chapter Summary

This chapter introduces the notion of coherent dependence clusters and presents definitions for its slice-based instantiation. The slice-based instantiation enables the identification of dependence clusters using program slicing. The chapter also introduces an efficient approximation for coherent clusters which enables the identification of such clusters using standard desktop and server systems. The new approximation is also found to have a precision of 99.72% which is over 20% higher than those used in previous studies.

Finally, using a three-tier classification where programs are divided into three groups (small, large, huge) depending on the size of the largest coherent cluster in the program, an empirical study finds that over 75% of the subject programs contain large clusters. This high occurrence of large coherent cluster indicates that these need to be studied and understood to appreciate their influence on program comprehension and maintenance.

Chapter 5

Cluster Visualisation

5.1 Overview

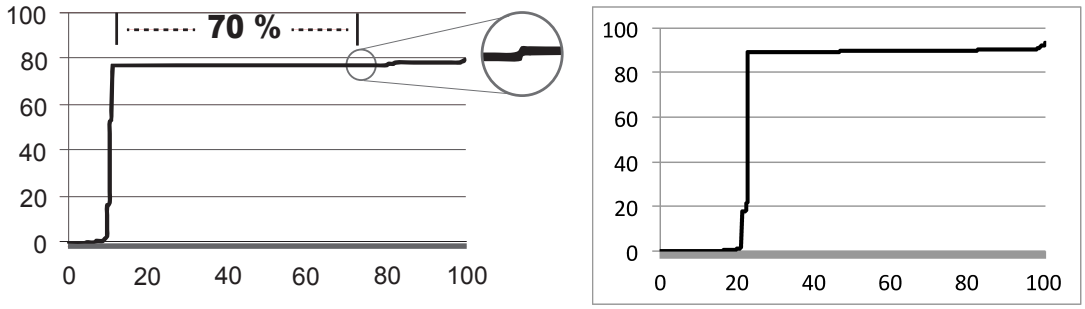
This chapter introduces the various cluster visualisation techniques. The chapter firstly introduces two graph-based visualisations which is followed by a discussion of the cluster visualisation tool *decluvi*. Graph-based visualisation is used to highlight patterns of clustering found in the subject programs. The cluster visualisation of *decluvi* is used in Chapter 6 to study clustering of programs in detail and identify program structures revealed by coherent cluster visualisation. The chapter also presents a study on the effect of pointer analysis accuracy on size of slices and the impact on coherent cluster sizes and patterns. Formally, this chapter addresses the following research question:

RQ2.1 Which patterns of clustering can be identified using graph-based cluster visualisation?

RQ2.2 What is the effect of pointer analysis precision on coherent clusters?

5.2 Graph Based Cluster Visualisation

This section presents two graph-based visualisations. The first visualisation is the Monotone Slice-size graph (MSG), it was introduced by Binkley and Harman [2005b] to visualise slice sizes and estimate the presence of same-slice-size clusters. The second graph-based visualisation is the Slice/Cluster size graph (SCG) [Islam et al., 2010a], which is an extension of the MSG to overcome the precision problems of the MSG and show clearer link between the slices and clusters.



(a) B-MSG (b) F-MSG

Figure 5.1: Monotone Slice-size Graph (MSG) for the program bc. The x-axis plots vertices with the slices in monotonically increasing order and the y-axis plots the size of the backward/forward slice.

5.2.1 Monotone Slice-Size Graph (MSG)

The first visualisation, the *Monotone Slice-size Graph* (MSG) [Binkley and Harman, 2005b], plots a landscape of monotonically increasing slice sizes where the y-axis shows the size of each slice, as a percentage of the entire program, and the x-axis shows each slice, in monotonically increasing order of slice size. In an MSG, a dependence cluster appears as a sheer-drop cliff face followed by a plateau. The visualisation assists 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?). An MSG drawn using backward slice sizes is referred to as a backward-slice MSG (B-MSG), and an MSG drawn using forward slice sizes is referred to as a forward-slice MSG (F-MSG).

As an example, the open source calculator bc contains 9,438 lines of code represented by 7,538 SDG vertices. The B-MSG for bc, shown in Figure 5.1a, contains a large plateau that spans almost 70% of the MSG. Under the assumption that same slice size implies the same slice, this indicates a large same-slice cluster. However, "zooming" in reveals that the cluster is actually composed of several smaller clusters made from slices of very similar size. The tolerance implicit in the visual resolution used to plot the MSG obscures this detail.

5.2.2 Slice/Cluster Size Graph (SCG)

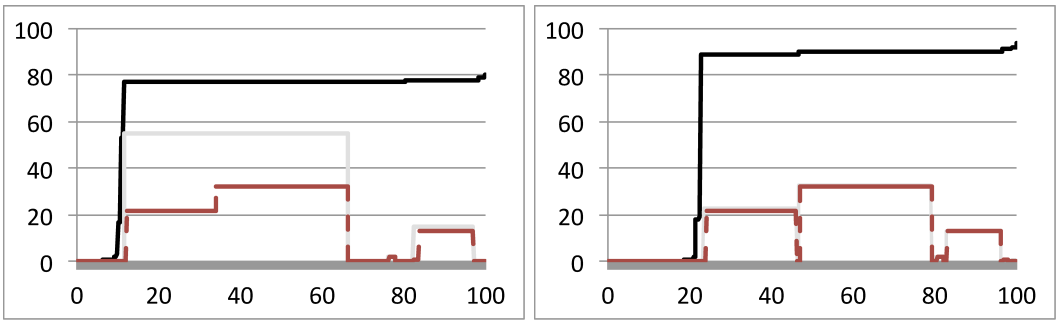
The second graph-based visualisation is the *Slice/Cluster Size Graph* (SCG) [Islam et al., 2010b], that alleviates the resolution problem by combining both

slice and cluster sizes. It plots three landscapes, one of increasing slice sizes, one of the corresponding same-slice cluster sizes, and the third of the corresponding coherent cluster sizes. In the SCG, vertices are ordered along the x-axis using three values, primarily according to their slice size, secondarily according to their same-slice cluster size, and finally according to the coherent cluster size. Three values are plotted on the y-axis: slice sizes form the first landscape, and cluster sizes form the second and third. Thus, SCGs not only show the sizes of the slices and the clusters, they also show the relation between them and thus bring to light interesting links. Two variants of the SCG are considered: the backward-slice SCG (B-SCG) is built from the sizes of backward slices, same-backward-slice clusters, and coherent clusters, while the forward-slice SCG (F-SCG) is built from the sizes of forward slices, same-forward-slice clusters, and coherent clusters. Note that both backward and forward SCGs use the same coherent cluster sizes.

The B-SCG and F-SCG for the program `bc` are shown in Figure 5.2. In both graphs the slice size landscape is plotted using a solid black line, the same-slice cluster size landscape using a grey line, and the coherent cluster size landscape using a (red) broken line. The B-SCG (Figure 5.2a) shows that `bc` contains two large same-backward-slice clusters consisting of almost 55% and almost 15% of the program. Surprisingly, the larger same-backward-slice cluster is composed of smaller slices than the smaller same-backward-slice cluster; thus, the smaller cluster has a bigger impact (slice size) than the larger cluster. In addition, the presence of three coherent clusters spanning approximately 15%, 20% and 30% of the program's statements are also visible in the graphs.

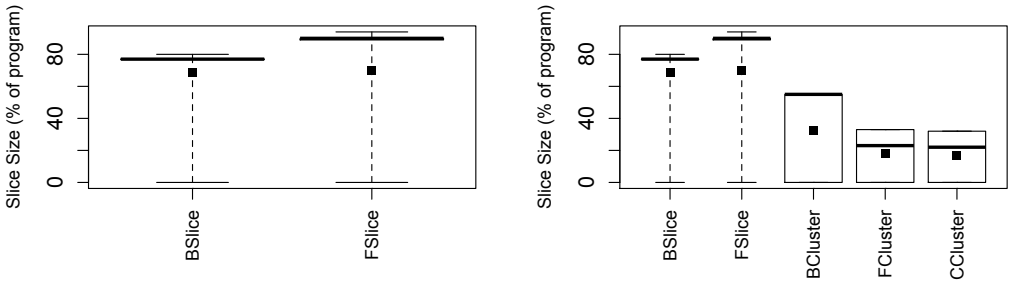
5.2.3 Box Plot Visualisation

Figure 5.3a shows two box plots depicting the distribution of (backward and forward) slice sizes for `bc`. The average size of the slices are also displayed in the box plot using a solid square box. Comparing the box plot information to the information provided by the MSGs, we can see that all the information available from the box plots can be derived from the MSGs itself (except for the average). However, MSGs show a landscape (slice profile) which cannot be obtained from the box plots. Similarly, the box plots in Figure 5.3b show the size distributions of the various clusters (i.e. a vertex is in a cluster of size x) in addition to the slice size distributions. Although the information from these



(a) B-SCG (b) F-SCG

Figure 5.2: Slice/Cluster Size Graph (SCG) for the program bc. The x-axis plots vertices ordered by monotonically increasing order of slices, same size clusters and coherent clusters. The y-axis plots the size of the backward/forward slices, same-slice clusters and coherent clusters.



(a)Slice Size Distribution (b) Slice/Cluster Size Distribution

Figure 5.3: Slice/Cluster size distribution for bc

box plots can not be derived from the SCGs shown in Figures 5.2a and 5.2b directly, the profiles (landscapes) give a better intuition about the clusters, the number of major clusters and their sizes. For our empirical study we use the size of individual clusters and the cluster profile to find mappings between the clusters and program components. Therefore, we drop box plots in favour of SCGs to show the cluster profile and provide additional statistics in tabular format where required.

5.3 Patterns of clustering

This section presents a visual study of SCGs for the three program groups (identified in Section 4.7) and addresses research question *RQ2.1.* Figures 5.4–5.6 show graphs for the three categories (small, large and huge). The graphs in

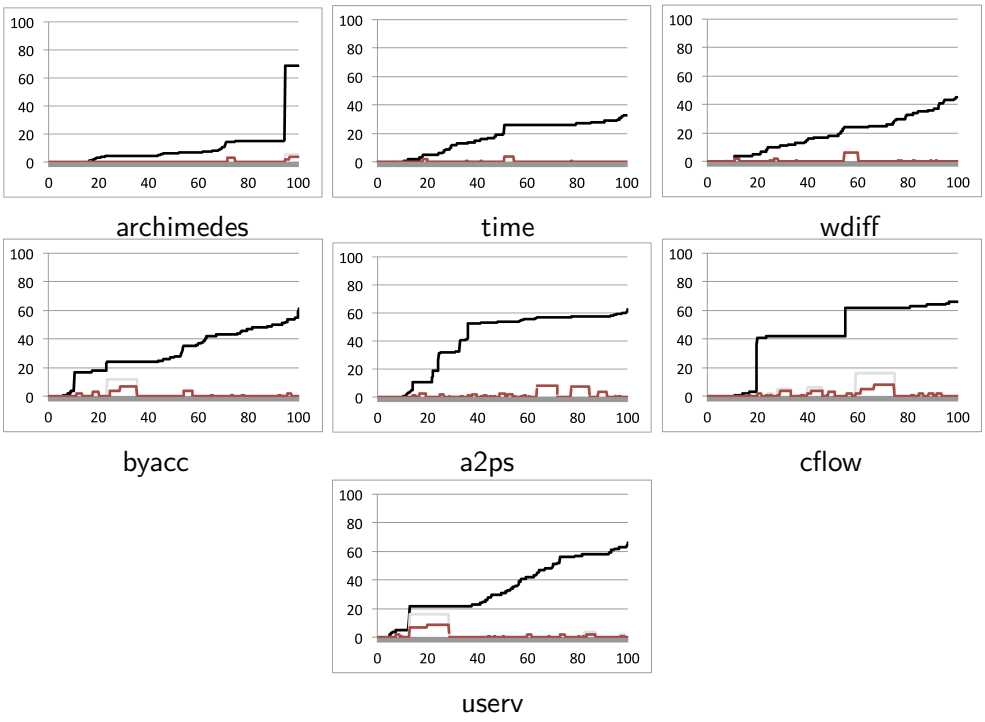


Figure 5.4: Programs with *small* coherent clusters

the figures are laid out in ascending order based on the largest coherent cluster present in the program and thus follow the same order as seen in Figure 4.1.

Figure 5.4 shows SCGs for the seven programs of the *small* group. In the SCGs of the first three programs (archimedes, time and wdiff) only a small coherent cluster is visible in the red landscape. In the remaining four programs, the red landscape shows the presence of multiple small coherent clusters. It is very likely that, similar to the results of the case studies presented later, these clusters also depict logical constructs within each program.

Figure 5.5 shows SCGs of the 19 programs that have at least one large, but not huge, coherent cluster. That is, each program has at least one coherent cluster covering 10% to 50% of the program. Most of the programs have multiple coherent clusters as is visible on the red landscape. Some of these have only one large cluster satisfying the definition of *large*, such as acct. The clustering of acct is discussed in further detail in Section 6.2.1. Most of the remaining programs are seen to have multiple large clusters such as bc, which is also discussed in further detail in Section 6.2.3. The presence of multiple large coherent cluster hints that the program consists of multiple functional components. In three of the programs (which, gnuedma and copia) the landscape is

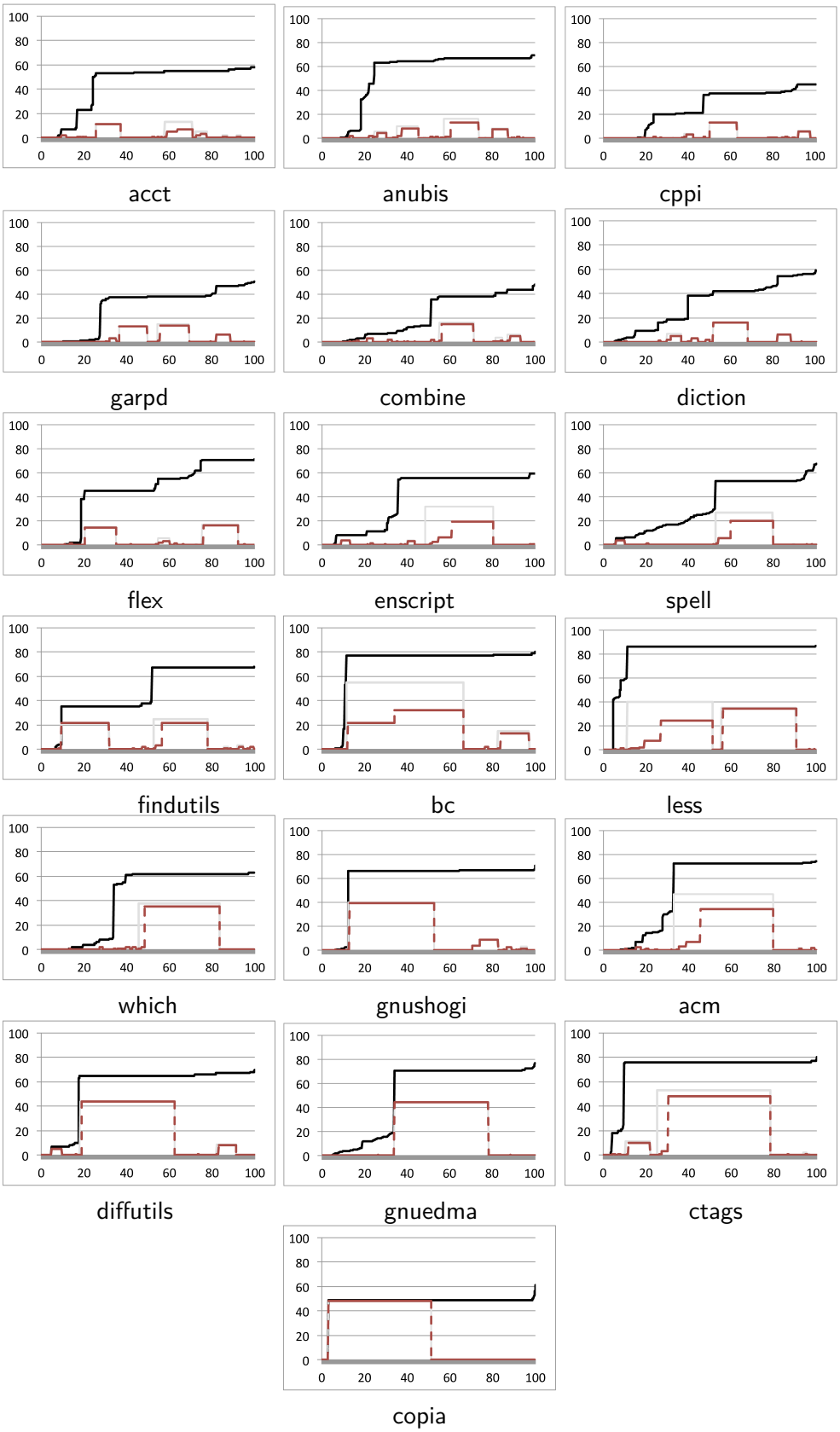


Figure 5.5: Programs with *large* coherent clusters

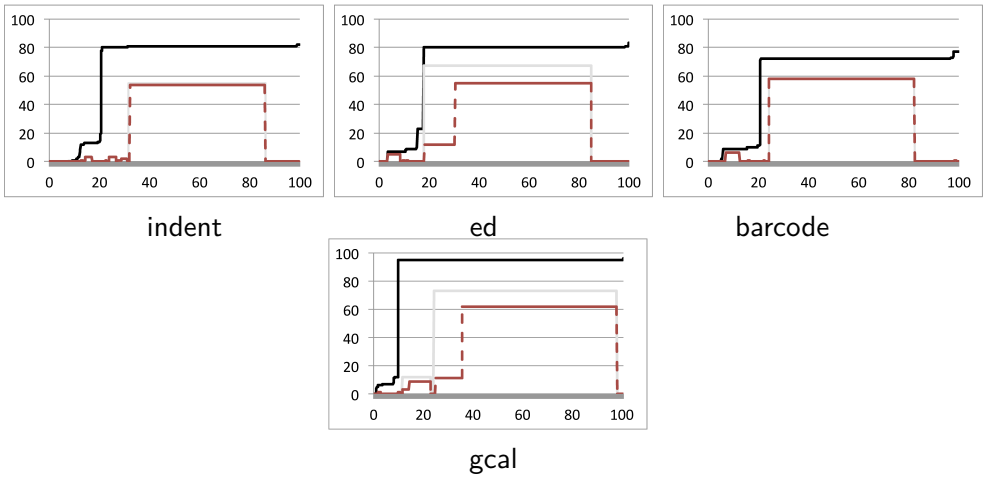


Figure 5.6: Programs with *huge* coherent clusters

completely dominated by a single large coherent cluster. In which and gnuedma this cluster covers around 40% of the program vertices whereas in copia the cluster covers 50%. The presence of a single large dominating cluster points to a centralised functionality or structure being present in the program. Copia is presented as a case study in Section 6.2.4 where its clustering is discussed in further detail.

Finally, SCGs for the four programs that contain *huge* coherent clusters (with size >50%) are found in Figure 5.6. In all four landscapes there is a very large dominating cluster with other smaller clusters also being visible. This pattern supports the conjecture that the program has one central structure or functionality which consists of most of the program elements, but also has additional logical constructs that work in support of the central idea. Indent is one program that falls in this category and is discussed in further detail in Section 6.2.2.

As an answer to *RQ2.1*, the study finds that most programs contain multiple coherent clusters. Furthermore, the visual study reveals that a third of the programs have multiple large coherent clusters. Only three programs copia, gnuedma, and which show the presence of only a single (overwhelming) cluster covering most of the program. Having shown that coherent clusters are prevalent in programs and that most programs have multiple significant clusters, Chapter 6 presents a series of four case studies that looks at how program structures are represented by these clusters.

5.4 Cluster Splitting

Coherent clusters are a stricter form of same-slice clusters, in fact, the intersection of a same-backward-slice cluster with a same-forward-slice cluster will produce a coherent MDS and if the intersection is not empty, a coherent cluster. This often leads to observed *splitting* of the same-backward-slice clusters and same-forward-slice clusters. This splitting can be seen visually in the B-SCG for `bc` (Figure 5.2a), which includes a large same-backward-slice cluster (the grey landscape) that runs from 10% to 65% on the horizontal axis. The statements that make up this same-backward-slice cluster break in two coherent-slice clusters (the dashed landscape): the first runs from 10% to 35% and the second from 35% to 65%. Since these two coherent-slice clusters comprise the same statements (the same segment of the x-axis) they represent a splitting of the single same-backward-slice cluster. This splitting phenomenon is found to be very common and almost all programs exhibit this phenomenon in either their B-SCG or F-SCG. It should be noted that it is possible for same-backward-slice and same-forward-slice clusters for the same program to have different size and frequency, and thereby capture different properties of the program.

5.5 Impact of Pointer Analysis Precision

Recall that the definition of a coherent dependence cluster is based on an underlying *depends-on* relation, which is approximated using program slicing. Pointer analysis plays a key role in the precision of slicing and the interplay between pointer analysis and downstream dependence analysis precision is complex [Shapiro and Horwitz, 1997]. To understand how pointer analysis precision impacts the clustering of the programs we study the effect in this section.

Usually, one would choose the pointer analysis with the highest precision but there may be situations where this is not possible and one has to revert to lower precision analysis. This section presents a study on the effect of various levels of pointer analysis precision on the size of slices and subsequently on coherent clusters. It addresses research question *RQ2.2 What is the effect of pointer analysis precision on coherent clusters?*

CodeSurfer provides three levels of pointer analysis precision (Low, Medium, and High) that provide increasingly precise points-to information at the expense of additional memory and analysis time. The Low setting uses a

minimal pointer analysis that assumes every pointer may point to every object that has its address taken (variable or function). At the Medium and High settings, CodeSurfer performs extensive pointer analysis using the algorithm proposed by Fahndrich et al. [1998], which implements a variant of Andersen's pointer analysis algorithm [Andersen, 1994] (this includes parameter aliasing). At the Medium setting, fields of a structure are not distinguished while the High setting distinguishes structure fields. The High setting should produce the most precise slices but requires more memory and time during SDG construction, which puts a functional limit on the size and complexity of the programs that can be handled by CodeSurfer. There is no automatic way to determine whether the slices are correct and precise, Weiser [1984] considers smaller slices to be better. Slice size is often used to measure the impact of the analysis precision [Shapiro and Horwitz, 1997], similarly we also use slice size as a measure of precision.

The study compares slice and cluster size for CodeSurfer's three precision options (Low, Medium, High) to understand the impact of pointer analysis precision on slice and cluster size. The results are shown in Table 5.1. Column 1 lists the programs and the other columns present the average slice size, maximum slice size, average cluster size, and maximum cluster size, respectively, for each of the three precision settings. The results for average slice size deviation and largest cluster size deviation are visualised in Figures 5.7 and 5.8. The graphs use the High setting as the base line and show the percentage deviation when using the Low and Medium settings.

Figure 5.7 shows the average slice size deviation when using the lower two settings compared to the highest. On average, the Low setting produces slices that are 14% larger than the High setting. Program userv has the largest deviation of 37% when using the Low setting. For example, in userv the minimal pointer analysis fails to recognise that the function pointer oip can never point to functions sighandler_alrm and sighandler_child and includes them as called functions at call sites using *oip, increasing slice size significantly. In all 30 programs, the Low setting yields larger slices compared to the High setting.

The Medium setting always yields smaller slices when compared to the Low setting. For eight programs, the Medium setting produces the same average slice size as the High setting. For the remaining programs the Medium

Program	Average Slice Size			Max Slice Size			Average Cluster Size			MaxCluster Size		
	L	M	H	L	M	H	L	M	H	L	M	H
a2ps	25223	23085	20897	45231	44139	43987	2249	1705	711	10728	9295	4002
acct	763	700	621	1357	1357	1357	79	66	40	272	236	162
acm	19083	17997	16509	29403	28620	28359	3566	3408	4197	9356	9179	10809
anubis	11120	10806	9085	16548	16347	16034	939	917	650	2708	2612	2278
archimedes	113	113	113	962	962	962	3	3	3	39	39	39
barcode	3523	3052	2820	4621	4621	4621	1316	1870	1605	2463	2970	2793
bc	5278	5245	5238	7059	7059	7059	1185	1188	1223	2381	2384	2432
byacc	3087	2936	2886	9036	9036	9036	110	110	103	583	583	567
cflow	7314	5998	5674	11856	11650	11626	865	565	246	3060	2191	1097
combine	3512	3347	3316	13448	13448	13448	578	572	533	2252	2252	2161
copia	1844	1591	1591	3273	3273	3273	1566	1331	1331	1861	1607	1607
cppi	1509	1352	1337	4158	4158	4158	196	139	139	825	663	663
ctags	12681	11663	11158	15483	15475	15475	7917	4199	3955	11080	7905	7642
diction	421	392	387	1194	1194	1194	46	37	37	217	196	196
diffutils	5049	4546	4472	7777	7777	7777	3048	1795	1755	4963	3596	3518
ed	4203	3909	3908	5591	5591	5591	2099	1952	1952	3281	3146	3146
enscript	7023	6729	6654	16130	16130	16130	543	554	539	3140	3242	3243
findutils	7020	6767	5239	11075	11050	11050	1969	1927	1306	4489	4429	2936
flex	9038	8737	8630	17257	17257	17257	622	657	647	3064	3064	3064
garpd	284	242	224	628	628	628	32	31	29	103	103	103
gcal	132860	123438	123427	142739	142289	142289	40885	40614	40614	93541	88532	88532
gnuedma	385	369	368	730	730	368	178	176	174	333	331	330
gnushogi	9569	9248	9141	14726	14726	14726	1577	2857	2820	3787	6225	6179
indent	4104	4058	4045	5704	5704	5704	2036	2032	1985	3402	3399	3365
less	13592	13416	13392	16063	16063	16063	4573	3074	3035	7945	5809	5796
spell	359	293	291	845	845	845	58	31	48	199	128	174
time	201	161	158	730	730	730	4	3	3	35	33	33
userv	1324	972	964	2721	2662	2662	69	32	53	268	154	240
wdiff	687	582	561	2687	2687	2687	33	21	19	184	158	158
which	1080	1076	1070	1744	1744	1744	413	413	410	798	798	793

Table 5.1: CodeSurfer pointer analysis

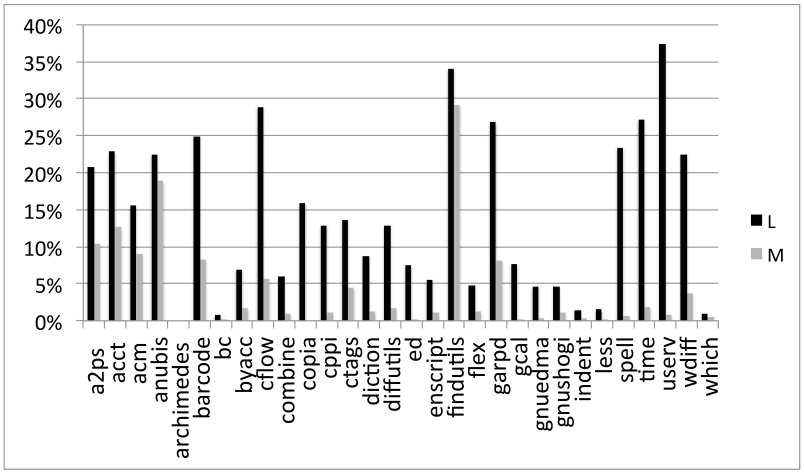


Figure 5.7: Percentage deviation of average slice size for Low and Medium CodeSurfer pointer analysis settings

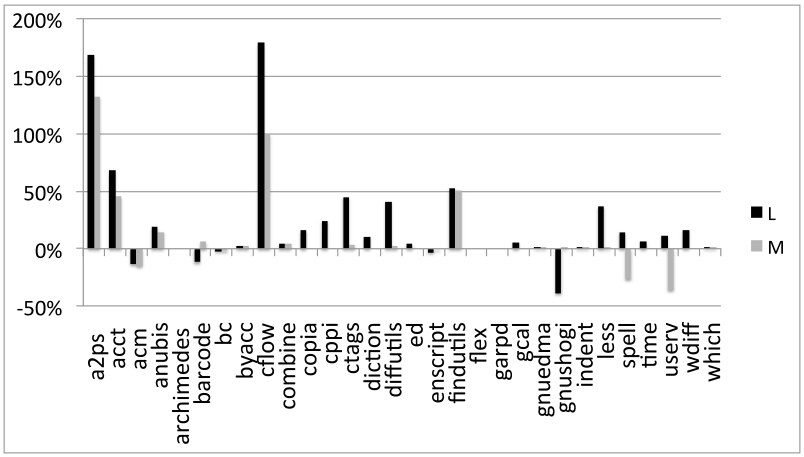


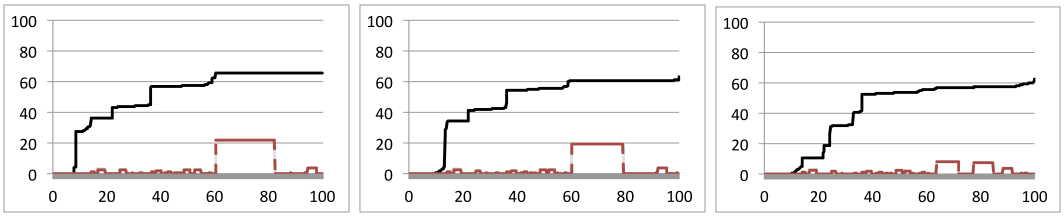
Figure 5.8: Percentage deviation of largest cluster size for Low and Medium CodeSurfer pointer analysis settings

setting produces slices that are on average 4% larger than when using the High setting. The difference in slice size occurs because the Medium setting does not differentiate between structure fields, which the High setting does. The largest deviation is seen in findutils at 34%. With the Medium setting, the structure fields (options, regex_map, stat_buf and state) of findutils are lumped together as if each structure were a scalar variable, resulting in larger, less precise, slices.

The increase in slice size observed is expected to result in larger clusters due to the loss of precision. The remainder of this section studies the effect of pointer analysis on cluster size. Figure 5.8 visualises the deviation of the largest coherent cluster size when using the lower two settings compared to the

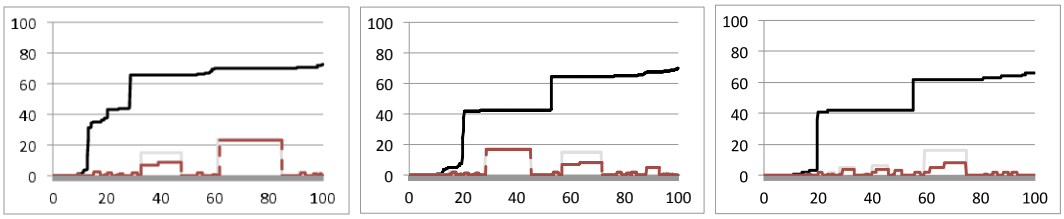
highest. The graph shows that the size of the largest coherent clusters found when using the lower settings is larger in most of the programs. On average there is a 22% increase in the size of the largest coherent cluster when using the Low setting and a 10% increase when using the Medium setting. In a2ps and cflow the size of the largest cluster increases over 100% when using the Medium setting and over 150% when using the Low setting.

The B-SCGs for a2ps for the three settings is shown in Figure 5.9a. In the graphs it is seen that the slice sizes get smaller and have increased steps in the (black) landscape indicating that they become more precise. The red landscape shows that there is a large coherent cluster detected when using the Low setting running from approx. 60% to 85% on the x-axis. This cluster drops in size when using the Medium setting. At the High setting this coherent cluster breaks up into multiple smaller clusters which causes a drop in the cluster sizes.



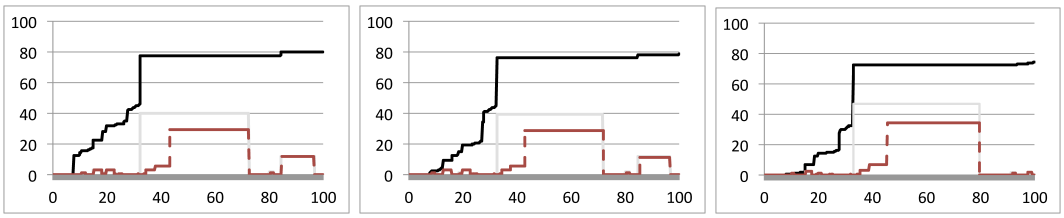
Low Medium High

(a) a2ps



Low Medium High

(b) cflow



Low Medium High

(c) acm

Figure 5.9: SCGs for Low, Medium and High pointer settings of CodeSurfer

In the SCGs for cflow (Figure 5.9b) a similar drop in the slice size and cluster size is observed. However, unlike a2ps the large coherent cluster does not break into smaller clusters but only drops in size. The largest cluster when using the Low setting runs from 60% to 85% on the x-axis. This cluster reduces in size and shifts position running 30% to 45% x-axis when using the Medium setting. The cluster further drops in size down to 5% running 25% to 30% on the x-axis when using the High setting. In this case the largest cluster has a significant drop in size but does not break into multiple smaller clusters.

```
f6(x) {
  f = *p(42, 4);
  return f;
}
```

Figure 5.10: Replacement for coherent cluster example

Surprisingly, Figure 5.8 also shows seven programs where the largest coherent cluster size actually increases when using the highest pointer analysis setting on CodeSurfer. Figure 5.9c shows the B-SCGs for acm which falls in this category. This counter-intuitive result is seen only when the more precise analysis determines that certain functions cannot be called and thus excludes them from the slice. Although in all such instances slices get smaller, the clusters may grow if the smaller slices match other slices already forming a cluster.

For example, consider replacing function f6 in Figure 3.9 with the code shown in Figure 5.10, where f depends on a function call to a function referenced through the function pointer p. Assume that the highest precision pointer analysis determines that p does not point to f2 and therefore there is no call to f2 or any other function from f6. The higher precision analysis would therefore determine that the forward slices and backward slices of a, b and c are equal, hence grouping these three vertices in a coherent cluster. Whereas the lower precision is unable to determine that p cannot point to f2, the backward slice on f will conservatively include b. This will lead the higher precision analysis to determine that the set of vertices {a, b, c} are one coherent cluster whereas the lower precision analysis include only set of vertices {a, c} in the same coherent cluster.

Although we do not explicitly report the project build times on CodeSurfer and the clustering runtimes for the lower settings, it has been our experience

that in the majority of the cases the build times for the lower settings were smaller. However, as lower pointer analysis settings yield large points-to sets and subsequently larger slices, the clustering runtimes were higher than when using the highest setting. Moreover, in some cases with lower settings there was an explosive growth in summary edge generation which resulted in exceptionally high project build times and clustering runtimes.

As an answer to *RQ2.2*, we find that in the majority of the cases the Medium and Low settings result in larger coherent clusters when compared to the High setting. For the remaining cases we have identified valid scenarios where more precise pointer analysis can result in larger coherent clusters. The results also confirm that a more precise pointer analysis leads to more precise (smaller) slices. Because it gives the most precise slices and most accurate clusters, the experiments in this thesis uses the highest CodeSurfer pointer analysis setting.

5.6 Cluster Visualisation Tool

This section gives details of the *decluvi* tool. It describes the design considerations, multi-level visualisations and the evaluation of the interface.

5.6.1 Design Consideration

Several guidelines have been proposed for the construction of effective visualisation tools. Two of these are used to ensure that *decluvi* is of high-quality. First is the framework proposed by Maletic et al. [2002] and second is the interface requirements proposed by Shneiderman [1996]. Maletic et al.'s framework considers the *why*, *who*, *what*, *where*, and *how* of a visualisation. For *decluvi* this leads to the following:

Tasks – *why will the visualisation help?*

The visualisation helps to quickly identify clusters of dependence and their interaction in programs. In Chapter 6 we will see that the visualisation helps to understand program structure and helps in highlighting potential re-structuring opportunities to improve cohesion and design abstraction. The visualisation therefore makes it easier to understand and modify programs.

Audience – *who will make use of the visualisation?*

Developers will use the visualisation to gain an understanding of the functional components or the logical program structure where they are

unfamiliar with the system. Others will be able to use the visualisation to check if their implementation matches their documented architecture and to identify potential problems in the structure. Maintainers will also use the visualisation to gain overall understanding of the program and estimate the impact of changes.

Target – *what data source is to be represented?*

Details of dependence clusters calculated from program.

Medium – *where to represent the visualisation?*

The visualisation will involve highly interactive computer graphics being displayed on a colour monitor.

Representation – *how to represent the data?*

The representation of the data will be through various abstract and concrete views, allowing both an overall architectural understanding of the system and also details of the implementation.

Shneiderman's requirements are aimed at providing high-quality interfaces for visualisation tools. They include *Overview, Zoom, Filter, Details-on-demand, Relate, History* and *Extract*. These features were used to guide the development of *decluvi* and are presented in Section 5.6.4 making it possible to evaluate the tool's interface against these requirements.

5.6.2 Multi-level Views

Cluster visualisations such as the SCG can provide an engineer a quick high-level overview of how difficult a program will be to work with. High-level abstraction can cope with a tremendous amount of code (millions of lines) and reveal the high-level structure of a program. This overview can help an engineer form a mental model of a program's structure and consequently aid in tasks such as comprehension, maintenance, and reverse engineering [Balzer et al., 2004]. However, the high-level nature of the abstraction implies less detail. Furthermore, programmers are most comfortable in the spatial structure in which they read and write (i.e., that of source code). To accommodate the need for multiple levels of abstraction, the cluster visualisation tool *decluvi* provides four views: a *Heat Map* view and three different source-code views. The latter three include the *System View*, the *File View*, and the *Source View*, which allow a program's clusters to be viewed at increasing levels of detail. A common colouring scheme is used in all four views to help tie the different

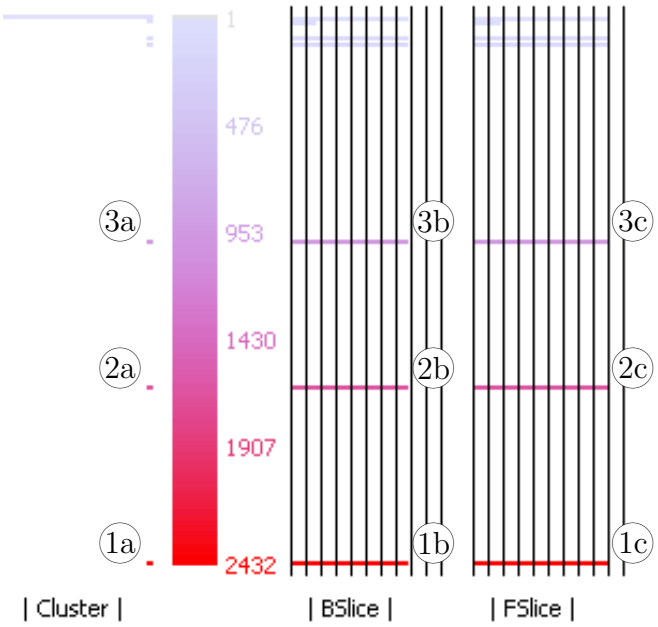


Figure 5.11: Heat Map View for bc

views together.

Heat Map View

The *Heat Map view* aids an engineer in creating a mental model of the overall system. This overview can be traced to the source code using the other three views. The Heat Map provides a starting point that displays an overview of all the clusters using colour to distinguish clusters of varying sizes. The view also displays additional statistics such as the size of the backward and forward slices for each coherent cluster and the number of clusters of each size. Figure 5.11 shows the Heat Map for bc, which has been annotated for the purpose of this discussion. The three labels 1a, 1b, and 1c highlight statistics for the largest cluster (Cluster 1) of the program, whereas 2a, 2b, and 2c highlight statistics of the 2^{nd} largest cluster (Cluster 2) and the 3's the 3^{rd} largest cluster (Cluster 3). Starting from the left of the Heat Map, using one pixel per cluster, horizontal lines (limited to 100 pixels) show the number of clusters that exist for each cluster size. This helps identify cases where there are multiple clusters of the same size. For example, the single dot next to the labels 1a, 2a and 3a depict that there is one cluster for each of the three largest sizes. A single occurrence is common for large clusters, but not for small clusters as illustrated by the long line at the top left of the Heat Map, which indicates multiple (uninteresting) clusters of size one.

To the right of the cluster counts is the actual Heat Map (colour spectrum) showing cluster sizes from small to large reading from top to bottom using colours varying from blue to red. In grey scale this appear as shades of grey, with lighter shades (corresponding to blue) representing smaller clusters and darker shades (corresponding to red) representing larger clusters. Red is used for larger clusters as they are more likely to encompass complex functionality, making them more important "hot topics."

A numeric scale on the right of the Heat Map shows the cluster size (measured in SDG vertices). For program bc, the scale runs from 1 – 2432, depicting the sizes of the smallest cluster, displayed using light blue (light grey), and the largest cluster, displayed in bright red (dark grey).

Finally, on the right of the number scale, two slice size statistics are displayed: |BSlice| and |FSlice|, which show the sizes of the backward and forward slices for the vertices that form a coherent cluster. The sizes are shown as a percentage of the SDG's vertex count, with the separation of the vertical bars representing 10% increments. For example, Cluster 1's BSlice (1b) and FSlice (1c) include approximately 80% and 90% of the program's SDG vertices.

System View

Turning to *decluvi*'s three source-code views, the *System View* is at the highest level of abstraction. Each file containing executable source code is abstracted into a column. For bc this yields the nine columns seen in Figure 5.12. The name of the file appears at the top of each column, colour coded to reflect the size of the largest cluster found within the file. The vertical length of a column represents the length of the corresponding source file. To keep the view compact, each line of pixels in a column summarises multiple source lines. For moderate sized systems, such as the case studies considered herein, each pixel line represents about eight source code lines. The colour of each line reflects the largest cluster found among the summarised source lines, with light grey denoting source code that does not include any executable code. Finally, the numbers at the bottom of each column indicate the presence of the top 10 clusters in the file, where 1 denotes the largest cluster and 10 is the 10^{th} largest cluster. Although there may be other smaller clusters in a file, numbers are used to depict only the ten largest clusters because they are most likely to be of interest. In the case studies considered in Chapter 6, only the five largest coherent clusters are ever found to be interesting.

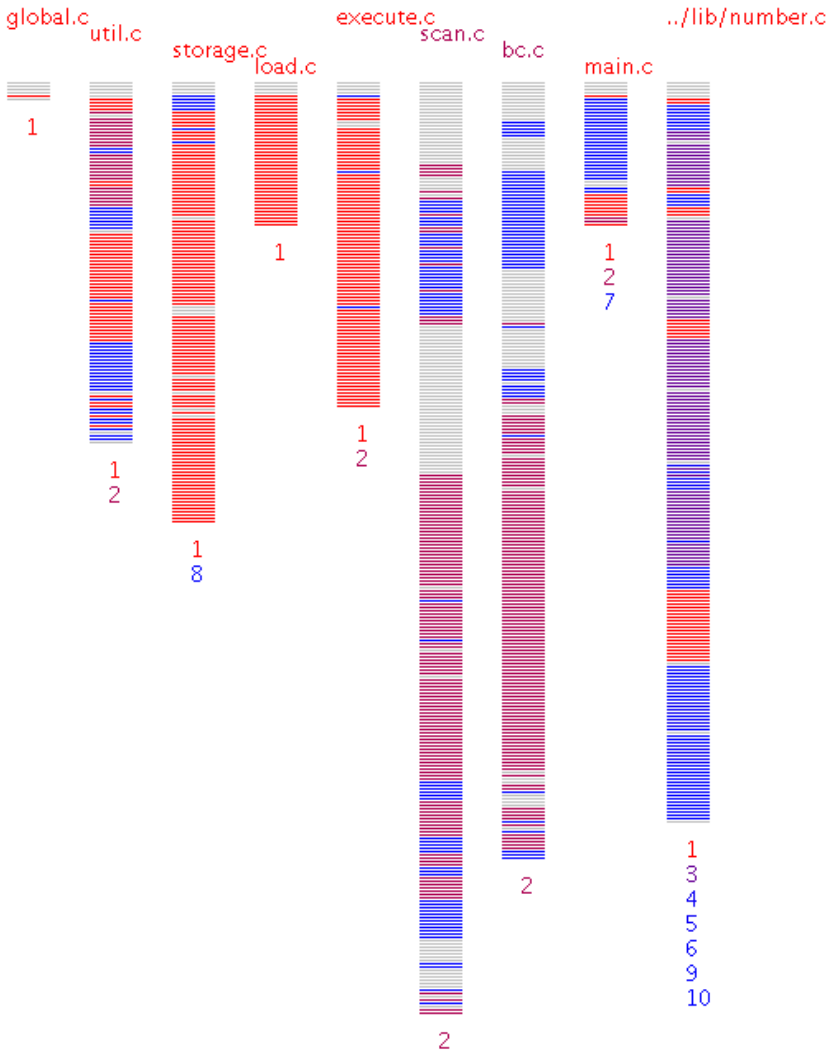


Figure 5.12: System View for the Program bc showing each file using one column and each line of pixels summarising *eight* source lines. Blue colour (medium grey in black & white) represent lines whose vertices are part of smaller size clusters than those in red colour (dark grey), while lines not containing any executable lines are always shown in light grey.

File View

The *File View*, illustrated in Figure 5.13, is at a lower level of abstraction than the System View. It essentially zooms in on a single column of the System View. In this view, each pixel line corresponds to one line of source code. The pixel lines are indented to mimic the indentation of the source lines and the number of pixels used to draw each line corresponds to the number of characters in the represented source code line. This makes it easier to relate this view to actual source code. The colour of a pixel line depicts the size of the largest coherent cluster formed by the SDG vertices from the corresponding source code line. Figure 5.13 shows the File View of bc's file util.c, filtered to show only the two largest coherent clusters, while smaller clusters and non-executable lines are shown in light grey.

Source View

While the first two views aid in locating parts of the system involved in one or more clusters, the *Source View* allows a programmer to see the actual source code lines that makes up each cluster. This can be useful in addressing questions such as *Why is a cluster formed? What binds a cluster together?* and *Is there unwanted/unnecessary dependence?* The *Source View*, illustrated in Figure 5.14, is a concrete view that maps the clusters onto actual source code lines. The lines are displayed in the same spatial context in which they were written, line colour depicts the size of the largest cluster to which the SDG vertices representing the line belong. Figure 5.14 shows Lines 241–257 of bc's file util.c, which has again been filtered to show only the two largest coherent clusters. The lines of code whose corresponding SDG vertices are part of the largest cluster are shown in red (dark grey) and those lines whose SDG vertices are part of the second largest cluster are shown in blue (medium grey). Other lines that do not include any executable code or whose SDG vertices are not part of the two largest clusters are shown in light grey. On the left of each line is a *line tag* with the format $a:b|c/d$, which presents the line number (a), the cluster number (b), and an identification c/d for the c^{th} of d clusters having a given size. For example, in Figure 5.14, Lines 250 and 253 are both part of a 20^{th} largest cluster (clusters with same size have the same rank) as indicated by the value of b; however they belong to different clusters as indicated by the differing values of c in their line tags.

Figure 5.13: File View for the file util.c of Program bc. Each line of pixels correspond to one source code line. Blue (medium grey in black & white) represents lines with vertices belonging to the 2^{nd} largest cluster and red (dark grey) represents lines with vertices belonging to the largest cluster. The rectangle marks function init_gen, part of both clusters.

Figure 5.14: Source View showing function init_gen in file util.c of Program bc. The *decluvi* options are set to filter out all but the two largest clusters thus blue (medium grey in black & white) represents lines from the 2^{nd} largest cluster and red (dark grey) lines from the largest cluster. All other lines including those with no executable code are shown in light grey.

5.6.3 Other features

Decluvi has features such as *filtering* and *relative colouring*. These features help to isolate and focus on a set of clusters of interest. Filtering allows a range of cluster sizes of interest to be defined. Only clusters whose size falls within the filtered range are shown using the Heat Map colours. Those outside the specified range along with non-executable lines of code are shown in light grey where in grayscale they appear in the lightest shade of grey. The filtering system incorporates a feature to *hide* non-executable lines of code as well as clusters whose size falls outside the specified range. In addition, relative colouring allows the Heat Map colours to be automatically adjusted to fit within a defined cluster size range. Relative colouring along with filtering overcomes the problem where clusters of similar sizes are represented using similar colours, making them indistinguishable.

5.6.4 Decluvi's Interface Evaluation

This subsection provides an evaluation of *decluvi*'s user interface against the list of features suggested by Shneiderman [1996].

Overview – *Gain an overview of the entire collection of data that is represented.*

The abstract Heat-Map View and compact System View provide an

overview of the clustering for an entire system.

Zoom – *Zoom in on items of interest.*

From the System View it is possible to zoom into individual files in either a lower level of abstraction (File View) or the concrete (Source View) form.

Filter – *Filter out uninteresting items.*

The control panel, shown in Figure 5.15, includes sliders and 'fast cluster selection' buttons. These allow a user to filter out uninteresting clusters and thus focus only on clusters of interest. The tool also provides option to hide non-executable lines and clusters whose size fall outside a specified range.

Details-on-demand – *Select an item or group and obtain details when needed.*

Although details for all items shown in the visualisation cannot be obtained, cluster related details are available. For example, clicking on a column of the System View opens the File View for the corresponding file and clicking on a line in the File View highlights the corresponding line in the Source View. Finally, the fast cluster selection buttons allow the user to demand and get details on a given cluster.

Relate – *Clear relationship between the various views.*

There is a hierarchical relationship between the various views provided by *decluvi*. Common colouring is used throughout to tie abstract elements of the higher level views with the concrete source lines in the Source View. In addition, File View and Source View preserve the layout of the underlying source code (e.g., the indentation of the lines).

History – *Keep history of actions to support undo, replay and progressive refinement.*

Decluvi currently meets this requirement partially. The various views of the tool retain their settings and viewing positions when toggled. However, current version of *decluvi* lacks support for undo, replay, or history.

Extract – *Allow extraction of sub-collections and of the query parameters.*

The tool provides support for exporting slice/cluster statistics.

Figure 5.15: *Decluvi* control panel

5.7 Related Work

This review does not attempt to survey the area of software visualisation but concentrates on the techniques which is the basis of the dependence cluster visualisation presented in this chapter.

The *Seesoft* System [Eick et al., 1992] is a seminal tool for visualising line oriented software statistics. The system pioneered the idea of abstracting source code view to represent each source code line using a line of pixels. This allowed for visualisation of up to 50,000 lines of code on a single screen. The rows were coloured to represent the values of statistics being visualised. The system pioneered four key ideas: reduced representation, colouring by statistic, direct manipulation, and capability to read actual code. The reduced representation was achieved by displaying files as columns and lines of code as thin rows. The system was originally envisioned to help in a lot of areas including program understanding. Ball and Eick [1994] also presented *SeeSlice*, a tool for interactive slicing. This was the first slicing visualisation system that allowed for a global overview of a program. Our visualisation inherits these approaches and extends them to be effective for dependence clusters.

The approach pioneered by *Seesoft* was also used in many other visualisation tools. The SeeSys System [Baker and Eick, 1995] was developed to localize error-prone code through visualisation of 'bug fix' statistics. The tool extended the Seesoft approach by introducing tree maps to show hierarchical data. It displayed code organised hierarchically into subsystems, directories, and files by representing the whole system as a rectangle and recursively representing the various sub-units with interior rectangles. The area of each rectangle was used to reflect a statistic associated with its sub-unit. Tarantula [Jones et al., May 2001] also employs the "line of pixel" style code view introduced by Seesoft. The tool was aimed at visualising the pass/fail of test cases. It extended the idea of using solid colours to represent statistics by using hue and brightness to encode additional information. CVSscan [Voinea et al., 2005] also inherited and extended the "line of pixel" based representation by introducing "dense pixel display" to show the overall evolution of programs. The tool has a bi-level code display that provide views of both the contents of a

code fragment and its evolution over time. Source Viewer 3D [Marcus et al., 2003] is a software visualisation framework that is based on Seesoft and adds a third dimension (3D) to the original approach allowing additional statistics to be visualised. Augur [Froehlich and Dourish, 2004] is also based on the line-oriented approach of Seesoft. The primary view is spatially organised as in Seesoft with additional columns to display multiple statistics for each line. Aspect Browser (Nebulous) [Yoshikiyo et al., 1999] provides a global view of how the various aspect entries cross-cut the source code using "line of pixels" view and uses Aspect Emacs to get the statistics and provide the concrete source code view. BLOOM [Reiss, 2001b] uses the BEE/HIVE [Reiss, 2001a] architecture, a powerful back-end that supports a variety of high-density, high-quality visualisation one of which (File Maps) is based on the Seesoft layout.

The final set of systems discussed are those that aim to help in reverse engineering but are not based on the "line of pixels" approach. Most of these tools focus on visualising high-level system abstractions (often referred to as 'clustering' or 'aggregation') such as classes, modules, and packages, using a graph-based approach. Rigi [Storey et al., 1997] is a reverse engineering tool that uses Simple Hierarchical Perspective (SHriMP) views, employs fisheye views of nested graphs. Creole[1] is an open-source plugin for the Eclipse (IDE) based on SHriMP. Tools such as GOOSE[2], Sotograph[3] and VizzAnalyzer [Panas et al., 2004] work on the class and method levels allowing information aggregation to form higher levels of abstractions. There are tools (Borland Together, Rational Rose, ESS-Model, BlueJ, Fujaba, GoVisual [Diehl, 2005]) which also help in reverse engineering by producing UML diagrams from source code.

5.8 Summary

This chapter firstly introduces the graph-based visualisations for dependence clusters and uses these visualisations to identify patterns of clustering among the subject programs. The study shows that most programs contain multiple large clusters making them worthy of further detailed study. Four subject programs showing interesting patterns are identified for further in-depth study.

This is followed by a study on the effect of varying the precision level of CodeSurfer's pointer analysis on size and frequency of coherent clusters. The

[1]http://www.thechiselgroup.org/creole
[2]http://esche.fzi.de/PROSTextern/ software/goose/index.html
[3]http:// www.software-tomography.com/html/sotograph.html

various levels of pointer analysis offer a trade-off between required resources and accuracy. The results show that the highest level of pointer analysis precision results in smaller slices and clusters, which is considered to be more precise. Furthermore, contrary to common intuition, it was found that using the lower pointer analysis settings often required larger SDG build times because of explosion in the number of summary edges that need to be calculated. The chapter finally discusses the multi-level cluster visualisation tool *decluvi*. The following chapter uses visualisations generated from *decluvi* to identify program structure and shows other applications of the cluster visualisation.

Chapter 6

Program Structure and Coherent Clusters

6.1 Overview

This chapter presents discussion of how coherent clusters map to the program structure. The chapter presents a series of four case studies of programs taken from the *large* and *huge* categories defined in Section 4.7. The case studies aim to reveal how coherent clusters in the programs compare against the logical structure identified by manual inspection of the systems. The manual examination is done by experts who have several years of software development and program analysis experience.

Following the case studies, a quantitative study of how coherent clusters and functions overlap. More formally, this chapter addresses the following research questions:

RQ3.1 Which structures within a program can coherent cluster analysis reveal?

RQ3.2 How do functions and clusters overlap, and do overlap and size correlate?

6.2 Coherent Cluster and program decomposition

This section presents four case studies acct, indent, bc and copia. The case studies form a major contribution of this thesis and collectively addresses research question *RQ3.1 Which structures within a program can coherent cluster anal-*

ysis reveal? The programs have been chosen to represent the *large* and *huge* groups identified in the previous chapter. Three programs are taken from the *large* group as majority of the programs fall in this category, and one from the *huge* group as it has only three programs. Each of the three programs from the *large* group were chosen because they exhibit interesting patterns. Acct has multiple coherent clusters visible in its profile and has the smallest large cluster in the group, bc has multiple large coherent clusters, and, copia has only a single large coherent cluster dominating the entire landscape.

It should be noted that the following case studies perform an in-depth qualitative study of the systems to identify mappings between coherent clusters and logical program structure. Although multiple smaller clusters were found in the systems, they were too small to map to logical structures of the programs and thus were ignored.

6.2.1 Case Study: acct

The first of the series of case studies is acct, an open-source program used for monitoring and printing statistics about users and processes. The program acct is one of the smaller programs with 2,600 LoC and 1,558 SLoC from which CodeSurfer produced 2,834 slices. The program has seven C files, two of which, getopt.c and getopt1.c, contain only conditionally included functions. These functions provide support for command-line argument processing and are included if needed library code is missing.

Table 6.1 shows the statistics for the five largest clusters of acct. Column 1 gives the cluster number, where 1 is the largest and 5 is the 5^{th} largest cluster measured using the number of vertices. Columns 2 and 3 show the size of the cluster as a percentage of the program's vertices and actual vertex count, as well as the line count. Columns 4 and 5 show the number of files and functions where the cluster is found. The cluster sizes range from 11.4% to 2.4%. These five clusters can be readily identified in the Heat-Map visualisation (Figure 6.1) of *decluvi*. The rest of the clusters are very small (less than 2% or 30 vertices) in size and are thus of little interest.

The B-SCG for acct (row one of Figure 5.5) shows the existence of these five coherent clusters along with other same-slice clusters. *Splitting* of the same-slice cluster is evident in the SCG. Splitting occurs when the vertices of a same-slice cluster become part of different coherent clusters. This happens when vertices have either the same backward slice or the same forward slice but *not*

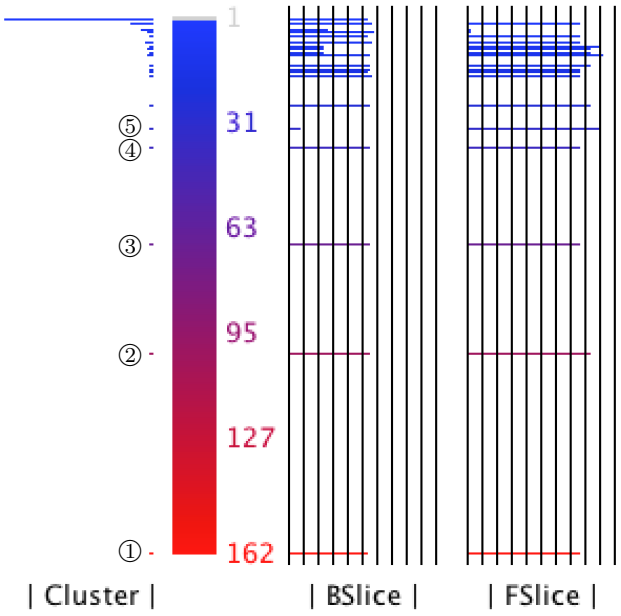


Figure 6.1: Heat Map View for acct

Cluster	Cluster Size		Files	Functions
	%	vertices/lines	spanned	spanned
1	11.4%	162/88	4	6
2	7.2%	102/56	1	2
3	4.9%	69/30	3	4
4	2.8%	40/23	2	3
5	2.4%	34/25	1	1

Table 6.1: acct's cluster statistics

both. This is because either same-backward-slice or same-forward-slice clusters only capture one of the two external properties captured by coherent clusters (Equation 4.1). In acct's B-SCG the vertices of the largest same-backward-slice cluster spanning the x-axis from 60% to 75% are not part of the same coherent cluster. This is because the vertices do not share the same forward slice which is also a requirement for coherent clusters. This phenomenon is common in the programs studied and is found in both same-backward-slice and same-forward-slice clusters. This is another reason why coherent clusters are often smaller in size then same-slice clusters.

Decluvi visualisation (not shown) of acct reveals that the largest cluster spans four files (file_rd.c, common.c, ac.c, and utmp_rd.c), the 2^{nd} largest cluster spans only a single file (hashtab.c), the 3^{rd} largest cluster spans three files

(file_rd.c, ac.c, and hashtab.c), the 4^{th} largest cluster spans two files (ac.c and hashtab.c), while the 5^{th} largest cluster includes parts of ac.c only.

The largest cluster of acct is spread over six functions, log_in, log_out, file_open, file_reader_get_entry, bad_utmp_record and utmp_get_entry. These functions are responsible for *putting accounting records into the hash table* used by the program, *accessing user-defined files*, and *reading entries* from the file. Thus, the purpose of the code in this cluster is to track user login and logout events.

The second largest cluster is spread over two functions hashtab_create and hashtab_resize. These functions are responsible for *creating fresh hash tables* and *resizing existing hash tables* when the number of entries becomes too large. The purpose of the code in this cluster is the memory management in support of the program's main data structure.

The third largest cluster is spread over four functions: hashtab_set_value, log_everyone_out, update_user_time, and hashtab_create. These functions are responsible for *setting values of an entry*, *updating all the statistics* for users, and *resetting the tables*. The purpose of the code from this cluster is the modification of the user accounting data.

The fourth cluster is spread over three functions: hashtab_delete, do_statistics, and hashtab_find. These functions are responsible for *removing entries* from the hash table, *printing out statistics* for users and *finding entries* in the hash table. The purpose of the code from this cluster is maintaining user accounting data and printing results.

The fifth cluster is contained within the function main. This cluster is formed due to the use of a while loop containing various cases based on input to the program. Because of the conservative nature of static analysis, all the code within the loop is part of the same cluster.

Finally, it is interesting to note that functions from the same file or with similar names do not necessarily belong to the same cluster. Intuitively, it can be presumed that functions that have similar names or prefixes work together to provide some common functionality. In this case, six functions that have the same common prefix "hashtab" all perform operations on the hash table. However, these six functions are not part of the same cluster. Instead the functions that work together to provide a particular functionality are found in the same cluster. The clusters help identify functionality which is not obvious

from the name of program artefacts such as functions and files. As an answer to *RQ3.1*, in this case study we find that each of the top five clusters map to specific logical functionality of the program.

6.2.2 Case Study: indent

Cluster	Cluster Size		Files	Functions
	%	vertices/lines	spanned	spanned
1	52.1%	3930/2546	7	54
2	3.0%	223/136	3	7
3	1.9%	144/72	1	6
4	1.3%	101/54	1	5
5	1.1%	83/58	1	1

Table 6.2: indent's cluster statistics

The next case study uses indent to further support the answer found for *RQ3.1* in the acct case study. The characteristics of indent are very different from those of acct as indent has a very large dominant coherent cluster (52%) whereas acct has multiple smaller clusters with the largest being 11%. We include indent as a case study to ensure that the answer for *RQ3.1* is derived from programs with different cluster profiles and sizes giving confidence as to the generality of the answer.

Indent is a Unix utility used to format C source code. It consists of 6,978 LoC with 7,543 vertices in the SDG produced by CodeSurfer. Table 6.2 shows statistics of the five largest clusters found in the program. The BSCG for indent is shown in the first row of Figure 5.6.

Indent has one extremely large coherent cluster that spans 52.1% of the program's vertices. The cluster is formed of vertices from 54 functions spread over 7 source files. This cluster captures most of the logical functionalities of the program. Out of the 54 functions, 26 begin with the common prefix of "handle_token". These 26 functions are individually responsible for handling a specific token during the formatting process. For example, handle_token_colon, handle_token_comma, handle_token_comment, and handle_token_lbrace are responsible for handling the colon, comma, comment, and left brace tokens, respectively.

This cluster also includes multiple handler functions that check the size of the code and labels being handled, such as check_code_size and check_lab_size. Others, such as search_brace, sw_buffer, print_comment, and reduce, help with

tracking braces and comments in code. The cluster also spans the main loop of indent (indent_main_loop) that repeatedly calls the parser function parse.

Finally, the cluster consists of code for outputting formatted lines such as the functions better_break, computer_code_target, dump_line, dump_line_code, dump_line_label, inhibit_indenting, is_comment_start, output_line_length and slip_horiz_space, and ones that perform flag and memory management (clear_buf_break_list, fill_buffer and set_priority).

Cluster 1 therefore consists of the main functionality of this program and provides support for *parsing, handling tokens, associated memory management*, and *output*. The parsing, handling of individual tokens and associated memory management are highly inter-twined. For example, the handling of each individual token is dictated by operations of indent and closely depends on the parsing. This code cannot easily be decoupled and, for example, reused. Similarly the memory management code is specific to the data structures used by indent resulting in these many logical constructs to become part of the same cluster.

The second largest coherent cluster consists of 7 functions from 3 source files. These functions handle the arguments and parameters passed to indent. For example, set_option and option_prefix along with the helper function eqin to check and verify that the options or parameters passed to indent are valid. When options are specified without the required arguments, the function arg_missing produces an error message by invoking usage followed by a call to DieError to terminate the program.

Cluster 3, 4 and 5 are less than 3% of the program and are too small to warrant a detailed discussion. Cluster 3 includes 6 functions that generate numbered/un-numbered backup for subject files. Cluster 4 has functions for reading and ignoring comments. Cluster 5 consists of a single function that reinitialises the parser and associated data structures.

The case study of indent further illustrates that coherent clusters can capture the program's logical structure as an answer to research question *RQ3.1*. However, in cases such as this where the internal functionality is tightly knit, a single large coherent cluster maps to the program's core functionality.

6.2.3 Case Study: bc

The third case study in this series is bc, an open-source calculator, which consists of 9,438 LoC and 5,450 SLoC. The program has nine C files from

which CodeSurfer produced 15,076 slices (backward and forward).

Analysing bc's SCG (row 4, Figure 5.5), two interesting observations can be made. First, bc contains two large same-backward-slice clusters visible in the light grey landscapes as opposed to the three large coherent clusters. Second, looking at the B-SCG, it can be seen that the x-axis range spanned by the largest same-backward-slice cluster is occupied by the top two coherent clusters shown in the dashed red (dark grey) landscape. This indicates that the same-backward-slice cluster splits into the two coherent clusters.

The statistics for bc's top five clusters are given in Table 6.3. Sizes of these five clusters range from 32.3% through to 1.4% of the program. Clusters six onwards are less than 1% of the program. The Project View (Figure 5.12) shows their distribution over the source files.

Cluster	Cluster Size		Files	Functions
	%	vertices/lines	spanned	spanned
1	32.3%	2432/1411	7	54
2	22.0%	1655/999	5	23
3	13.3%	1003/447	1	15
4	1.6%	117/49	1	2
5	1.4%	102/44	1	1

Table 6.3: bc's cluster statistics

In more detail, Cluster 1 spans all of bc's files except for scan.c and bc.c. This cluster encompasses the core functionality of the program – *loading and handling of equations, converting to* bc's *own number format, performing calculations*, and *accumulating results.* Cluster 2 spans five files, util.c, execute.c, main.c, scan.c, and bc.c. The majority of the cluster is distributed over the latter two files. Even more interestingly, the source code of these two files (scan.c and bc.c) map only to cluster 2 and none of the other top five clusters. This indicates a clear purpose to the code in these files. These two files are solely used for *lexical analysis* and *parsing* of equations. To aid in this task, some utility functions from util.c are employed. Only five lines of code in execute.c are also part of Cluster 2 and are used for *flushing output* and *clearing interrupt signals.* The third cluster is completely contained within the file number.c. It encompasses functions such as _bc_do_sub, _bc_init_num, _bc_do_compare, _bc_do_add, _bc_simp_mul, _bc_shift_addsub, and _bc_rm_leading_zeros, which are responsible for *initialising* bc's *number formatter, performing com-*

parisons, modulo and other *arithmetic operations.* Clusters 4 and 5 are also completely contained within number.c. These clusters encompass functions to perform *bcd operations for base ten numbers* and *arithmetic division*, respectively.

As an answer to *RQ3.1*, the results of the cluster visualisations for bc reveal its high-level structure. This aids an engineer in understanding how the artefacts (e.g., functions and files) of the program interact, thus aiding in program comprehension. The remainder of this subsection illustrates a side-effect of *decluvi*'s multi-level visualisation, how it can help find potential problems with the structure of a program aiding in program maintenance.

Util.c consists of small utility functions called from various parts of the program. This file contains code from Clusters 1 and 2 (Figure 5.12). Five of the utility functions belong with Cluster 1, while six belong with Cluster 2. Furthermore, Figure 5.13 shows that the distribution of the two clusters in red (dark grey) and blue (medium grey) within the file are well separated. Both clusters do not occur together inside any function with the exception of init_gen (highlighted by the rectangle in first column of Figure 5.13). The other functions of util.c thus belong to either Cluster 1 or Cluster 2. Separating these utility functions into two separate source files where each file is dedicated to functions belonging to a single cluster would improve the code's logical separation and file-level cohesion. This would make the code easier to understand and maintain at the expense of a very simple refactoring. In general, this example illustrates how *decluvi* visualisation can provide an indicator of potential points of code degradation during evolution.

Finally, the Code View for function init_gen shown in Figure 5.14 includes Lines 244, 251, 254, and 255 in red (dark grey) from Cluster 1 and Lines 247, 248, 249, and 256 in blue (medium grey) from Cluster 2. Other lines, shown in light grey, belong to smaller clusters and lines containing no executable code. Ideally, clusters should capture a particular functionality; thus, functions should generally not contain code from multiple clusters (unless perhaps the clusters are completely contained within the function). Functions with code from multiple clusters reduce code separation (hindering comprehension) and increase the likelihood of ripple-effects [Black, 2001]. Like other initialisation functions, bc's init_gen is an exception to this guideline.

This case study not only provides support for the answer to research ques-

Cluster number	Cluster Size %	Cluster Size vertices/lines	Files spanned	Functions spanned
1	48%	1609/882	1	239
2	0.1%	4/2	1	1
3	0.1%	4/2	1	1
4	0.1%	4/2	1	1
5	0.1%	2/1	1	1

Table 6.4: copia's cluster statistics

tion *RQ3.1* found in previous case studies, but also illustrates that the visualisation is able to reveal structural defects in programs.

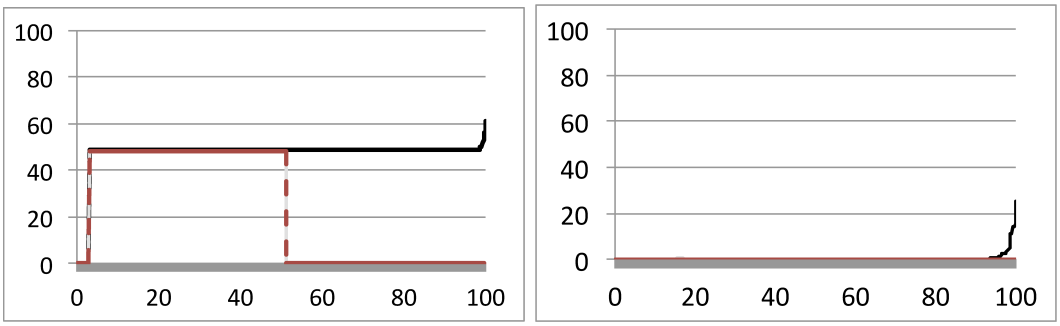
6.2.4 Case Study: copia

The final case study in this series is copia, an industrial program used by the ESA to perform signal processing. Copia is the smallest program considered in this series of case studies with 1,168 LoC and 1,111 SLoC all in a single C file. Its largest coherent cluster covers 48% of the program. The program is at the top of the group with large coherent clusters. CodeSurfer extracts 6,654 slices.

The B-SCG for copia is shown in Figure 6.2a. The single large coherent cluster spanning 48% of the program is shown by the dashed red (dark grey) line (running approx. from 2% to 50% on the x-axis). The plots for same-backward-slice cluster sizes (light grey line) and the coherent cluster sizes (dashed line) are identical. This is because the size of the coherent clusters are restricted by the size of the same-backward-slice clusters. Although the plot for the size of the backward slices (black line) seems to be the same from the 10% mark to 95% mark on the x-axis, the slices are not exactly the same. Only vertices plotted from 2% through to 50% have exactly same backward and forward slice resulting in the large coherent cluster.

Table 6.4 shows statistics for the top five coherent clusters found in copia. Other than the largest cluster which covers 48% of the program, the rest of the clusters are extremely small. Clusters 2–5 include no more than 0.1% of the program (four vertices) rendering them too small to be of interest. This suggests a program with a single functionality or structure.

During analysis of copia using *decluvi*, the File View (Figure 6.3) reveals an intriguing structure. There is a large block of code with the same spatial arrangement (bounded by the dotted black rectangle in Figure 6.3) that belongs to the largest cluster of the program. It is unusual for so many consecutive



(a) *Original* (b) *Modified*

Figure 6.2: SCGs for the program copia

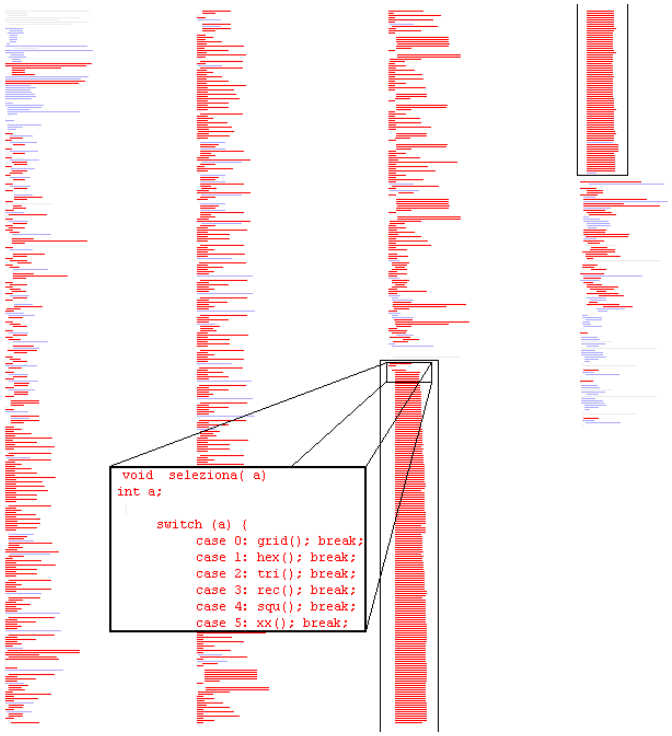


Figure 6.3: *Decluvi*'s File View for the file copia.c of Program copia. Each line of pixels represent the cluster data for one source code line. The lines in red (dark grey in black & white) are part of the largest cluster. The lines in blue (medium grey) are part of smaller clusters. A rectangle highlights the switch statement that holds the largest cluster together.

source code lines to have nearly identical length and indentation. Inspection of the source code reveals that this block of code is a switch statement handling 234 cases. Further investigation shows that copia has 234 small functions that eventually call one large function, seleziona, which in turn calls the smaller functions effectively implementing a finite state machine. Each of the smaller functions return a value that is the next state for the machine and is used by the switch statement to call the appropriate next function. The primary reason for the high level of dependence in the program lies with the statement switch(next_state), which controls the calls to the smaller functions. This causes what might be termed 'conservative dependence analysis collateral damage' because the static analysis cannot determine that when function f() returns the constant value 5 this leads the switch statement to eventually invoke function g(). Instead, the analysis makes the conservative assumption that a call to f() might be followed by a call to any of the functions called in the switch statement, resulting in a mutual recursion involving most of the program.

Although the coherent cluster still shows the structure of the program and includes all these stub functions that work together, this is a clear case of dependence pollution [Binkley and Harman, 2005b], which is avoidable. To illustrate this, the code was re-factored to simulate the replacement of the integer next_state with direct recursive function calls. The SCG for the modified version of copia is shown in Figure 6.2b where the large cluster has clearly disappeared. As a result of this reduction, the potential impact of changes to the program will be greatly reduced, making it easier to understand and maintain. This is even further amplified for automatic static analysis tools such as CodeSurfer. Of course, in order to do a proper re-factoring, the programmer will have to consider ways in which the program can be re-written to change the flow of control. Whether such a re-factoring is deemed cost-effective is a decision that can only be taken by the engineers and managers responsible for maintaining the program in question.

This case study reiterates the answer for *RQ3.1* by showing the structure and dependency within the program. It also identifies potential refactoring points which can improve the performance of static analysis tools and make the program easier to understand.

6.3 Cluster and Function Mapping

The four case studies of this chapter provide a qualitative analysis of the mapping between clusters and functions, and show that dependence clusters do not simply correspond to functional decomposition of programs but rather map to higher-level logical structures. This section presents a quantitative study of this mapping and addresses research question *RQ3.2 How do functions and clusters overlap, and do overlap and size correlate?* For the purpose of this discussion let C be the set of all coherent clusters, F the set of all functions, *cLargest* the largest coherent cluster and *fLargest* the largest function, and $\mathsf{Overlap}(c, f)$ the intersection $V(c) \cap V(f)$, where $V(c)$ represents the set of vertices in cluster c and $V(f)$ represents the set of vertices in function f.

Figures 6.4 and 6.5 show graphs comparing coherent cluster size to the number of functions that the cluster overlaps. In the graphs, the x-axis shows the function count while the y-axis shows the cluster size as a percentage of the program. All the graphs show the same trend, the largest cluster overlaps more functions than any other individual cluster. That is, $\forall_{c \in C} |\{f \in F : \mathsf{Overlap}(c, f) \neq \emptyset\}| \leq |\{f \in F : \mathsf{Overlap}(cLargest, f) \neq \emptyset\}|$.

Figures 6.6 and 6.7 show graphs comparing function size to the number of clusters that overlap the function. Intuitively, larger functions should have more overlapping clusters. The x-axis shows a count of the clusters overlapped, while the y-axis shows the size of the function as a percentage of program's size. Unlike the graphs in Figures 6.4 and 6.5, the trend of all the graphs in Figures 6.6 and 6.7 do not agree. Here, acct is an exception. Its largest function, main, overlaps more clusters than any other function in the program. In all the remaining programs, the largest function does not have the maximum number of clusters overlapping it. Still, overall it is not the case that the largest function has the most overlapping clusters (i.e., $\forall_{f \in F} |\{c \in C : \mathsf{Overlap}(c, f) \neq \emptyset\}| \leq |\{c \in C : \mathsf{Overlap}(c, fLargest) \neq \emptyset\}|$).

As an answer to *RQ3.2*, the largest coherent cluster always overlaps more functions than the other clusters. However, with the exception of acct, the largest function is never overlapped by the highest number of clusters. The study therefore finds support for the consistent overlap of the largest cluster with the most functions, but not vice versa. This is also reflected in the case studies where we found the largest five clusters in programs to be of importance.

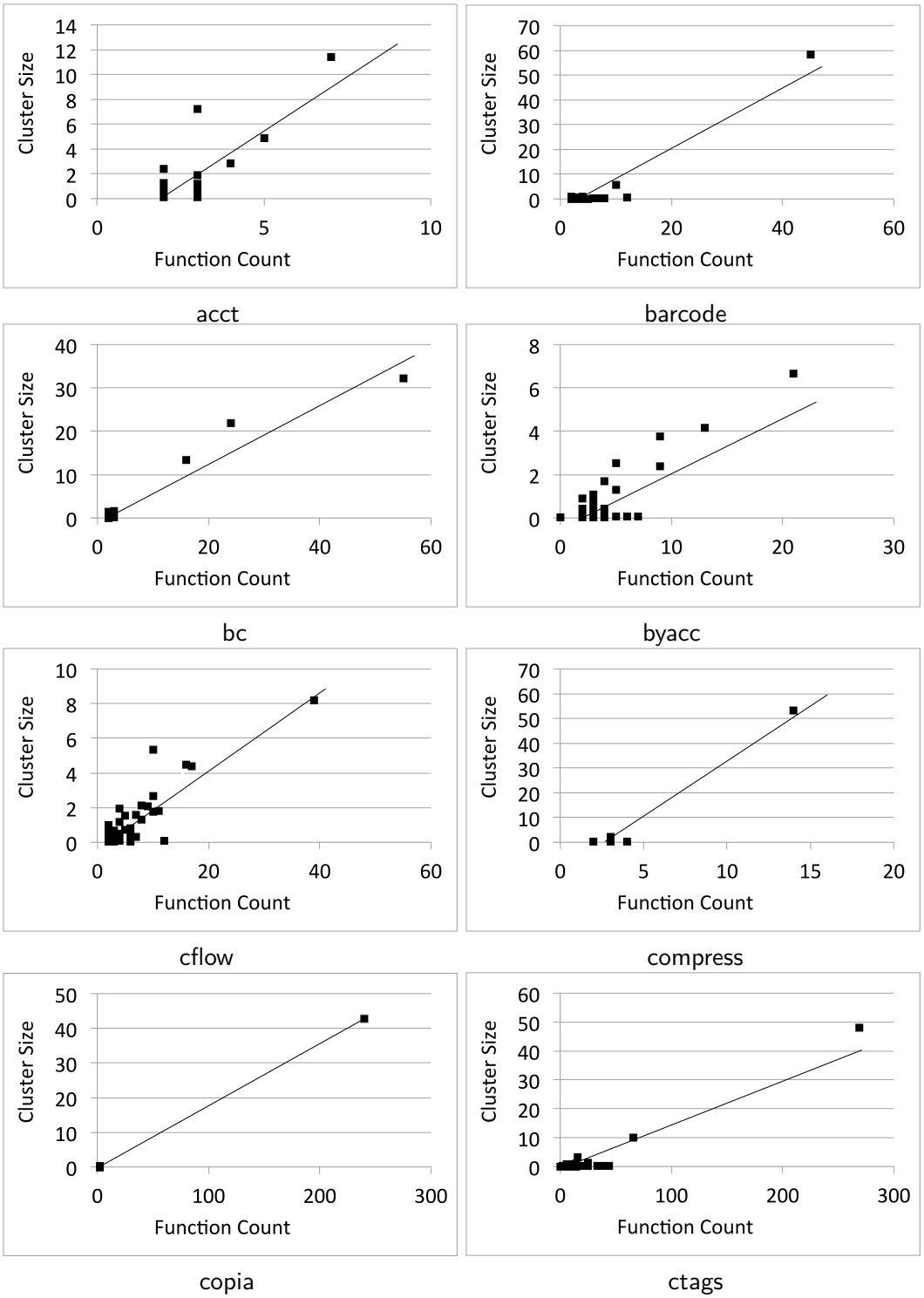


Figure 6.4: Cluster size vs. function count analysis

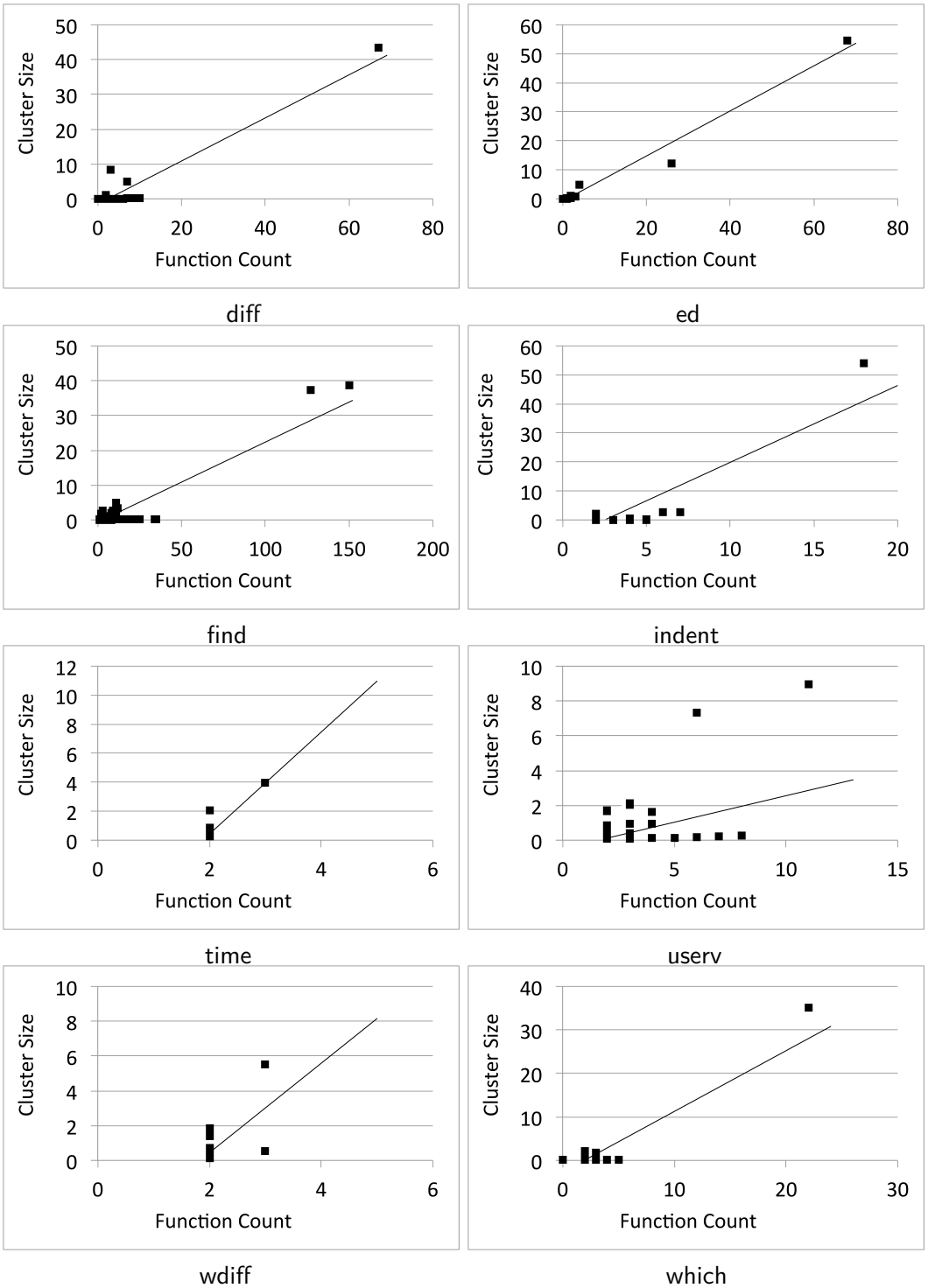


Figure 6.5: Cluster size vs. function count analysis

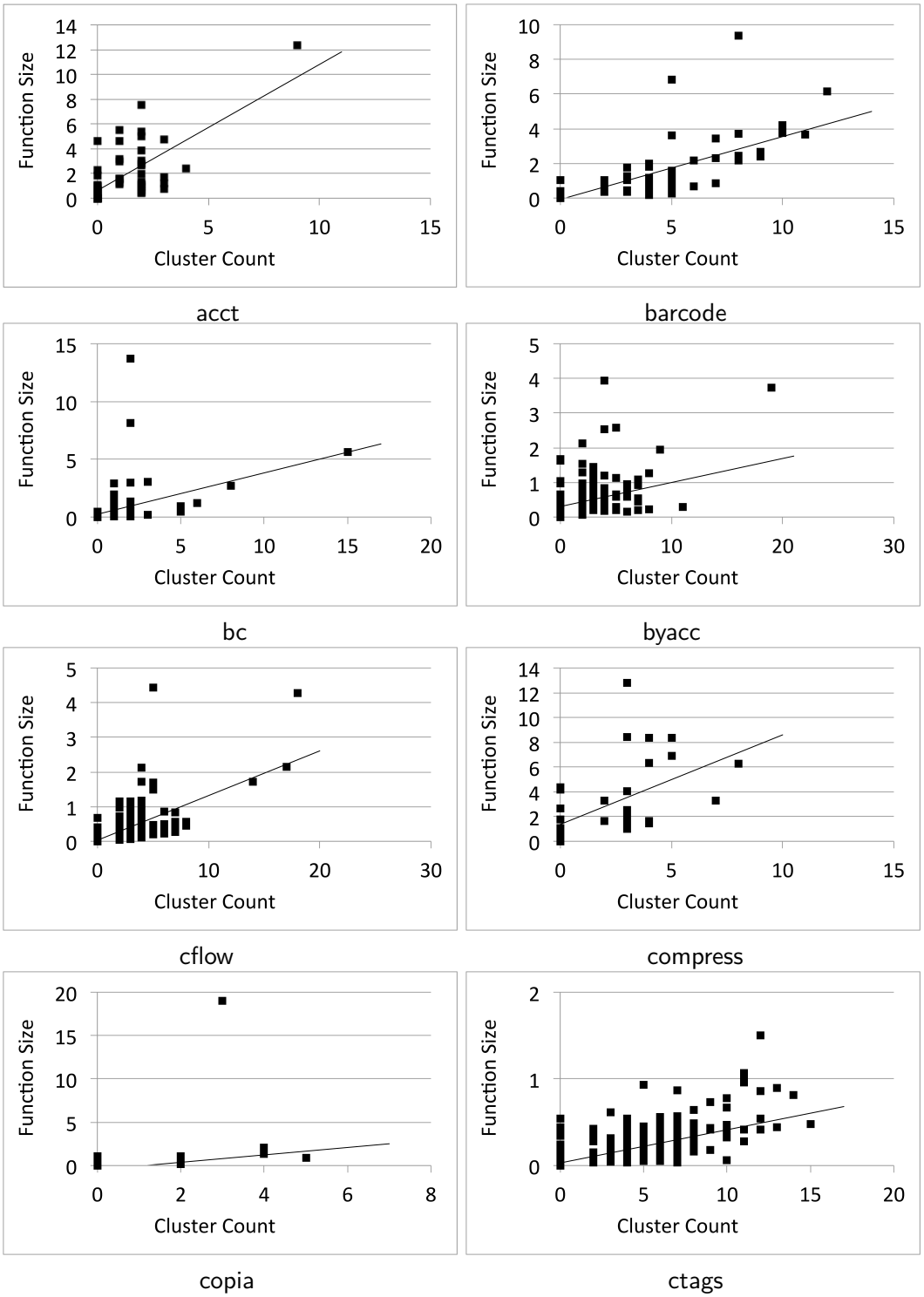


Figure 6.6: Function size vs. cluster count analysis

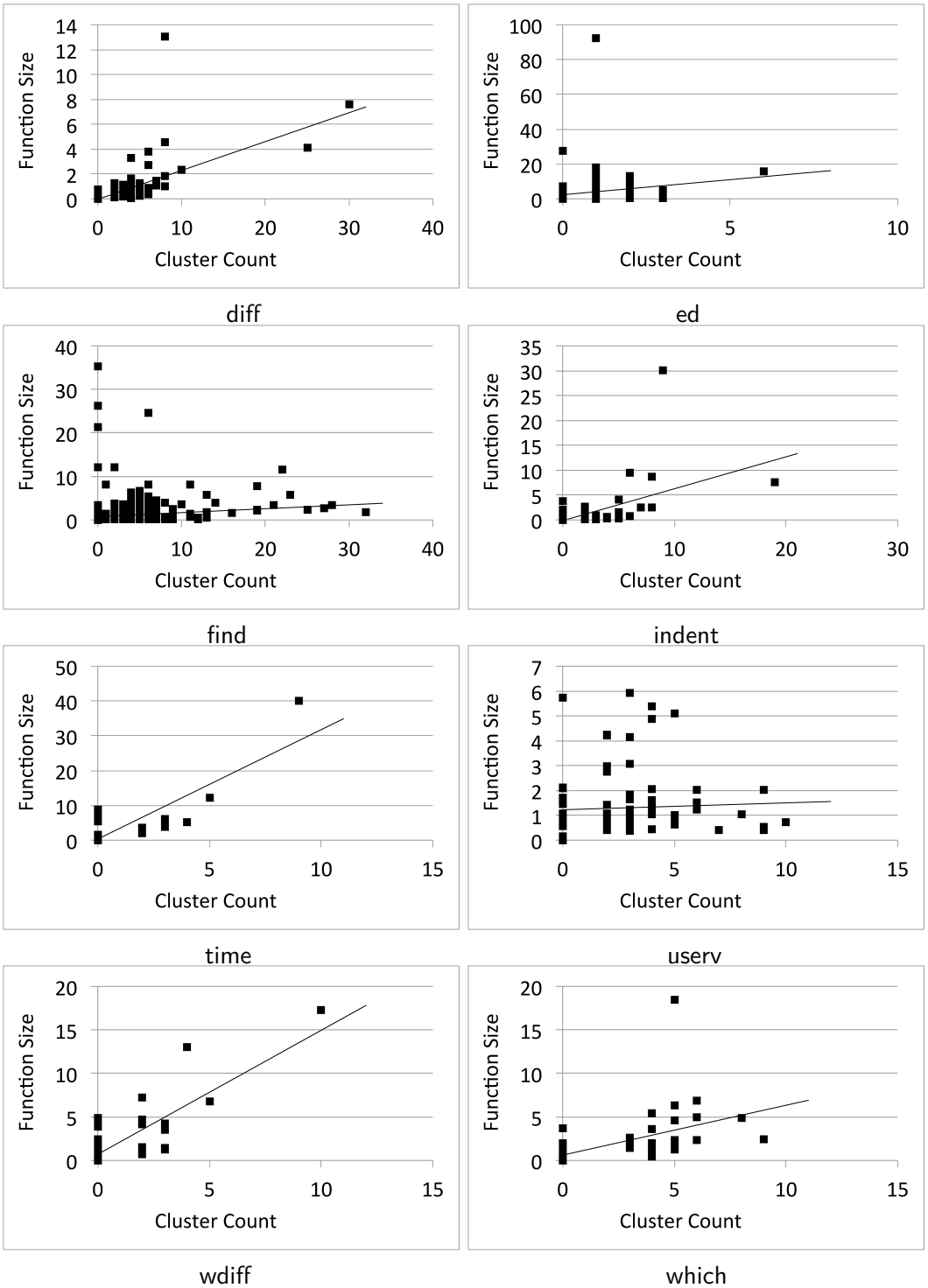


Figure 6.7: Function size vs. cluster count analysis

6.4 Related Work

Automatic feature location is an area that has seen significant research [Dit et al., 2011]. There are interactive approaches where carefully designed test cases are used to probe code to locate features [Wilde et al., 1992, Wilde and Scully, 1995]. Semi-automated approaches also exist that helps build a 'mental map' and locate features in code by identifying observable features triggered by user [Eisenbarth et al., 2003]. There are automatic approaches for inferring features from code, for example by studying transcripts of programmer interaction during development and debugging [Robillard and Murphy, 2003], and, text-based independent component analysis [Grant et al., 2008]. Similarly we use coherent clusters to highlight features and logical functions in programs.

Binkley et al. [2008a] study the relationship between concepts expressed in source code and dependence. Their work found that for domain-level concepts expressed in programs, it is likely that such concepts would have a degree of semantic cohesion. This cohesion is often represented as dependence between statements all of which contribute to the computation of the same concept. They also find that code associated with concepts has a greater degree of coherence, with tighter dependence. Li [2009] proposed a framework for combining low-level program slicing and high-level concepts assignments and defined metrics for evaluating concept extension, concept abbreviation and concept refinement.

Recent work by Hamilton and Danicic [2012] applies community detection algorithms to identify communities within software systems. They adapt algorithms from social networks and show that software systems exhibit community structure where the communities represent software components. They apply the Louvian Method [Blondel et al., 2008] to software dependency graphs of slice inclusion relationship. Although their approach works on small example programs, it has not been evaluated using large industrial programs. Also, they use Gephi [Bastian et al., 2009] for visualisation of the community structures detected. In our experience Gephi fails to visualise graphs for even medium sized programs and puts functional limit on the size of the graphs that may be visualised (50K nodes and 500K edges).

6.5 Chapter Summary

This chapter forms a major contribution of the thesis. It shows that coherent clusters in programs map to logical program constructs and can be used to gain an understanding of the program architecture. The chapter demonstrates these through a series of four case studies which were chosen to represent various interesting clustering patterns identified in Section 5.3.

All previous work on dependence clusters have emphasised on dependence clusters being problems and encourages techniques for identifying their causes for the purpose of reduction or removal. Instead, this chapter highlights that coherent clusters occur naturally in programs and their visualisation can help in program comprehension. As such, we are providing tools for developers to identify dependence clusters in their system and to help with their comprehension.

The detailed study of the cluster visualisation for the four case studies also show other positive side-effects. For example, in bc we identified a potential refactoring opportunity which will lead to better code structure, and, in Copia we identified a large unwanted dependence structure which can also be significantly reduced by refactoring.

Chapter 7

Other applications of Coherent Clusters

7.1 Overview

Having demonstrated that coherent clusters are indeed prevalent and that coherent clusters in programs map to logical constructs, this chapter presents a few segregated studies into the properties of coherent clusters. It firstly considers a study of coherent clusters and program bug fixes to establish a (lack of) link between the two. It then presents further studies into the changes in coherent clusters during system evolution. The chapter then presents the notion of inter-cluster dependency and Cluster Dependency Graphs (CDG). Finally, the chapter studies the existence of coherent cluster in object-oriented paradigm by considering Java programs. More formally, this chapter addresses the following research questions:

RQ4.1 How do program faults relate to coherent clusters?

RQ4.2 How stable are coherent clusters during system evolution?

RQ4.3 What are the implications of inter-cluster dependence between coherent clusters?

RQ4.4 Are coherent clusters prevalent in object-oriented programs?

7.2 Dependence Clusters and Bug Fixes

Initial work on dependence clusters advised that they might cause problems in software maintenance, and thus even be considered harmful because they

represent an intricate interweaving of mutual dependencies between program elements. Thus a large dependence cluster might be thought of as a bad code smell [Elssamadisy and Schalliol, 2002] or an anti-pattern [Binkley et al., 2008b]. Black et al. [2006] suggest that dependence clusters are potentially where bugs may be located and also suggest the possibility of a link between clusters and program faults. This section further investigates this issue using a study that explores the relationship between program faults and dependence clusters. In doing so, it addresses research question *RQ4.1 How program faults relate to coherent clusters?*

Barcode, an open source utility tool for converting text strings to printed bars (barcodes) is used in this study. A series of versions of the system are available for download from GNU repository[1]. There are nine public releases for barcode, details of which are shown in Table 7.1. Column 1 and column 2 shows the release version and date, Columns 3–6 show various metrics for the size of the system in terms of number of source files and various source code size measures. Columns 7–9 report the number of SDG vertices, SDG edges and the number of slices produced for each release. Finally, Column 10 reports the number of faults that were fixed since the previous release of the system. In Table 7.1 the size of barcode increases from 1,352 lines of code in version 0.90 to 3,968 lines of code in version 0.98. The total number of faults that were fixed during this evolution was 39.

Version	Release Date	C Files	LoC	SLoC	ELoC	SDG Vertices	SDG Edges	Total Slices	Faults Fixed
0.90	29-06-1999	6	1,352	891	716	7,184	23,072	3,148	-
0.91	08-07-1999	6	1,766	1,186	949	8,703	30,377	5,328	5
0.92	03-09-1999	8	2,225	1,513	1,221	10,481	37,373	5,368	9
0.93	26-10-1999	8	2,318	1,593	1,284	11,415	42,199	5,722	5
0.94	26-10-1999	8	2,318	1,593	1,284	11,414	41,995	5,722	1
0.95	03-02-2000	8	2,585	1,785	1,450	12,202	45,830	6,514	3
0.96	09-11-2000	11	3,249	2,226	1,799	14,733	56,802	8,106	9
0.97	17-10-2001	13	3,911	2,670	2,162	16,602	64,867	9,530	2
0.98	03-03-2002	13	3,968	2,685	2,177	16,721	65,395	9,602	5

Table 7.1: Fault fixes for barcode

Fault data gathered by manually analyzing the publicly available version control repository[2] for the system shows that total number of commits for barcode during these releases were 137. Each update was manually checked using CVSAnaly [Robles et al., 2004] to determine whether the update was a

[1] `http://gnu.mirror.iweb.com/gnu/barcode/`

[2] `cvs.savannah.gnu.org:/sources/barcode`

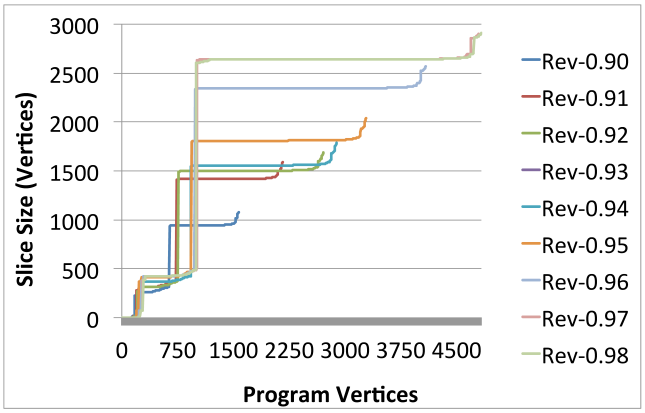


Figure 7.1: Backward slice sizes for barcode releases

bug fix or simply an enhancement or upgrade to the system. Those commits that were identified as bug fixes were isolated and mapped to the release that contained the update. All the bug fixes made during a certain release cycle were then accumulated to give the total number of bugs fixed during a particular release cycle (Column 10 of Table 7.1). The reported number only includes bug fixes and does not include enhancement or addition of new functionality.

Figure 7.1 shows the backward slice size plots for all versions of barcode in a single graph. The values of the axises in Figure 7.1 are shown as vertex counts rather than relative values (percentages) as done with the MSG. This allows the growth of barcode to be easily visualised. From the plots it is seen that the size of the program increases progressively with each new release. The graphs also show that a significant number of vertices in each revision of the program yields identical backward slices and the proportion of vertices in the program that have identical backward slices stays roughly the same. Overall, the profile of the clusters and slices remains consistent. The graph also shows that the plots don't show any significant change in their overall shape or structure. Interestingly, the plot for version 0.92 with 9 fault fixes is not different in shape from revision 0.94 where only a single fault was fixed.

As coherent clusters are composed of both backward and forward slices, the stability of the backward slice profile itself does not guarantee the stability of coherent cluster profile. The remainder of this section looks at how the clustering profile is affected by bug fixes. Figure 7.2 shows individual SCGs for each version of barcode. As coherent clusters are dependent on both backward and forward slices, such clusters will be more sensitive to changes in dependences

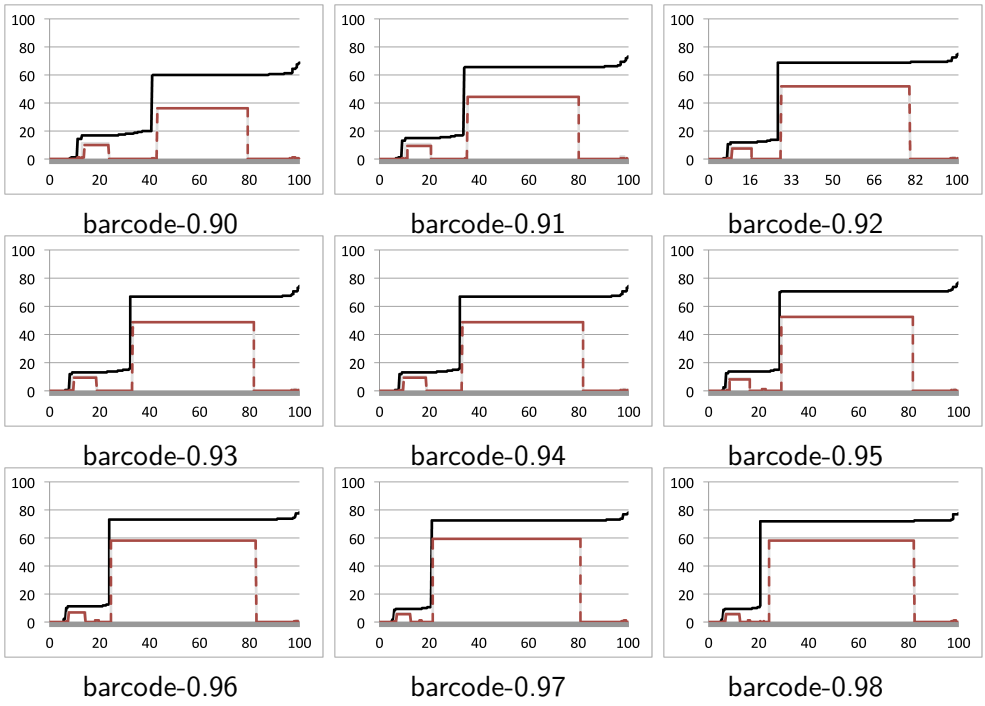


Figure 7.2: BSCGs for various barcode versions

within the program. The SCGs show that in the initial version barcode-0.90 there were two coherent clusters in the system. The smaller one is around 10% of the code while the larger is around 40% of the code. As the system evolved and went through various modifications and enhancements, the number of clusters and the profile of the clusters remained consistent other than its scaled growth with the increase in program size. It is also evident that during evolution of the system, the enhancement code or newly added code formed part of the larger cluster. This is why in the later stages of the evolution we see an increase in the size of the largest cluster, but not the smaller one.

However, we do not see any significant changes in the slice and cluster profile of the program that can be attributed to bug fixes. For example, the single bug fixed between revisions 0.93 and 0.94 was on a single line of code from the file code128.c. The changes to the line is shown in Figure 7.3 (in version 0.93 there is an error in calculating the checksum value, which was corrected in version 0.94). As illustrated by this example, the data and control flow of the program and thus the dependencies between program points are not affected by the bug fix and hence no change is observed between the SCGs of the two releases (Figure 7.2).

barcode-0.93, code128.c, line 139
```
checksum += code * i+1;
```

barcode-0.94, code128.c, line 139
```
checksum += code *(i+1);
```

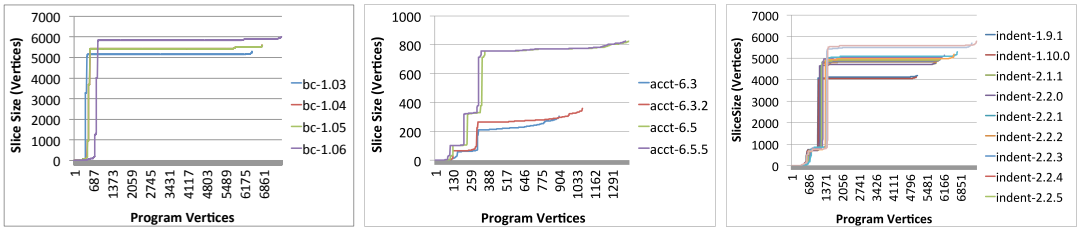
Figure 7.3: Bug fix example

If dependence clusters correlated to faults, or, if dependence clusters were directly related to the number of faults in a program, then a significant difference would be expected in the shape of the SCG when faults were rectified. The SCGs for program barcode (Figure 7.2) show no change in their profile when faults within the program are fixed. This provides evidence that faults may not be dictated by the presence or absence of dependence clusters. As an answer to *RQ4.1*, the study of barcode finds no correlation between the existence of dependence clusters and program faults and their fix. We have to be careful in generalising the answer to this question because of the small dataset considered in this study, further extended research is needed to derive a more generalised answer. Moreover, this does not exclude the possibility that most program faults occur in code that are part of large clusters. In future we plan to extend this experiment in a qualitative form to study whether program faults lie within large or small clusters, or outside them altogether.

7.3 Dependence Clusters and System Evolution

The previous section showed that for barcode the slice and cluster profiles remain quite stable (through bug fixes) during system evolution and the system's growth of almost 2.5 times over a period of 3 years. This chapter extends that study by looking for cluster changes during system evolution. It addresses *RQ4.2 How stable are coherent clusters during system evolution?* using longitudinal analysis of the case studies presented earlier. From the GNU repository we were able to retrieve four releases for bc, four releases for acct and 14 releases for indent. As copia is an industrial program, we were unable to obtain any previous versions of the program and thus the program is excluded from this study.

The graphs in Figure 7.4 show backward slice size overlays for every version of each program. Figure 7.4a and Figure 7.4c for bc and indent show that these



(a) bc (b) acct (c) indent

Figure 7.4: Backward slice size plots for multiple releases

systems grow in size during its evolution. The growth is more prominent in indent (Figure 7.4c) where the program grows from around 4800 vertices in its initial version to around 7000 vertices in the final version. The growth for bc is smaller, it grows from around 6000 vertices to 7000 vertices. This is partly because the versions considered for bc are all minor revisions. For both bc and indent the slice-size graphs show very little change in their profile. The graphs mainly show a scale up that parallels the growth of the system.

For acct (Figure 7.4b) the plots do not simply show a scale up but show a significant difference. In the 4 plots, the revisions that belong to the same major release are seen to be similar and show a scaled growth, whereas those from different major releases show very different landscapes. The remainder of this section gives detail of these clustering profile changes.

Figure 7.5 shows the BSCGs for the four versions of bc. Initially, the backward slice size plots (solid black lines) show very little difference. However, upon closer inspection of the last three versions we see that the backward slice size plot changes slightly at around the 80% mark on the x-axis. This is highlighted by the fact that the later three versions show an additional coherent cluster spanning from 85%–100% on the x-axis which is absent from the initial release. Upon inspection of the source code changes between versions bc-1.03 and bc-1.04 the following types of updates were found:

1. bug fixes
2. addition of command line options
3. reorganisation of the source tree
4. addition of new commands for dc

The reorganisation of the program involved significant architectural changes that separated out the code supporting bc's related dc functionality into a

separate hierarchy and moved files common to both bc and dc to a library. This refactoring of the code broke up the largest cluster into two clusters, where a new third cluster is formed as seen in the SCG. Thus, the major restructuring of the code between revisions 1.03 and 1.04 causes a significant change in the cluster profile with the addition of a new significant cluster. This supports the answer for *RQ3.1* by showing that the separation of the program logic leads to separation of the clusters. Almost no other change is seen in the cluster profile between the remaining three bc revisions 1.04, 1.05, and 1.06, where no significant restructuring took place.

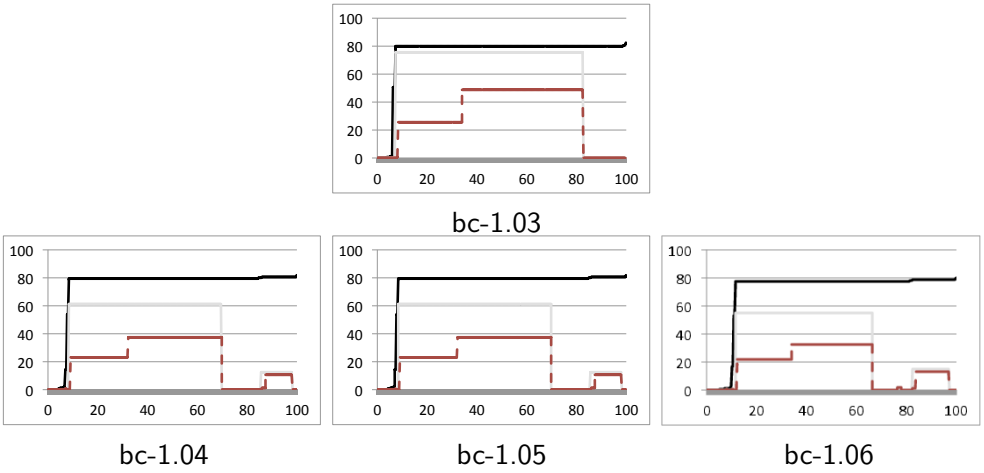


Figure 7.5: BSCGs for various bc versions

Figure 7.6 shows the SCGs for the four versions of acct considered in this study. The slice profile and the cluster profile show very little change between acct-6.3 and acct-6.3.2. Similarly, not much change is seen between acct-6.5 and acct-6.5.5. However, the slice and the cluster profiles change significantly between major revisions, 6.3.X and 6.5.X. The change log of release 6.5 notes "Huge code-refactoring." The refactoring of the code is primarily in the way system log files are handled using utmp_rd.c, file_rd.c, dump-utmp.c and stored using hash tables whose operations are defined in hashtab.c and uid_hash.c.

Finally, Figure 7.7 shows the SCGs for the 14 versions of indent. These revisions include two major releases. It is evident from the SCGs that the slice profile during the evolution hardly changes. The cluster profile also remains similar through the evolution. The system grows from 4,466 to 6,521 SLoC during its evolution which is supported by Figure 7.4c showing the growth of the system SDG size. Indent is a program for formatting C programs. A study of

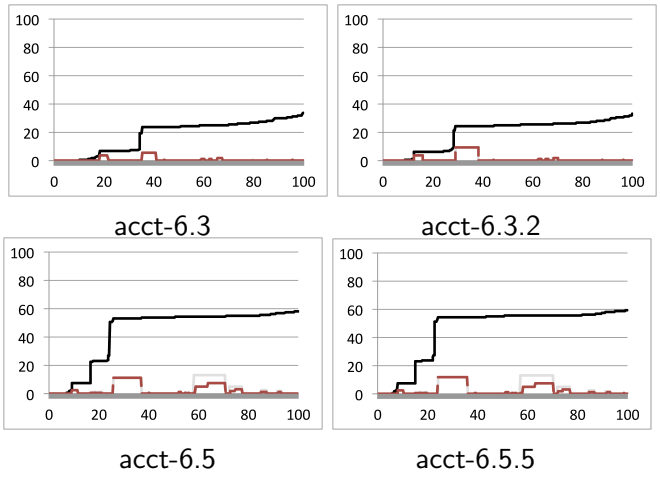


Figure 7.6: BSCGs for various acct versions

the change logs for indent did not reveal any major refactoring or restructuring. The changes to the system were mostly bug fixes and upgrades to support new command line options. This results in almost negligible changes in the slice and cluster profiles despite the system evolution and growth.

As an answer to *RQ4.2*, this study finds that unless there is significant refactoring of the system, coherent cluster profiles remain stable during system evolution and thus captures the core architecture of programs. The answer found for *RQ4.2* complements that for *RQ3.1* providing further support for the use of coherent clusters in revealing and understanding programs structure.

Future work will replicate this longitudinal study on a large code corpus to ascertain whether this stability holds for other programs. If such stability is shown to hold, then coherent clusters can act as signature of the program architecture opening up application in many areas of software engineering.

7.4 Inter-cluster Dependence

This section addresses research question *RQ4.3 What are the implications of inter-cluster dependence between coherent clusters?* The question attempts to reveal whether there is dependence (slice inclusion) relationship between the vertices of different coherent clusters. A slice inclusion relationship between two clusters X and Y exist, if $\exists x \in X : \mathsf{BSlice}(x) \cap Y \neq \emptyset$. If such containment occurs, it must be a strict containment relationship ($\mathsf{BSlice}(x) \cap Y = Y$) because of the external and internal requirements of coherent clusters.

In the series of case studies presented earlier we have seen that coherent

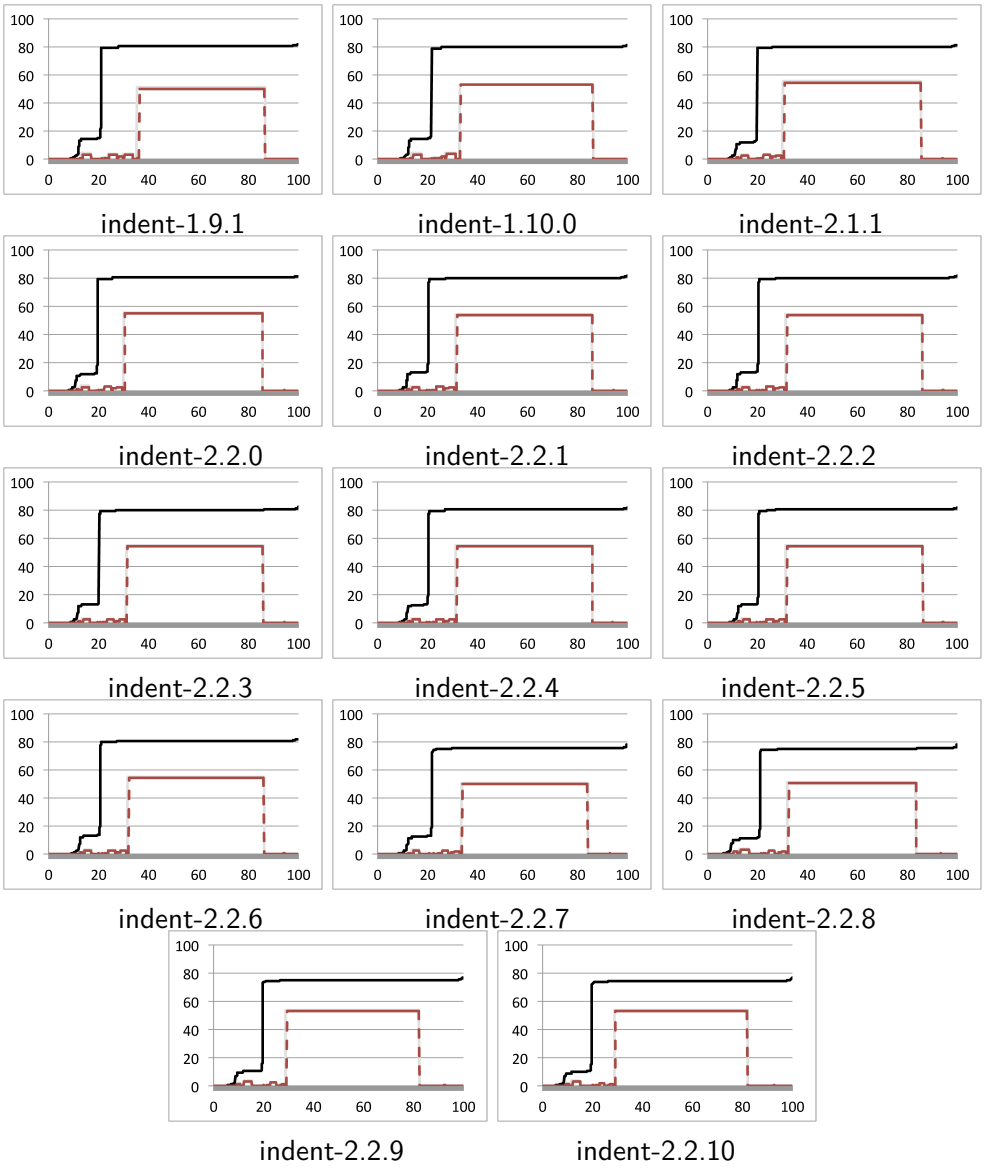


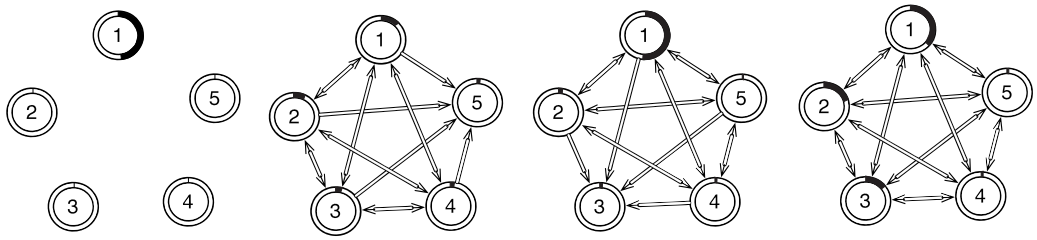
Figure 7.7: BSCGs for various indent versions

clusters map to logical components of a system and can be used to gain an understanding of the architecture of the program. If such inter-cluster dependencies exist that allows entire clusters to depend on other clusters, then this dependence relationship can be used to group clusters to form a hierarchical decomposition of the system where coherent clusters are regarded as sub-systems, opening up the potential use of coherent clusters in reverse engineering. Secondly, if there are mutual dependency relations between clusters then such mutual dependency relationships can be used to provide a better estimate of slice-based clusters.

All vertices of a coherent cluster share the same external and internal dependence, that is, all vertices have the same backward slice and also the same forward slice. Because of this, any backward/forward slice that includes a vertex from a coherent cluster will also include all other vertices of the same cluster (Equation 4.1). The study exploits this unique property of coherent clusters to investigate whether or not a backward slice taken with respect to a vertex of a coherent cluster includes vertices of another cluster. Note that if vertices of coherent cluster X are contained in the slice taken with respect to a vertex of coherent cluster Y, then all vertices of X are contained in the slice taken with respect to each vertex of Y (follows from Equation 4.1).

Figure 7.8 shows *Cluster Dependence Graphs* (CDG) for each of the four case study subjects. Only the five largest clusters of the case study subjects are considered during this study. The graphs depict slice containment relationships between the top five clusters of each program. In these graphs, the top five clusters are represented by nodes (1 depicts the largest coherent cluster, while 5 is the 5^{th} largest cluster) and the directional edges denote backward slice[3] inclusion relationships: $A \rightarrow B$ depicts that vertices of cluster B depend on vertices of cluster A, that is, a backward slice of any vertex of cluster B will include all vertices of cluster A ($\forall x \in B : \mathsf{BSlice}(x) \cap A = A$). Bi-directional edges show mutual dependencies, whereas uni-directional edges show dependency in one direction only. The outer ring of the nodes are also marked in solid black to represent the percentage of the program covered by the particular cluster that the node represents. For example, the largest cluster (node 1) of copia (Figure 7.8a) constitutes of approximately 48% of the program. In the graph for copia, the top five clusters have no slice inclusion relationships

[3]A definition based on forward slices will have the same results with reversed edges.



(a) copia (b) acct (c) indent (d) bc

Figure 7.8: Cluster dependence graphs

between them (absence of edges between the nodes of the CDG). Looking at size of the clusters, only the largest cluster of copia is truly large at 48%, while the other four clusters are extremely small making them unlikely candidates for inter-cluster dependence.

For acct (Figure 7.8b) there is dependence between all of the top five clusters. In fact, there is mutual dependence between clusters 1, 2, 3 and 4, while cluster 5 depends on all the other four clusters but not mutually. Clusters 1 through 4 contain logic for manipulating, accessing, and maintaining the hash tables, making them interdependent. Cluster 5 on the other hand is a loop structure within the main function for executing different cases based on command line inputs. Similarly for indent (Figure 7.8c), clusters 1, 2, 4, and 5 are mutually dependent and 3 depends on all the other top five clusters but not mutually.

Finally, in the case of bc (Figure 7.8d), all the vertices from the top five clusters are mutually inter-dependent. The rest of this section uses bc as an example where this mutual dependence is used to identify larger dependence structures by grouping of the inter-dependent coherent clusters.

At first glance it may seem that the grouping of the coherent clusters is simply reversing the splitting of same-backward-slice or same-forward-slice clusters observed earlier in Section 6.2.3. However, examining the sizes of the top five same-backward-slice clusters, same-forward-slice clusters and coherent clusters for bc illustrates that it is not the case. Table 7.2 shows the size of these clusters both in terms of number of vertices and as a percentage of the program. The combined size of the group of top five inter-dependent coherent clusters is 70.43%, which is 15.67% larger than the largest same-backward-slice cluster (54.86%) and 37.91% larger than the same-forward-slice cluster

Cluster Number	Same Backward-Slice Cluster Size vertices	%	Same Forward-Slice Cluster Size vertices	%	Coherent Cluster Size vertices	%
1	4,135	54.86	2,452	32.52	2,432	32.26
2	1,111	14.74	1,716	22.76	1,655	21.96
3	131	1.74	1,007	13.36	1,003	13.31
4	32	0.42	157	2.08	117	1.55
5	25	0.33	109	1.45	102	1.35
					Group Size:	70.43

Table 7.2: Various cluster statistics of bc

(32.35%). Therefore, the set of all (mutually dependent) vertices from the top five coherent clusters when taken together form a larger dependence structure, a closer estimate of a slice-based cluster.

As an answer to *RQ4.3*, this section shows that there are dependence relationships between coherent clusters and in some cases there are mutual dependencies between large coherent clusters. It also shows that it may be possible to leverage this inter-cluster relationship to build a hierarchical system decomposition. Furthermore, groups of inter-dependent coherent clusters form larger dependence structures than same-slice clusters and provides a better approximation for slice-based clusters. This indicates that the sizes of dependence clusters reported by previous studies [Binkley et al., 2008b, Binkley and Harman, 2005b, 2009, Harman et al., 2009, Islam et al., 2010b] maybe conservative and mutual dependence clusters are *larger* and more prevalent than previously reported.

7.5 Coherent Clusters in Object Oriented Paradigm

Recently, our finding that large clusters are widespread in C systems has been replicated for other languages and systems by other authors, both in open source and in industrial systems [Beszédes et al., 2007, Savernik, 2007, Szegedi et al., 2007, Hajnal and Forgács, 2011]. This section presents a study which looks at whether coherent clusters are prevalent or not in Java programs. Formally, this section aims to answer research question *RQ4.4 Are coherent clusters prevalent in object-oriented programs?*

The current slicing tools for Java are not very mature when compared to the tools available for C programs. Most of the Java slicing tools have limitations and are aimed at dynamic slicing [Hammacher, 2008]. Static slicing of Java is supported by Wala[4], Indus [Jayaraman et al., 2005] and Ariadne[5]. For the results presented in this section, we use the Indus Slicing tool.

7.5.1 Slicing Tool: Indus Slicer

The Indus framework works with Jimple [Vallee-Rai and Hendren, 1998], an intermediate representation of Java, provided by the Soot toolkit [Lam et al., 2011]. The framework is a collection of program analyses and transformations implemented in Java to customise and adapt Java programs. Indus combines various static analyses which enable static slicing of Java programs.

Indus defines Object-flow Analysis (OFA), which is a points-to analysis for Java. Each allocation site in the analysed system is treated as an abstract object and its flow through the system is tracked to infer the possible types. The precision of the analysis can be varied in terms of flow-sensitiveness for method local variables and object-sensitiveness for instance fields.

Dependence Analyses in Indus includes entry-based control, exit-based control, identifier-based data, reference-based data, interference, ready, synchronisation, and divergence analysis, needed for program slicing and partial evaluation. Additionally, Indus also includes escape analysis, side-effect analysis, monitor analysis, safe lock analysis and atomic analysis. Some analyses have varying levels of precision which can be varied.

The Java slicer for Indus can be used as a standalone command line tool or can be used as a plug-in for development platforms such as Kaveri for Eclipse [Ranganath and Hatcliff, 2007]. Indus supports backward and forward slice generation. Complete slices (union of backward and forward slices starting from the same slice criteria) can also be generated.

7.5.2 Experimental Subjects

The Java test subjects that are chosen for this experiment are all small open-source tools that are available as compilable/runnable JAR units. The experiment was conducted on 380 such programs. The details of the programs are listed in Table 7.3. Column 1 gives the name of the programs, Column 2 gives the number of statements, Column 3 and 4 list the size of the largest backward

[4] `http://wala.sourceforge.net/`

[5] `http://www.4dsoft.eu/solutions/4d-ariadne/try-ariadne`

and forward slices respectively. Columns 5 and 6 show the size of the largest backward and forward same-slice clusters. Finally, Column 7 shows the size of the largest coherent clusters. The size of the slices and clusters are presented as a percentage of the program and not absolute statement counts.

The smallest program considered is frt2-1.0 with a mere 7 statements while the largest program considered is commons-io-1.4 which is much larger with 7,886 statements. The size of backward slices range from 2% to 81% and for forward slices range from 1% to 97%. The average size of backward slice for the programs is 27% whereas that for forward slices is 31%. This finding is similar to results found by Binkley and Harman [2005a], where they looked at 45 C programs and found backward slices to be smaller than forward slices. Finally, the table also show programs that do not contain any significant (larger than 1%) coherent clusters and ones with *huge* coherent cluster covering over 50% of the program statements.

Program	Number of Statements	Largest Backward Slice	Largest Forward Slice	Largest same-backward cluster	Largest same-forward cluster	Largest Coherent cluster
addheader-1.0.0	475	63	79	17	17	17
ag32_gui	135	19	16	4	1	1
AgiTerm_02	1764	8	7	1	1	1
ast-dpc	763	24	55	3	23	1
AudioWrapper-0.1.1	1512	11	17	3	3	3
awesum2	433	13	22	4	5	4
BabelFiche	5222	45	70	9	13	9
Base64-0.91	392	36	61	10	14	10
base91	397	52	81	18	13	11
baseconv-1.2	1054	10	11	0	0	0
beanpeeler	1200	21	20	7	5	3
bearjangler.07	1113	11	27	3	3	3
bincodec	331	5	5	2	0	0
binddata2ui	1591	7	14	1	1	0
binpacking-1.0	739	27	26	1	9	0
bitmapfontcreator	1076	8	13	1	1	1
bitXml09	160	62	83	22	11	4
BlinkenApplet0.7	4096	57	36	10	8	8
BloatedToString-0.11	760	35	39	21	33	21
bromo	3806	26	45	2	4	2
BrowserLauncher	689	65	97	24	4	4
brtool	728	11	30	1	0	0
btsim-0.3	3979	28	50	1	44	1
BusinessDaysCalendar	254	28	34	2	1	1
bwbunit-1.0	272	39	72	15	39	15
canyon	141	20	21	4	1	1
CatChat	2411	18	21	2	3	2
CBinstaller	947	51	69	14	15	14
cc004	2212	16	20	3	4	2
Chartster-0.1	2061	19	23	0	0	0
chatw-0.01-gui-client	944	13	13	13	12	12
chronicj-1.0-alpha-3	887	34	33	11	10	8
CIDEMORF_1.1	2289	27	37	2	3	1
Cipher.Cracker-0.1	2484	31	29	1	3	0
CipherCore-0.1.0	2358	14	31	3	4	1
cips-client-1_0	233	26	35	9	29	5
cjan-zip	734	51	62	8	8	6
classes	3108	28	30	1	2	1
classfind-v1.2	481	28	44	3	2	2
Classifier4J-0.6	3595	17	30	14	15	14
clcl-0.5.1	1786	20	27	2	5	1

clientsession-0.2	279	25	15	2	1	1
CloneSweeper-bin-0.8	1624	37	68	13	8	5
clremote	1054	22	23	3	2	2
colorcrap06A	2985	15	11	1	1	1
comicviz-2.1	2052	20	23	6	6	6
commons-io-1.4	7886	9	10	2	2	1
CountdownTimer	2897	5	10	0	0	0
CpGIE2.0	3716	10	36	1	2	1
CreateTN	3815	15	24	2	3	2
CreditCardValidation	239	44	43	4	1	1
CrossIP	2241	42	61	9	8	7
CrozzSum (Kakuro)	1689	7	10	2	1	1
csgraph-1.00alpha1	1950	26	43	5	2	2
CUSP-0.2.5	1811	25	48	5	21	4
custom-phrase-1.0	656	16	21	1	2	0
DailyProphet-0.0.2	190	16	14	4	1	1
datapump	1578	16	27	8	7	6
Dataset_1.0	1934	14	22	1	1	1
DatePicker	13	14	1	0	0	0
de.axelrosenthal.jru	386	45	53	3	1	1
dealer2002	2810	29	35	1	4	1
DebtPayoff-1.01	406	22	45	3	1	1
denvir-1.0.1	268	32	57	8	7	7
DibujaChatUdp	989	25	11	6	6	6
diff	1196	81	78	28	28	27
DivideAndDo-0.6.0	2387	15	13	1	1	1
dom-result-set-1.0.rc-2	1096	19	18	2	3	2
dtdparser121	5009	40	47	11	17	10
easylayouts	1467	23	21	2	0	0
eclipseclient	89	60	65	24	24	24
EdCide	2778	9	8	1	1	1
editable-0.8.1beta	1493	3	6	1	1	1
Einsatzplanung	1240	37	31	0	0	0
elk_2.0	245	42	44	9	7	7
elvis	1409	48	51	2	3	2
enaf	1463	18	25	1	2	1
es.unizar.tecnodiscap	10	20	10	10	10	10
event4j-0.0.99	70	40	54	11	3	3
ExcelUtil	174	71	55	3	3	3
expresslib-1.2	1201	42	63	20	38	16
ezcontent-api	481	12	11	1	1	1
falseirc-20030930	4608	27	47	3	27	3
FARClient0.1.1	1549	28	30	2	7	2
feed-alfalfa	251	41	29	10	14	10
FileGrouper	586	18	30	2	1	1
filereplacer-0.3	515	45	59	3	3	3
filewatcher-1.1	1001	11	13	5	3	3
findEndorsedDirs	80	21	15	6	1	1
findinfiles	2210	20	21	3	5	1
finj-1.1.5	2670	20	23	1	2	1
flexitree	787	17	26	3	1	1
floatingAPI	615	17	29	2	2	1
form-authentication	253	38	35	1	0	0
FormantStat	3512	9	11	1	1	1
foxhunt-0.4	1325	22	14	3	0	0
framelauncher	185	31	34	6	6	6
framerd	4195	27	33	1	6	0
FreeGetNeo-0.2_bin	1425	24	33	14	15	14
freelance-bin-0.5	3543	9	17	1	1	1
freevoremotegui	229	10	11	3	1	1
freshtrash	474	18	7	1	0	0
frt2-1.0	7	29	14	14	14	14
fsp	4176	24	30	2	2	2
FtpGUI1_0	2166	14	20	1	3	1
Galaxian-.6	993	42	36	22	1	1
gandalf-0.2	627	5	10	1	1	1
gdsprint_0_2	722	9	8	1	0	0
goalSeeker1.0b	1523	35	34	8	6	3
gomoku-1.0	1290	38	36	11	16	10
graf-0.1	317	11	8	3	3	3
GraphADT	263	19	18	5	5	4
groab-0.2	1632	23	41	8	8	7
GuideLogFormatter	98	70	69	23	28	15

HatChat0.2	1481	6	10	3	3	3
HOM	3351	34	14	3	3	2
Howdy2	2135	20	36	6	18	5
hsolitaire-0.9	865	33	21	5	5	5
ht-commons-rating	476	60	22	1	0	0
ibtrader_0.1.2	4531	36	41	6	10	5
image2html	776	38	52	2	1	0
ImageSlideShow	2965	30	28	9	6	5
infomap.graph	939	17	15	1	1	1
ipcv0-991	2779	20	20	8	9	8
ISOCalc	743	53	37	2	1	1
ITClient	326	70	73	34	30	30
Jaguar	303	13	9	3	3	3
jakarta-regexp	3548	60	72	13	12	7
jasciidammit	196	34	56	7	5	5
jauus	3241	18	30	5	9	5
Java-5.1	2542	46	62	10	27	6
java-br	160	62	58	3	4	3
java_color	4702	38	50	21	14	10
javaconfig	1626	24	37	2	8	2
javaintegrity	3153	35	40	2	3	1
javaonjava	1437	33	48	8	20	3
JavaTextEditor	818	8	6	1	1	1
JavelinOrgChart	1576	19	26	1	1	0
JBackgammon	2577	24	27	1	3	1
jcalendar	576	12	11	1	0	0
jcalendarbutton	1312	7	7	0	0	0
jcbed	471	21	19	3	0	0
jcola_01_beta	1379	13	15	3	3	3
jcom	549	41	46	15	3	2
JConPool	1193	27	38	7	17	7
jconx_0_1_1	709	42	44	18	34	16
jdbcfacade-0_7	1577	24	30	1	1	1
jdbctester	2201	33	20	1	1	1
jdbnuke	118	23	19	6	6	6
jdonkey	2645	10	11	4	8	4
jdstream_0.6	2323	25	40	3	3	3
JDynDns02	510	19	27	5	10	5
jekyll-0.9	2359	25	46	1	2	1
jeline020925	1120	33	27	2	8	2
JEOPS	5064	18	52	5	13	5
JExplosion.0.6.3	2570	24	40	8	4	3
jfeedreader	2942	22	22	9	10	8
JFEP-0.1.0	2453	21	23	9	12	6
JFontChooser	351	13	10	1	0	0
jfreeprofilingcenter	1649	16	20	1	1	1
jFunction	2014	24	43	11	20	11
jibte	1303	15	14	2	5	2
JIsdnCall	2160	37	33	6	18	6
jlibdiff	3011	17	21	1	2	1
jline	3241	22	25	2	12	1
JListeners	335	14	16	4	1	1
jll	4408	43	20	4	2	2
JMasqDiale	19	22	2	5	2	2
jmbf-0.8	482	36	43	26	32	20
jmerge	3848	6	9	2	1	1
jmines	694	43	31	6	1	1
jmisc-2.0	4291	13	13	2	5	2
jmsg	1158	18	23	3	1	1
JMTF	4131	24	45	2	5	1
JMUT-0.1	727	17	24	2	0	0
jnetlib_1-0	625	35	27	16	11	10
jnetsend	1043	13	13	5	5	5
Joppelganger	488	44	40	26	33	26
jPad	1383	11	6	0	0	0
JPassGen	670	14	18	2	0	0
jperfanal	2633	15	35	5	8	4
jpicprog-1.1alpha	3605	21	17	2	3	2
jplus	828	28	17	4	2	2
jpui-0.4.0	1071	11	10	1	1	1
JQHull	20	15	5	5	5	5
jreg	892	24	17	3	1	1
jRegexDebugger	385	9	9	2	2	2

jRegexDyn-1.0.0	81	49	52	15	9	6
JReplace	742	23	36	7	7	7
jsiebex	1056	15	16	1	4	1
jsplitter	1620	32	44	8	32	8
jsrlog	1346	19	11	1	1	1
jSudoku	2071	28	22	4	5	3
jsvg	334	4	8	1	0	0
Jsymmetric	2515	26	30	6	2	2
jtemplate	2484	4	8	0	1	0
JTimeTrack	2976	13	9	1	1	1
jtranslit_1.3	1520	17	6	1	1	1
juc-1.0-b1	1001	23	46	7	9	7
jwf-1.0.1	520	14	9	2	0	0
JWizardComponent	2203	21	20	0	1	0
JXmlSql	3798	31	47	3	2	2
jz-decliner	6196	76	95	8	4	3
k5n-calendarpanel	2167	7	16	2	2	2
KFileCardLearn	4336	18	20	1	1	1
KGD	3756	60	50	19	36	10
ki18n	210	75	79	55	71	51
klang	965	55	54	7	6	6
kmap_ime-0.9	1466	27	34	5	30	5
Kritter	254	5	2	2	0	0
kxml2-2.3.0	6096	25	28	6	8	2
Laden	709	6	15	1	0	0
LayoutUtils	959	10	12	2	2	2
LBPool	1154	14	7	3	2	2
lexer	1364	31	34	5	13	2
lhadecompressor	3591	57	29	9	6	4
lib-pkg	792	10	14	5	10	5
lib4j-0.0.1	86	35	50	2	1	1
limacat-full-0.0.0.3	1073	19	9	1	0	0
lines	2497	19	22	2	0	0
litehttpd	1048	27	54	6	6	6
ln2-1-0RC	1489	20	25	3	5	3
LoadOMeter	518	9	15	3	4	3
log4jdbc	7803	49	24	0	0	0
LogView-1.1	869	11	13	2	2	2
LogWriter	2350	22	57	0	0	0
magelan	625	28	28	17	17	17
MailboxPkg	1179	18	22	3	7	3
ManyFold1.4	619	24	53	2	0	0
Mastermind	1200	49	24	8	9	8
matexRechnerPCE	2062	40	73	11	56	3
mathj-0.7	3246	28	36	11	7	5
maven-maximo	368	15	43	3	4	3
mbells_1.1	889	15	14	2	1	1
mcGPX-0.1	2158	32	21	0	1	0
MetaBossComponent	1168	35	61	11	17	10
mf2	1740	36	25	7	7	7
MicroDOM	532	32	43	9	15	8
microlaunch	5107	30	57	4	9	3
microten	650	28	51	2	11	1
millionenklick	529	19	70	6	12	4
MiniETL-jre14	586	49	84	11	19	8
MJFtpCL-0.3a	1178	43	78	2	17	2
ModuleDebug	158	11	7	3	1	1
monsoon-0.3	2968	7	12	1	1	1
Mp3Knife1	2593	35	32	7	4	3
mp3Splitter	1624	32	31	11	3	3
MRTGSpikeRemover	1333	23	23	5	4	3
msnj-0.3	2417	31	45	8	20	7
mucode	2851	23	45	3	9	3
multi-find	1057	15	12	5	7	5
mulumis	1111	32	39	10	6	5
nanoxml-2.2.1	3510	21	25	1	2	1
nanoXML-j2me	2900	46	52	8	6	5
NEOHacker0.7	3109	21	39	3	2	2
NetAntServer	437	78	76	57	57	57
netstool-0.2a	600	36	39	25	38	24
o3s	2081	22	34	8	14	8
objectsx	767	34	65	5	25	4
oow	1156	21	18	1	1	1

openbil	658	12	9	4	1	1
openMosixInfoServer	1431	57	56	5	27	5
openspreadsheet	4782	6	11	1	2	1
opera2mozilla	2534	26	24	11	11	10
OQuery_release	4100	27	38	12	19	11
ottobus-1.0-bin	3559	27	32	1	3	1
ouvert-gui-0_7	1081	5	6	1	1	1
oxo	423	66	67	28	27	23
passwordAnyhwere	1109	11	19	1	1	1
pathway2d	193	2	1	1	1	1
PatternSandbox-svn	2069	4	4	1	1	1
perlforjava	787	42	25	6	7	6
phantom-common	1263	25	21	4	2	1
picomapping-0.1.0	603	27	14	3	4	3
pisolutions	65	18	20	8	12	8
PlayC4Networked	1360	22	37	13	13	12
plotlib	488	36	24	6	2	2
pmon1.0bin	353	48	23	2	2	1
POC_Network	277	72	59	10	26	9
PocketKrHyperJ2SE	5357	38	54	4	9	3
Pol-IPj	2141	12	19	3	4	3
poormscapi	651	16	16	1	2	0
Pop3Browser	2625	45	47	3	10	2
prefedit-0.7	1790	16	12	1	0	0
proxool-0.5	4062	40	57	5	9	4
pvc	1994	25	26	2	2	1
Pwing_Alpha	1554	13	13	1	0	0
qciutil	3617	21	28	2	3	2
qf-misc-1.01	105	19	36	7	4	4
qLearningFramework	2105	30	11	1	0	0
queryfish-1.0	934	12	21	4	5	1
queryviewer-0.2	1061	12	13	1	1	1
Quick_STV_1_2	3968	26	35	7	1	1
Quickit-0.1	3106	6	9	1	1	1
quj	4009	14	25	2	2	2
RacingGame	468	19	28	1	0	0
ran.1.00	2855	12	39	1	0	0
RandomSelector	3341	5	5	3	1	1
rbc	714	15	14	1	0	0
RegExp-0.1	681	34	36	4	3	1
replayer_1_0b4	2049	17	15	3	3	3
repo	434	14	22	2	4	0
ResBundleEdit	3621	13	23	1	3	1
ritopt-0.2.1-bin	2916	27	37	7	24	6
RoMap-0.0.1	814	57	66	51	56	49
romzinger-0_1	1763	24	27	5	10	5
RPGServer	1173	57	60	4	0	0
rpnfc	2482	27	31	3	2	1
RWFile	335	48	64	3	1	1
SA-TriZ	1696	34	32	15	6	5
safejdbc	1795	20	16	2	1	1
sbev	575	11	13	6	4	1
scbugdb	2527	16	27	3	4	1
SciPlot0.1	421	29	71	1	0	0
scoof	662	23	39	2	2	2
scratch	938	20	48	3	1	1
Sequence_Splitter	626	53	55	15	15	15
shipConstructor	2590	61	39	2	1	1
Shougou-0_01	407	57	71	28	23	22
SIM	68	51	72	10	1	1
SIMPLE-0.1	2563	17	35	2	33	2
simplesqltools	939	27	37	2	8	1
SmartFAQ-1.0	982	17	34	4	5	4
smoothmetal	1151	6	3	0	0	0
SMS	895	7	8	1	1	1
smslib-0.1b	1246	15	50	1	1	1
somipla_0-3-0	1163	60	80	37	23	17
SpamFryer	221	17	20	10	8	8
sphaero	5337	19	28	2	8	2
spontane-0.1	869	12	8	1	0	0
sqlblox_A1	1613	30	21	1	0	0
sqlrecordset	865	47	68	11	11	9
stax-1.0	1326	6	6	1	0	0

stringtree-json	1081	38	46	16	22	15
Stringulator	1480	16	16	8	1	1
stucksweep	79	32	33	6	1	1
sudoku_0.1.1	947	38	44	16	19	12
sumproduct	20	65	35	10	5	5
swtautotester-0.15	423	12	23	3	1	1
syncop-gui-1.0	2017	6	17	1	1	1
tableview-1.4	3147	14	16	2	2	2
tagbuffer-beta4	571	49	56	3	2	1
tail	314	25	19	5	5	4
TallXSLT-0.0.2	585	29	29	10	7	3
TaxM	1027	11	7	0	0	0
tcpreflector-1.0.4	1041	8	14	2	5	1
test	62	18	11	8	3	3
tgIndex	2146	35	32	2	2	2
ThatIP-J	843	28	34	1	0	0
The15	1089	41	37	11	17	5
thememanager	732	20	26	9	9	8
ThreeD-2.0	4315	63	57	14	5	4
thumbnail1-beta	434	20	22	1	0	0
TiniDynDns-1.0	1206	58	78	19	54	17
TinyXML4Jsrc	152	9	7	1	1	1
tiwarriors	3355	29	18	2	2	1
tournament_manager	4830	9	13	1	1	1
TTT	759	18	12	6	0	0
TTTApplication	271	31	33	18	0	0
TWedit-beta4-source	1137	7	9	1	1	1
useful	830	22	8	2	2	2
variantcodec-0.0.3	3043	22	23	2	3	2
VersionCheckerV1.0	458	35	70	4	38	4
vschess-0.0	2545	38	42	1	2	1
watershed	475	31	32	5	2	2
WaveTree0.1	2518	58	57	4	10	3
weeje020924	729	13	9	1	0	0
WinRegistry-4.4	2625	64	56	4	4	4
WordLCSTest_V1	500	22	41	1	0	0
wsjl-swt-v0.2	651	12	9	0	0	0
xineo-xml-1.1.0	2725	24	30	5	13	4
xlife-masterserver	859	70	50	8	30	6
xml-im-exporter1	1640	12	16	3	2	1
xmlgridlayout0.4	457	18	33	4	0	0
xmltv2db-0.2	896	40	45	7	13	7
xparse-j-1.1	571	33	63	11	48	10
xsldb	718	77	65	9	12	9
XTransfer	1936	12	19	7	6	5
yadameter	1209	17	17	1	1	1
yajil-0.3.3	1974	16	11	4	2	2
Avg	1610	27	31	6	7	4
Min	7	2	1	0	0	0
Max	7886	81	97	57	71	57

Table 7.3: Subject programs

7.5.3 Do coherent clusters exist in Java Programs?

Figure 7.9 summarises the size of coherent clusters and shows the size of the clusters as a percentage of the program on the y-axis while the x-axis shows the program count in decreasing order of largest cluster size. The graph shows that only two of the programs have a *huge* cluster whose size is over 50%. Even more surprisingly only 41 programs contain a *large* coherent cluster whose size is between 10% and 50%. All the remaining programs contain small clusters which have size under 10%. In fact, 271 of the 380 subjects have clusters that are no larger than 4% of the program.

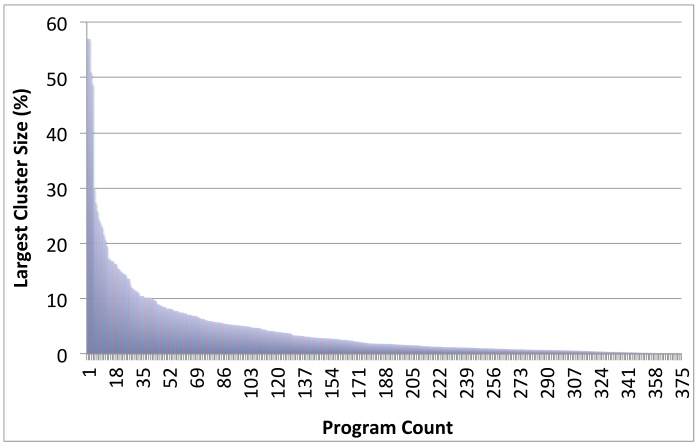


Figure 7.9: Coherent clusters in Java programs

The study therefore finds that only around 16% of the programs contain at least one *large* cluster covering 10% of the program. This is significantly lower than our findings for C programs where almost 70% of C programs contained a large coherent cluster. It should however be noted that Indus is not as mature and widely used as CodeSurfer and therefore there is less confidence in the results. Furthermore, Indus works on a intermediate representation of the Java byte code and does not provide mapping back to the original Java source code. The lack of this mapping did not allow for a manual validation of the quality of the results produced by Indus.

Additional quantitative study is also necessary to see if only programs above or below a certain size contain such large clusters. Qualitative studies of whether these clusters map to logical constructs in Java programs are also worthy of future research. Finally, it would be interesting to study whether this reduced clustering is specific to Java programs or are they a characteristics of object-oriented programs in general.

This study therefore finds an affirmative answer to research question *RQ4.4*, large dependence clusters do exist in Java programs, but they are less common than in C programs.

7.6 Related Work

Black et al. [2009] present a study on fault analysis based on program slicing. They investigate faults in barcode and extract measurements for Tightness and Overlap and functions. They find that the Tightness metric can be used as a predictor for program faults, functions with lower Tightness value were more

prone to containing faults.

Csaba et al. [2013] recently presented one of the first studies which explicitly looks at the relationship between clusterisation measures and software quality. They define various clusterisation measures for SEA-based dependence clusters and study the relationship of the clusters to software quality measured using probabilistic software quality model [Bakota et al., 2011]. They find that software quality has low correlation with dependence clusters.

Similar to our work on Java programs, Szegedi et el. [2007] analysed Java programs where they found existence of large dependence clusters amongst the programs. Hajnal and Forgács [2011] in their study find large dependence clusters in legacy industrial-scale COBOL systems.

7.7 Chapter Summary

This chapter presents four different studies between coherent clusters and various aspects of software engineering. The first of these studies looks at bug fixes made to the open-source program barcode and finds that coherent clusters have no links to bug fixes. The second study finds that coherent clusters remain very stable during system evolution and depict the core architecture of systems. Both these studies support the results of the previous chapter where we saw that coherent clusters depict the systems logical structure and should not be readily considered as harmful.

Inter-cluster dependency study shows that entire coherent clusters can be mutually dependent and this dependency can be used to form a hierarchical decomposition of programs. This relationship in future may be leveraged to perform automated software reverse engineering. Finally, this chapter also presents a study of 380 Java programs and finds that large coherent clusters are much rarer in Java programs compared to C programs.

Chapter 8

Coverage-based code reduction and Coherent Clusters

8.1 Overview

All work on dependence clusters thus far use static analysis in the form of static program slicing. The inherent conservative nature of static analysis can negatively impact the precision of static-analysis tools. The conservative nature of static slices often yields unnecessarily large slices [Binkley and Harman, 2003], leading to identification of large clusters. We saw an example of such problem in the case study for copia (Section 6.2.4), where (unwanted) dependence and conservatism of static analysis lead to the detection of a large dependence cluster. Some of this dependence is caused by unreachable code and code implementing cross-cutting concerns such as error-handling and debugging.

This chapter presents a framework for coverage-based static analysis in which coverage information is used to remove unexecuted and rarely executed code, followed by application of the static analysis on the reduced version of the program. The framework is applied to static slicing to reduce the size of the slices and thereby reducing the size of the clusters. The intuition behind the approach is that removing unwanted dependencies should break clusters, improving the mapping between clusters and logical program structures that we saw in Chapter 6.

More formally, this chapter addresses the following research questions:

RQ5.1 What is the impact of different test suites on static program slices and dependence clusters in coverage-based reduced programs?

RQ5.2 How large are coherent clusters that exist in the coverage-based reduced

programs and how do they compare to the original version?

RQ5.3 Which structures within a coverage-based reduced program can coherent cluster analysis reveal and how do they compare to the original version?

8.2 Background

Although static program slicing has matured significantly and is widely used by researchers, the uptake of the technology by software practitioners in the industry is limited. One of the main reasons is that most static slicing tools produce (necessary) conservative approximations. Some of this conservatism comes from over-estimation associated with having to analyse the whole system independent of 'typical' executions. For example, consider the need to analyse only the code executed under *normal* conditions, effectively ignoring error-handling code, which normally remains unexecuted. As a cross-cutting concern, error-handling code can often tie a system's components together, ignoring it can sharpen the static analysis and in particular produce smaller slices.

The combination of unexecuted code and the conservative nature of static analysis can negatively influence the clustering process and result in unwontedly large dependence clusters. For example, error handling code can create a single large dependence cluster when all modules depend on the error handling code and vice versa. If the error handling code is actually not executed and thus removed before clustering, the cluster may break up into multiple smaller clusters no longer held together by the error handling code. Moreover, the clustering will be based on the frequently executed parts of the program, which are more likely to be of interest to engineers working with the code.

This chapter presents a framework for *coverage-based code reduction*, which applies program slicing only to the sub-program built from the code executed (or covered) during typical executions. This will, for example, remove unreachable code, which is hard to identify during static analysis. While the framework can be instantiated with (and will work with) any static analysis, this thesis considers the impact of code reduction on static slicing and coherent clusters.

The approach is evaluated both quantitatively and qualitatively by studying clustering changes. Empirical validation includes a quantitative analysis of five open-source C programs. The study shows that large coherent clusters are present in both the original and the reduced versions of the programs, while a qualitative study of the program indent shows that clustering of the reduced

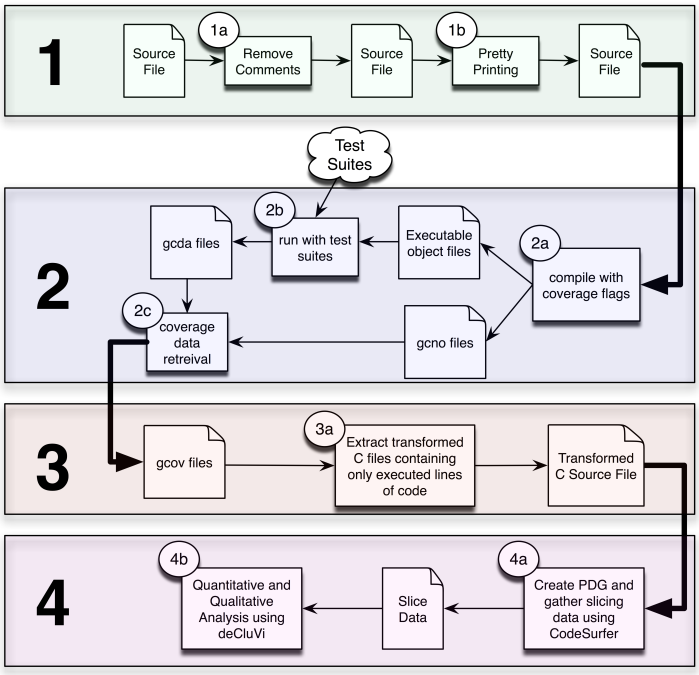


Figure 8.1: Coverage-based code reduction framework for static analysis

code better reflects a conceptual model of the program.

8.3 Framework: Coverage-based code reduction

This section outlines the reduction framework. It describes how coverage information replicating typical program execution or use cases is used to remove unexecuted code prior to static analysis.

The framework incorporates the following four stages:

1. Pre-processing – the first stage normalises the syntax of the code.

2. Gathering test coverage data – the second stage compiles and executes the normalised source to gather coverage information. For example using the programs regression test suite or typical use cases.

3. Code reduction – The coverage data is then used to remove code that has not been sufficiently covered. Here the level of coverage taken as sufficient is a parameter tuneable by the user. The most conservative options is to remove only unexecuted code.

4. Static analysis – The static analysis is applied to the reduced code.

Although the first three stages of the framework can be implemented in a single tool, our implementation of the framework for C relies on well-known Unix utilities as described below.

Stage 1: Pre-processing

The first stage pre-processes the system's source code by stripping comments and re-formatting the source using indent[1] to achieve uniform layout, which simplifies the reduction process in Stage 3.

1. **Remove comments:** All source files are pre-processed to remove single line and multiple line comments. This is achieved using regex [Friedl, 2006] and sed[2] utilities.

2. **Layout formatting:** This step uses indent to achieve a consistent layout for the source files to counter various layout styles/conventions followed by programmers. Consistent layout simplifies the process of removing unexecuted/uncovered lines of code. For example, the formatting involves limiting the maximum length of lines and placing entire method declarations/signature on the same line.

Stage 2: Gathering test coverage data

The second stage of the process involves compiling C source files obtained for Stage 1 with specific gcc flags to allow collection of test coverage data using gcov[3]. Gcov is a test coverage utility used in concert with gcc to analyse line coverage information when executing test suites.

1. **Instrumented application build:** This step involves building the project with coverage flags enabled to allow capture of profiling and coverage information. Projects built using GNU make will need the CFLAGS, CXXFLAGS and LDFLAGS in the Makefile set to `-fprofile-arcs -ftest-coverage`. The use of the coverage flag during compilation will not only result in executable object files being created but will also generate an additional .gcno file for each C source file. These gcno

[1] http://www.gnu.org/software/indent/
[2] http://www.gnu.org/software/sed/
[3] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

files contain profiling arcs information and are used by gcov for post-processing application's statistics collected at runtime.

2. **Execute test suites:** Following the application build, the program is executed using the test suites. Execution of the instrumented code collects coverage statistics at runtime and creates a set of gcda files (or updates existing ones) on exit. For every gcno file created during the build, a corresponding gcda file is created following execution. Gcda files are only generated when the application exits cleanly by either returning from `main()` or by calling `exit()`.

3. **Coverage statistics collection:** The last step of this stage involves running gcov tool on each of the C source files. This creates gcov files for each C source file from the corresponding gcno and gcda files. The gcov file is a copy of the corresponding C file with the addition of tags showing coverage information for each line. Gcov marks each line as either unexecutable, executable but not executed, or with the number of times the line was executed. These files are used in the next stage to perform transformation that removes executable but not executed lines of code.

Stage 3: Code reduction

A code transformation tool written by us is used to remove lines of code from the C files that remain unexecuted after running test suites in Stage 2. The tool uses information from the gcov files to perform the transformation. Gcov prepends each line of code in gcov files with either of three tags, '-' for lines that does not have any executable code, '#####' for lines that contain executable code but was not covered by test suites, and n where n is the number of times the line was executed. The transformation tool keeps lines marked with '-' and n intact and removes lines marked with '#####'. To keep the reduced code syntactically and semantically legal, there are several special cases that need to be handled and which makes the transformation challenging. For example, when a line being removed has non-matching '{' or '}' or when a function being removed is still referenced by other executed functions.

The reduction presently works at the line level because Gcov reports coverage information at line level. Using a zero percent threshold, this stage removes lines marked by Gcov as unexecuted (i.e., uncovered). The removal process is

made more aggressive by increasing the threshold and thus removing infrequently executed code (i.e., code not sufficiently covered during executions).

Stage 4: Static analysis

The framework's last stage is instantiated by any static analysis which is applied to the reduced code producing results specific to the covered code. In our case we use the following:

1. **Generating SDG and slicing data:** CodeSurfer is used to generate SDG for the program. The tool is also used to extract slice information for analysis.

2. **Cluster Analysis:** The slicing data extracted using CodeSurfer is used to locate coherent clusters which are analysed using *decluvi*.

8.4 Impact of coverage-based code reduction on Coherent Clusters

To empirically study the coverage-based static analysis framework, this section presents an empirical evaluation of coverage-based code reduction's impact on coherent clusters. After discussing the research questions, this section gives details of the experimental setup and the subject programs. It then presents a validation study on the use of regression test suites and the coverage achieved. This is followed by quantitative and qualitative studies of coherent clusters in the original and the reduced programs.

The section addresses three research questions. The first two research questions *RQ5.1 What is the impact of different test coverage suites on static program slices and dependence clusters in coverage-based reduced programs?* and *RQ5.2 How large are coherent clusters that exist in the coverage-based reduced programs and how do they compare to the original version?* provide empirical verification and validation. *RQ5.1* establishes the level of coverage achieved by regression test suites and compares it to the coverage for arbitrarily chosen test inputs. Whereas *RQ5.1* is concerned with the veracity of our approach, *RQ5.2* investigates its validity; if large coherent clusters are not present in the reduced code, our approach would not warrant further study. Additionally *RQ5.2* ascertains whether there are any discernible patterns in the way the clustering changes between the original and the reduced code. Finally,

Subject	C Files	Vertex count		Average Slice Size (vertices)		Largest Cluster Size	
		Original	Reduced	Original	Reduced	Original	Reduced
bc	9	7,557	6,510	5,144	4,126	32%	26%
byacc	13	9,515	7,041	2,852	1,993	6%	4%
cflow	25	12,329	10,957	5,127	4,303	8%	15%
ed	8	5,706	4,028	3,747	2,418	54%	62%
indent	11	7,543	5,172	4,487	2,361	52%	33%

Table 8.1: Subject programs

RQ5.3 Which structures within a coverage-based reduced program can coherent cluster analysis reveal and how do they compare to the original version? establishes whether clusters in the reduced program correspond to a logical system decomposition and whether this mapping is different from the one found in Chapter 6.

8.4.1 Experimental Subjects and Setup

The study considers five of the subject programs from Table 4.1. Table 8.1 presents statistics of these five programs, it shows the number of files containing executable C code (column 2), SDG vertex counts (column 3–4), average slice size (column 5–6), and size of the largest coherent clusters (column 7–8) as a percentage of the program. Two values are provided for the last three attributes, one for the original program (columns 3, 5 and 7) and the other for the reduced version (columns 4, 6 and 8).

8.4.2 Coverage achieved by regression suites

Research question *RQ5.1* ascertains the level of coverage achieved by the regression suite shipped with each system. For indent it also compares this coverage with that attained using an arbitrarily chosen test input. We employ an arbitrary test suite to do a two-fold verification study. Firstly, to check that regression suites achieve higher coverage than an arbitrary test suite, and secondly to understand the effect on clustering of using arbitrary test suite as opposed to that of using a carefully-crafted (regression) test suite.

Table 8.2 presents coverage information for the five programs. Column 2 gives the lines of code, column 3 shows the executable lines as counted by Gcov and subsequently columns 4 and 5 show the coverage achieved. The results show that coverage varies widely between the test subjects. The lowest coverage was recorded for ed at 42% with the highest at 81% for indent. The variation in the coverage is due to the nature of the programs. Ed is an interactive line-based text editor. As such, it is difficult to test its functionalities to the fullest

Subject	Lines of Code	Executable Lines	Covered Lines	Coverage (%)
bc	7,618	1,849	1,222	66%
byacc	7,320	3,450	2,604	76%
cflow	5,453	1,711	1,046	61%
ed	2,836	1,334	563	42%
indent	10,300	2,861	2,306	81%
Average	**6,705**	**2,241**	**1,548**	**65%**

Table 8.2: Regression suite coverage

via a scripted test suite and without user interaction. On the other hand, indent is a source code formatter whose functionality can be easily exercised using automated scripts. To answer the first half of *RQ5.1* as to how much coverage is achieved by regression test suites, we find that the average coverage achieved for the programs is 65%.

To answer the second part of *RQ5.1*, we compare the coverage achieved by the regression suite shipped with indent to that of an arbitrary test suite composed of an assortment of C source files. The assortment of C files include source code from Ubuntu and 30 open-source programs for a total of 21,085 C source files with a combined 1,581,658 LoC. For ease of discussion we will refer to this suite of files as the *arbitrary test suite*.

Table 8.3 shows the details of the coverage achieved by the two test suites. Column 1 lists the source files of indent, with column 2 and 3 presenting the lines of code in each source file and number of executable lines of code, respectively. Columns 4, 5, and 6 present the coverage achieved by the two test suites: using the supplied test suite (Column 4), the arbitrary test suite (Column 5), and their combination (Column 6).

The results show that the regression test suite achieves higher coverage than the arbitrary test suite. This is because the regression test suite is carefully designed to test system functionality by replicating use cases. The regression test suite and the arbitrary test suite independently achieve similar results for most of indent's source files with the exception of two. The regression suite scores 42% higher coverage for args.c. This is because the regression test suite exercises arguments and parameters to check their functionality, whereas, the arbitrary suite was run with the same standard arguments. On the other hand, the arbitrary test suite achieves 54% higher coverage for backup.c. This file has

File	LoC	Executable lines	Lines covered by test suite		
			Regression only	Arbitrary Only	Regression and Arbitrary
args.c	1,233	189	104	25	104
backup.c	533	106	18	75	75
code_io.c	522	121	82	82	82
comments.c	879	292	265	200	270
globs.c	124	26	14	14	14
gperf.c	205	8	8	8	8
indent.c	3,339	1,037	846	868	893
lexi.c	1,201	359	303	305	325
output.c	1,550	501	457	448	474
parse.c	669	217	209	210	210
utils.c	45	5	0	0	0
Overall coverage			**81%**	**78%**	**86%**

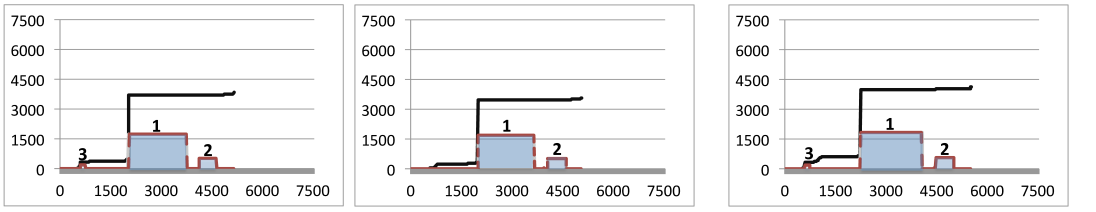
Table 8.3: Test coverage for individual files of indent

functions that create unnumbered backups by appending a tilde or create incremental numbered backups. The arbitrary suite is an assortment of C files taken from systems where many files have the same name, resulting in the use of incremental backup code. This increases the number of functions exercised within the file resulting in higher coverage.

Finally, the use of the regression test suite in conjunction with the arbitrary test suite results in 5% higher coverage than using regression suite alone. This could indicate that using a combination may be more suitable for our approach as it achieves higher coverage. However, we are also interested in knowing how this higher coverage relates to clustering.

The SCGs (Figure 8.2) for the reduced programs corresponding to each test suite usage are similar. All three SCGs show a drop in the program size from around 7,500 to 5,000. The graphs have a large coherent cluster (marked with 1) towards the middle of the plot, running approximately 1,750 to 4,250 on the x-axis, and a smaller cluster (marked with 2) just to the right, running approximately 4,300 to 4,600 on the x-axis. The only difference visible in the graphs is the presence of a small cluster (marked with 3) in the versions that use the regression test suite which is absent from the version that uses only the arbitrary test suite. This cluster spans 650 to 700 of the x-axis in the B-SCG of the versions using the regression suite.

Therefore, as an answer to second part of *RQ5.1*, we find that, although the regression test suite exercises indent using a much smaller input (11,517 LoC) when compared to the arbitrary test suite of over 1.5 million LoC, better coverage is achieved. Combined, the two suites yield slightly higher coverage, but do not produce differences in clustering. This gives us some confidence



Regression test suite Arbitrary test suite Regression & arbitrary suite

Figure 8.2: Indent B-SCGs for coverage based reduction

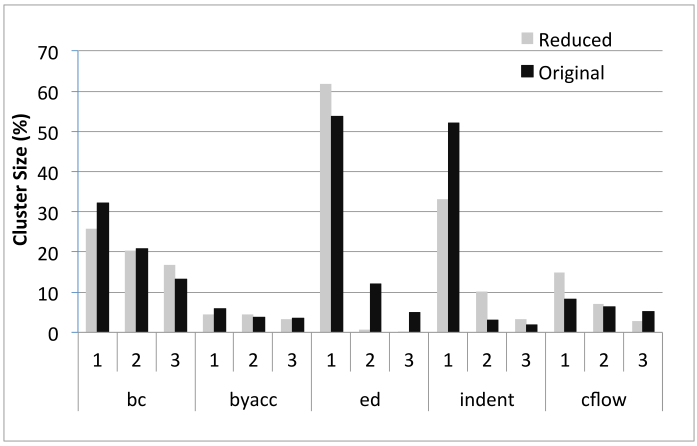


Figure 8.3: Top three coherent cluster sizes

that the use of regression test suite alone is sufficient. A future extended study is planned to compare the impact of dedicated (regression) test suites and those generated using automated test generation tools such as Klee [Cadar et al., 2008], Austin [Lakhotia et al., 2010] and Milu [Jia and Harman, 2008]. Henceforth, the chapter reports results attained using each program's supplied regression test suite.

8.4.3 Change in Coherent Cluster Sizes and Patterns

This subsection addresses the validation study and thus answers research question *RQ5.2*. This question asks whether large coherent clusters are present in the reduced code and what pattern of change can be identified in the clustering between the original and reduced version. If coherent clusters do not exist in the reduced code or if the changes in clustering between the original and the reduced version is minimal, then there is little value in applying the framework to coherent clusters.

To answer *RQ5.2*, the classification of programs introduced in Section 4.7 is used. Programs are classified as containing *small*, *large* and *huge* clusters

based on the size of the largest cluster in them. Figure 8.3 shows the size of the three largest coherent clusters in the subject programs for both the original and the reduced versions. 1, 2, 3 on the x-axis denotes these clusters, with 1 denoting the largest cluster, 2 denoting the second largest cluster and 3 denoting the third largest cluster in each of the programs.

The original versions of two of the five subject programs, byacc and cflow do not have any large coherent clusters. Original version of the remaining programs, ed and indent have *huge* coherent clusters, with bc having a *large* coherent cluster. The reduced version of bc also has a large coherent cluster of size 26% after reduction of 6%. The reduced version of indent no longer contains a *huge* cluster as the size of the largest cluster drops from 52% to 33% showing a significant reduction. Surprisingly, for ed the size of the largest cluster grows from 54% to 62% as both versions of the program contains *huge* clusters. Thus as answer to first part of *RQ5.2* we see that coherent clusters present in the reduced version of programs can be both *large* and *huge*.

Having established that coherent clusters are suitable for experiments with the framework, the remainder of this section presents the outcome of a visual inspection of the B-SCGs for the original and the reduced programs. The study addresses *RQ5.2* and identifies patterns of change in the clustering from the original to the reduced version of the programs.

Figure 8.4 present the B-SCGs for both the original and the reduced version of each program. The values of the axises of the graphs are shown as vertex counts rather than relative values (percentages). This helps appreciate the changes in the clustering following program transformation. The graphs in the left column are for the original programs, those on the right are for the reduced version. While not all large (surpassing the 10% threshold), coherent clusters are visible in SCGs for both the original and the reduced version of all five programs.

As shown by how far the plots extend along the x-axis, the SCGs of indent show a substantial reduction of program's size from approximately 7,500 to 5,000 vertices. The original version of indent has one large cluster running from approximately 2,200 to 6,200 on the x-axis. There are also two smaller clusters visible in the original version to the left of the main cluster. These two are approximately 223 and 144 vertices in size. The largest cluster in the reduced version is much smaller than the largest cluster in the original version.

However, the second largest cluster both appears to the right of the largest and is bigger than the second largest cluster from the original SCG. The implication here is that the largest cluster from the original version has broken into two clusters in the reduced version. This implication is confirmed by a detailed study of indent presented in Section 8.4.4.

The original version of ed has three identifiable coherent clusters, one of which is extremely large spanning from 1,650 to 4,800 on the x-axis of the B-SCG. The other two coherent clusters are approximately 600 and 250 vertices in size. The reduced version of the program on the other hand has only one visible coherent cluster spanning from 500 to 3,000 of the x-axis in the B-SCG. The largest cluster in the reduced version has also shifted left compared to the original, indicating that vertices yielding small slices are removed during reduction. The number of vertices in the program's SDG is also reduced from 5,706 to 4,028. A qualitative study of the clustering change in ed is presented in Section 8.4.5.

The SCGs of cflow show that coherent clusters present in both the original and the reduced versions of the program are much smaller than those seen in indent and ed. Multiple small coherent clusters are visible in the B-SCG of the original program with the largest being about 500 vertices in size. The reduced version also has multiple coherent clusters, with the largest two being clearly identifiable. However, the largest cluster in the reduced version is *bigger* in actual size than the largest cluster of the original version. This is the only instance where an increase in the size of the largest cluster is observed despite the reduction in program size. A detailed discussion of this phenomenon is given later in Section 8.4.5.

As was the case in the previous three subjects, the SCGs of bc shows that the size of the program decreases following reduction. However, bc exhibits the least change in clustering between the original and the reduced version of the code. Clustering in the reduced version remains identical except for a proportionate drop in the size of the top two clusters. The drop can be attributed to the reduction of the (average) slice size (solid black line).

For the last program, byacc, the SCGs show that the reduced program has significantly fewer vertices compared to the original program. Unlike other programs, the plot for slice size shows a significant change. However, there is no change observed in terms of clustering other than a small reduction. This

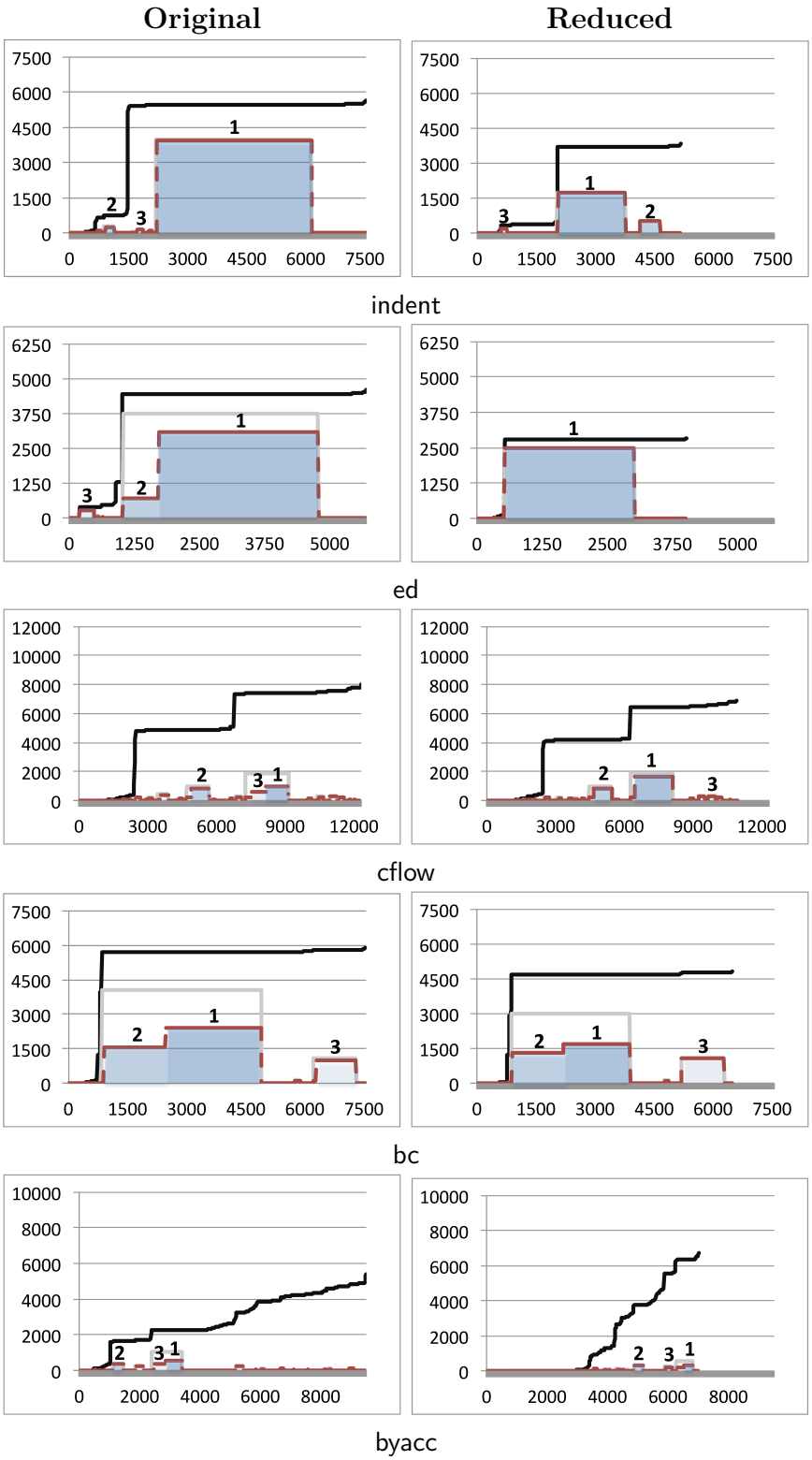


Figure 8.4: B-SCG (Backward Slice/Cluster Size Graph)

is attributed to the original version having small clusters to begin with.

The change in clustering observed amongst the subject programs varied significantly exhibiting some interesting patterns. There were four change patterns identified as detailed below:

break: This pattern occurs where a large cluster breaks into multiple clusters as in the case of indent. Although programs in this category will exhibit reduction in cluster sizes, the change is dominated by the break.

join: This pattern will only occur in rare cases where two separate clusters in the original version merge into a single cluster in the reduced version. This can lead to the formation of a larger cluster not found in the original version. Cflow is an example of where this pattern occurs.

remove: This pattern is exhibited by programs when clusters present in the original program disappear in the reduced program. Ed is an example where this pattern occurs.

drop: The fourth pattern occur when a reduction of the coherent clusters are observed. Both bc and byacc are examples of programs that fall in this category. Bc is a good example of a drop because the cluster sizes drop but the overall profile remains the same.

As an answer to *RQ5.2*, this study finds that reduced version of programs contain both *large* and *huge* coherent clusters. Furthermore, the SCGs show that there are significant differences in the cluster profile of the original and the reduced versions of the programs, where the changes can be classified into four patterns. This motivates detailed study of the clustering changes for the programs indent, ed and cflow, which are presented in the following sections.

It is also interesting to note that in each of the reduced version of the programs, the average slice size drops significantly. Reduction for individual programs are 20% for bc, 30% for byacc, 16% for cflow, 35% for ed and 47% for indent. As both the original and the reduced version of the program can be considered equivalent because both would give same results for the regression test suites, this reduction can be seen as an improvement on the precision of static slicing. Smaller slices make it easier for developers and programmers to understand and use slicing.

Cluster	Original				Reduced version (Regression Suite)			
	Cluster Size		Files	Functions	Cluster Size		Files	Functions
	%	vertices	spanned	spanned	%	vertices	spanned	spanned
1	52.1%	3930	7	54	33.2%	1715	8	36
2	3.0%	223	3	7	10.1%	521	3	13
3	1.9%	144	1	6	3.2%	164	2	3
4	1.3%	101	1	5	0.5%	24	1	1
5	1.1%	83	1	1	0.3%	15	1	1

Table 8.4: indent's cluster statistics

8.4.4 Case Study: indent

This subsection addresses *RQ5.3*. The subsection conducts a detailed comparison of how the composition of the clusters vary between the original and the reduced version of indent. We have shown that coherent clusters map to logical program constructs (Chapter 6) and hypothesise that removing unexecuted (or rarely executed) code (which may be holding clusters together) will allow clusters to reveal a finer grained logical model due to cluster breaking. We use indent as a case study as it exhibits the *break* pattern.

The original version of indent consists of 10,300 LoC with 7,543 vertices in its SDG. The reduced program on the other hand consists of 8,197 LoC and 5,172 vertices. Table 8.4 shows statistics of the five largest clusters found in the program. Column 1 gives the cluster number, where 1 is the largest and 5 is the 5^{th} largest cluster measured in number of vertices. Columns 2–5 give data for the original version of indent. Columns 2 and 3 show the size of the cluster as a percentage of the program's vertices and actual vertex count. Columns 4 and 5 show the number of files and functions where the cluster is found. Columns 6, 7, 8, and 9 give data for the reduced version of indent, and mirror those presented in columns 2, 3, 4 and 5, respectively.

The original version of indent has one extremely large coherent cluster covering 52% of the program. The remaining clusters are much smaller in size with the second largest being only 3%. The details of the top five clusters along with the functions that comprise of the cluster and their mapping to logical structure for the original program is discussed in Section 6.2.2. The remainder of this section discusses the change in clustering of the reduced version when compared to the original version of indent.

The size of the overall program drops from 7,543 to 5,172 vertices for the reduced version. This drop is due to the removal of code not covered by the regression test suite. The top two coherent clusters in the reduced version are

large clusters at 33.16% and 10.07%. The third cluster is just over 3% in size whereas cluster 4 and 5 are less than 0.5% in size.

Cluster 1 in the reduced code consists of 36 functions also found in cluster 1 of the original code. This cluster still contains all functions that begin with the prefix "handle_token" and the helper functions such as check_code_size, check_lab_size, search_brace, sw_buffer, print_comment and reduce. The cluster also consists of the main loop of indent (indent_main_loop) and the parser function parse.

Cluster 2 of the reduced code now consists of 13 functions that are part of cluster 1 in the original code. These functions which have broken away from cluster 1 to form their own cluster are responsible for outputting the formatted code. They include better_break, computer_code_target, dump_line, dump_line_code, dump_line_label, inhibit_indenting, is_comment_start, output_line_length and slip_horiz_space, and ones that do flagging and memory management, clear_buf_break_list, fill_buffer and set_priority. This breaking of a cluster supports the observation that post reduction clusters better reflect the logical decomposition. In this case cluster 1 is now responsible for *parsing* and *handing of the individual tokens*, and cluster 2 is responsible for *memory management* and subsequent *output*.

Some of the error handling code of cluster 2 from the original program no longer exist in the reduced program as it was not exercised by the regression test suite. Example of functions completely removed are DieError, usage and arg_missing. Furthermore, the code reduction doesn't always remove entire functions but also eliminates branches that remain uncovered by the regression test suite. Figure 8.5 shows an example, extracted from indent. The first element on each line (in red/grey) identifies whether the line has executable code and the execution count for the line (if applicable). The second element is the line number followed by the code. For example lines 664 and 667 executed 1,008 times while line 665 has executable code but is never covered. The code shows an if ... else block which executes 1,008 times and leads to the else branch every time. The true branch for the if statement is never covered by the regression suite. Removal of line 665 removes the call to function DieError. Similarly, another 4 uncovered calls to DieError are removed from indent, which ultimately results in the complete removal of function DieError from the reduced program. Removal of such code can improve the "sharpness" of static analysis

Original Code Functions		**Reduced Code Functions**		
Cluster 1	Cluster 2	Cluster 1	Cluster 2	Cluster 3
better_break	|DieError|	check_code_size	better_break	eqin
check_code_size	addkey	check_lab_size	clear_buf_break_list	option_prefix
check_lab_size	arg_missing	copy_id	compute_code_target	process_args
clear_buf_break_list	eqin	handle_the_token	dump_line	set_option
compute_code_target	option_prefix	handle_token_attribute	dump_line_code	
copy_id	process_args	handle_token_binary_op	dump_line_label	
dump_line	set_option	handle_token_colon	fill_buffer	
dump_line_code	usage	handle_token_comma	inhibit_indenting	
dump_line_label		handle_token_comment	is_comment_start	
fill_buffer		handle_token_decl	output_line_length	
handle_the_token		handle_token_doublecolon	set_next_buf_break	
handle_token_attribute		handle_token_form_feed	set_priority	
handle_token_binary_op		handle_token_ident	skip_horiz_space	
handle_token_casestmt		handle_token_lbrace		
handle_token_colon		handle_token_lparen		
handle_token_comma		handle_token_newline		
handle_token_comment		handle_token_nparen		
handle_token_decl		handle_token_postop		
handle_token_doublecolon		handle_token_preesc		
handle_token_form_feed		handle_token_question		
handle_token_ident		handle_token_rbrace		
handle_token_lbrace		handle_token_rparen		
handle_token_lparen		handle_token_semicolon		
handle_token_newline		handle_token_struct_delim		
handle_token_nparen		handle_token_swstmt		
handle_token_overloaded		handle_token_unary_op		
handle_token_postop		inc_pstack		
handle_token_preesc		indent_main_loop		
handle_token_question		parse		
handle_token_rbrace		parse_lparen_in_decl		
handle_token_rparen		print_comment		
handle_token_semicolon		reduce		
handle_token_sp_paren		search_brace		
handle_token_struct_delim		set_buf_break		
handle_token_swstmt		skip_buffered_space		
handle_token_unary_op		sw_buffer		
inc_pstack				
indent_main_loop				
inhibit_indenting				
is_comment_start				
lexi				
need_chars				
output_line_length				
parse				
parse_lparen_in_decl				
print_comment				
reduce				
search_brace				
set_buf_break				
set_next_buf_break				
set_priority				
skip_buffered_space				
skip_horiz_space				
sw_buffer				

Table 8.5: Function Cluster Mapping (Original and Reduced indent)

```
 1008: 664: if (!found)
#####: 665:    DieError (...)
   -: 666: else
 1008: 667:    ...
```

Figure 8.5: Uncovered code reduction

and the clustering as we have seen above.

Cluster 3 of size 3.2% now consists of functions that were part of cluster 2 in the original version of the program. These functions are responsible for handling command line options and include option_prefix, set_options, process_args and equin. As the remainder of the clusters are smaller than 1% they are not discussed in further detail.

The case study of indent illustrates that coherent clusters in the reduced version of programs obtained by removing unexecuted error-handling and debugging code can better capture the program's logical model. The breaking of the largest cluster by code reduction allows the clustering to be done on much finer level of logical constructs. As an answer to research question *RQ5.3*, coherent clusters in the reduced program also map to logical constructs of the program and possibly at a much finer level.

8.4.5 Increase in the size of the largest cluster

Coverage based code reduction (for coverage $< 100\%$) always leads to a drop in the program size and average slice size (Table 8.1). In the four test subjects bc, byacc, ed, and indent, the size of the largest coherent cluster also decreases. With the exception of ed, this decrease is also translated to a percentage drop. In the case of ed there is a positive percentage change, which means that the reduction removes more vertices outside the largest coherent cluster than from within the cluster. In addition, Cflow shows an actual increase in the size of the largest cluster.

The original version of ed has 5,706 vertices which is reduced to 4,028. The size of the largest cluster of the original program is 3,064 vertices as opposed to 2,485 vertices in the reduced version. However, despite the decrease in actual size, the relative size of the largest cluster increases by almost 8%. Cluster 1 of the original version includes 67 functions which increases to 74 functions in the reduced version. The additional functions are part of cluster 2 of the original version. The join happens because 9 of the functions in

cluster 1 and 6 of the functions in cluster 2 from the original code are removed during transformation. These functions are responsible for dealing with user interaction and error-handling. The removal causes the slices of the remaining vertices from both cluster 1 and cluster 2 to become identical. As vertices of both clusters following transformation result in the same backward and forward slice, they join into a single cluster.

The same phenomenon is also observed in cflow. Despite a drop in program size from 12,329 to 10,957 vertices, the largest coherent cluster increases both in actual and relative size. A detailed future study is planned to ascertain which vertices were originally separating these joined clusters using variation of linchpin identification technique [Binkley and Harman, 2009].

8.5 Related Work

Agarwal et al. [1993] present *approximate dynamic slicing* which is similar to the reduction framework introduced in this chapter. An approximate dynamic slice is obtained by taking the intersection of the appropriate static slice with the program execution path for the test case that is of interest. In contrast we first run all our test cases and identify all the execution paths exercised by the test cases. Subsequently, we perform static slicing on the reduced version of the program.

Ernst [2003] suggests that static and dynamic analysis have synergy and duality, and promotes their use in conjunction as a form of hybrid analysis. There are several slicing approaches that follow this idea. Call-mark slicing [Nishimatsu et al., 1999] uses static control and data dependence analysis along with function call information from dynamic execution traces to reduce static slice size. Similarly, dependence-cache slicing [Takada et al., 2002] uses static control dependence analysis and dynamic data dependence analysis to create program dependence graph used for slicing. Hybrid slicing [Gupta et al., 1997] uses dynamic function call graphs and break points information to reduce static slice size. Similarly, calling context captured using call stack information during execution is also used to reduce slices [Krinke, 2006]. Other methods focus on the semantics of programs to reduce slices. Conditioned slicing [Canfora et al., 1998] employs a condition in a slicing criterion, where statements not matching the condition are removed from slice. Amorphous slicing [Harman et al., 2003] allows for simplifying transformations that preserve semantic

projection. Although these approaches use both static and dynamic analysis, to the best of our knowledge we present the first approach where dynamic analysis is used to reduce code, followed by static analysis of the transformed program.

Binkley et al. [2013b] have recently introduced the notion of observation-based slicing. A slice is obtained through repeated statement deletion and validated by ensuring that the sliced and the original program behave exactly the same with respect to the slice criterion. This approach is similar to our code reduction where execution traces are used to identify and remove unwanted code thereby improving static analysis (slicing).

Acharya and Robinson [2011] present an approach that trades off static analysis (pointer analysis) precision for efficiency and speed, which is similar to our work where we are able to vary the definition of "less" executed code to reduce unwanted dependency. Beszédes [2007] introduce SEA-based clusters, these are clusters that consider program functions as entities rather than vertices of SDGs making the approach less precise but efficient. There are also studies on identification of 'linchpin vertices' that hold clusters together, removing which can reduce the size of dependence clusters significantly [Binkley et al., 2013a, Schrettner et al., 2012].

8.6 Chapter Summary

The chapter introduces a novel framework of using test cases for coverage-based code reduction to improve static analyses by reducing the source code to the frequently executed parts. The approach is applied to coherent cluster analysis. The change in the clustering following code reduction portrays four different patterns of change. These patterns reveal that removing debugging and error-handing code will often lead to breaking of larger clusters into smaller ones.

A validation study shows that large coherent clusters are present in the reduced code and a detailed study of indent shows that clusters in the reduced code better model program's logical structure. Future work will consider the impact of use cases and test suites generated by automated test generation tools within the framework and its application to other static analyses.

Chapter 9

Conclusion

This chapter starts with a discussion of the threats to validity for the results presented in this thesis. It then discusses the achievements of this thesis. This is followed by an outline of the various strands of future work to be done as follow up from this work and concluding summary.

9.1 Threats to validity

This section discusses the threats to the validity of the results presented in this thesis. Threats to three types of validity (external, internal and construct) are considered. The primary external threat arises from the possibility that the programs selected are not representative of programs in general (i.e., the findings of the experiments do not apply to 'typical' programs). This is a reasonable concern that applies to any study of program properties. To address this issue, a set of thirty open-source and industrial programs were analysed in the quantitative study. The only criteria used to select the programs was to ensure that CodeSurfer was able analyse each of the programs. However, these were from the set of programs that were studied in previous work on dependence clusters to facilitate comparison with previous results. The four case studies (acct, indent, bc and copia) were also selected based on the interesting patterns that their clustering profiles revealed during the experimentation with graph-based visualisation. In addition, the majority of the research conducted in this thesis was done on C programs, so there is greater uncertainty that the results will hold for other programming paradigms such as object-oriented or aspect-oriented.

Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variables on the dependent variable. The use of

hash values to approximate slice content during clustering is a source of potential internal threat. The approach assumes that hash values uniquely identify slice contents. Hash functions are prone to hash collision which in our approach can cause clustering errors. The hash function used is carefully crafted to minimise collision and its use is validated in Section 4.6. Furthermore, the identification of logical structure in programs were done by us (academic collaborators and me). As we were not involved in the development of any of the case study subjects, this brings about the possibility that the identified structures do not represent actual logical constructs of the programs. As the case studies are Unix utilities, their design specification are not available for evaluation. However, all of us that studied the subject programs and their architecture have many years of programming experience and independently did their analysis before coming to a consensus providing internal validation.

Construct validity refers to the validity that observations or measurement tools actually represent or measure the construct being investigated. In this thesis, one possible threat to construct arises from the potential for faults in the slicer. For the C programs analysed, a mature and widely used slicing tool (CodeSurfer) is used to mitigate this concern. Another possible concern surrounds the precision of the pointer analysis used. An overly conservative, and therefore imprecise, analysis would tend to increase the levels of dependence and potentially also increase the size of clusters. There is no automatic way to tell whether a cluster arises because of imprecision in the computation of dependence or whether it is 'real'. Section 5.5 discusses the various pointer analysis settings and validates its precision. CodeSurfer's most precise pointer analysis option was used for the study. There is however greater uncertainty regarding the quality of the slices produced by the Indus Slicer for the study concerning Java programs.

9.2 Achievements

This section gives a brief summary of the contributions from this thesis. The achievements are divided into primary and additional contributions.

9.2.1 Primary

The primary contributions are those achievements that make significant contributions to the area of software engineering and help us with improving our knowledge and understanding of (coherent) dependence clusters in particular.

1. Introduction of *coherent dependence clusters* along with formalisation and extensive examples for various form of dependence clusters. This allows for a new way to study the dependence structure of a program.

2. Empirical evaluation of the frequency and size of coherent dependence clusters in production grade software (open source and industrial) demonstrates the necessity of further study into coherent clusters as such clusters are found to be common and significant in production systems.

3. Empirical evaluation of pointer analysis extensively looks at the impact of various pointer analysis settings on slicing and clustering. Codesurfer is widely used in both the research community and the industry, making this study of significant importance not only to this thesis but the research community as a whole.

4. Developed cluster visualisation tool for graph-based and interactive multi-level visualisation of dependence clusters. This tool aids a developer or maintainer in visualising and studying the structure of systems with ease. The tool has been made freely available to motivate further research.

5. Detailed study and discussion of a series of four case studies show that coherent clusters map to logical program structure and can aid a software engineer gain an understanding of this structure.

6. Introduction of framework for test coverage-based code reduction provides a compromise between the dynamic and static analysis where execution traces from entire test suites are used to remove "less" important (rarely executed) code improving the sharpness of static analysis. The average size of static slices drop by 30%, and the average drop of the largest coherent clusters is over 8% in the reduced programs.

9.2.2 Additional

Additional achievements are classified as those contributions which introduce engineering improvements over previous techniques and probing research into new areas which can be extended in the future.

1. Introduction of hashing algorithm for efficient and accurate clustering. The hashing algorithm reduces the runtime and memory requirements

for the clustering making it implementable in practice. It also gives a significantly higher precision at 97% compared to previous approach with precision of 78%.

2. A study into the evolution of barcode and correction of faults that occur within the program over a period of three years shows that presence or fixing of bugs does not show a change in the clustering profile of the program. This dispels the previous notion of dependence clusters being responsible for program faults.

3. Longitudinal study of the evolution of four programs also establishes that coherent cluster profile of programs remain surprisingly stable during system evolution. This provides further support that coherent clusters represent the core architecture of the program.

4. Identification and formalisation of inter-cluster dependence lays the foundation for using coherent clusters as the building blocks of larger program dependence structures in support for reverse engineering endeavours.

Overall, this thesis makes six primary and four additional contributions which together improve the current understanding of dependence clusters and pave way for future research in areas where dependence clusters potentially have applications.

9.3 Future Work

This thesis is the first to consider dependence clusters as being constructs that naturally occur in program and to be useful in understanding program structure. It no longer considers dependence clusters as problems as done by previous studies (Binkley and Harman [2005b], Harman et al. [2009], Binkley et al. [2010, 2008b], Black et al. [2006]), but sees them as phenomenons that occur naturally in systems. As such, this thesis considers dependence clusters to map to logical structures of programs and help software engineers in various tasks such as comprehension, understanding and re-structuring. This thesis poses and answers many research questions but also gives motivation and lays down the foundation of future work in many areas.

The first set of future work is related to extending the work done in this thesis to obtain more generalised answers through extended empirical evidence.

The work on coherent clusters analysis presented in this thesis is for 30 production programs. However, a significant number of these programs are small Unix utility programs. The largest of these programs presented is just over 62KLoc with the average size of just under 11KLoc. Future work should endeavour to extend the size of the empirical study in terms of number of subject programs and more importantly the size of programs to represent large industrial system typically of hundreds of thousands of lines of code.

One issue seen with currently extending the study with large programs has been the limitations imposed by Codesurfer in the ability to analyse large systems and the scalability of the clustering algorithm. Codesurfer imposes limitations on the size of the programs that can be analysed, a newer version of Codesurfer is due to be launched which allows analysis of larger systems. Furthermore, the clustering algorithm which now runs in a pipeline architecture based on the scheme interface of Codesurfer has to be reimplemented to improve the runtime.

This thesis also draws links between coherent clusters and program structure based on four case studies. The number of case studies and the size of the program also need to be extended to be able to better generalise the answer. This also applies to the study of the changes in clustering during software evolution. Additional human studies can also be done to ascertain the intended program structure and compare to the results obtained using clustering.

The study that looks at the relationship between software faults and dependence clusters is currently based on a single case study for the program barcode. The study was not replicated using other cases because of the difficulty in manually gathering and assessing fault data. As the scaling of Codesurfer and the clustering algorithm improves it would be possible to analyse and study larger modern systems that have bug repositories. Extending this study with modern large systems will help derive a more generalised answer.

The application of the coverage based code reduction framework to static slicing and dependence clusters found improvement in the size of static slicing and improved mapping between clusters and logical program constructs. Future studies should consider the application of the framework to other static analysis. Furthermore, the current empirical validation which applies the framework on coherent clusters can also be extended by addition of further case studies.

Finally, most of the experiments here have been on programs written in C, which should be extended to other programming paradigms and languages. The study into the existence of coherent clusters in Java programs should also be replicated using other slicing tools such as Wala to validate results.

9.4 Summary

Previous work has deemed dependence clusters to be problematic as they inhibit program understanding and maintenance. This thesis views them in a new light, it introduces and evaluates a specialised form of dependence cluster: the *coherent cluster*. Such clusters have vertices that share the same internal and external dependencies. The thesis shows that such clusters are not necessarily problems but rather their analysis can aid an engineer understand program components and their interactions. This thesis is one of the first in the area of dependence clusters to suggest that dependence clusters (coherent clusters) are not problematic but represent program structure and give evidence to that cause. Developers can exploit knowledge of coherent clusters as such clusters represent logical constructs of the system and their interaction.

This thesis presents new approximations that support the efficient and accurate identification of coherent clusters. Empirical evaluation finds that 23 of the 30 subject programs have at least one large coherent cluster. A series of four case studies illustrate that coherent clusters map to a logical functional decomposition and can be used to depict the structure of a program. In all four case studies, coherent clusters map to subsystems, each of which is responsible for implementing concise functionality. As side-effects of the study, we find that the visualisation of coherent clusters can identify potential structural problems as well as refactoring opportunities.

This thesis provides further support for the view that coherent clusters map to logical program constructs through longitudinal study of four case studies. The longitudinal studies show that coherent clusters remain stable during system evolution as they depict the core architecture of the program. Furthermore, the thesis presents a study on how bug fixes relate to the presence of coherent clusters, and finds no relationship between program faults and coherent clusters.

Finally, this thesis presents a code reduction framework based on coverage information. The aim of the framework is to improve the “sharpness” of static

analysis to alleviate some of the inherent conservatism associated with static analysis. The framework allows for a blend between static and dynamic analysis by removing code that is regarded as "less" important, thereby improving the results of the static analysis. In particular we found reduction in size of static slices and better mapping between clusters and logical program constructs.

References

Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 746–755. ACM Press, 2011.

Rahmat Adnan, Bas Graaf, Arie van Deursen, and Joost Zonneveld. Using Cluster Analysis to Improve the Design of Component Interfaces. In *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 383–386. IEEE, September 2008.

Hiralal Agrawal, Richard A DeMillo, and Eugene H Spafford. Efficient debugging with slicing and backtracking. *Software Practice & Experience*, 23(6): 589–616, 1993.

Michael R. Anderberg. *Cluster analysis for applications*. Academic Press, New York :, 1973.

Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

Paul Anderson and Tim Teitelbaum. Software inspection using CodeSurfer. In *First Workshop on Inspection in Software Engineering*, pages 1–9, 2001.

P Andritsos and V Tzerpos. Information-theoretic software clustering. *IEEE Transactions on Software Engineering*, 31(2):150–165, 2005.

N. Anquetil and T.C. Lethbridge. Comparative study of clustering algorithms and abstract representations for software remodularisation. *IEE Proceedings – Software*, 150(3):185, 2003.

Nicolas Anquetil and Timothy Lethbridge. File clustering using naming conventions for legacy systems. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research*, page 2, Toronto, Ontario, Canada, October 1997. IBM Press.

Nicolas Anquetil and Timothy C. Lethbridge. Extracting concepts from file names; a new file clustering criterion. In *Proceedings of the 20th International Conference on Software Engineering*, pages 84–93. IEEE Comput. Soc, 1998.

Nicolas Anquetil and Timothy C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11(3):201–221, May 1999.

Nicolas Anquetil, Cedric Fourrier, and Timothy C. Lethbridge. Experiments with hierarchical clustering algorithms as software remodularization methods. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, page 235, 1999.

M.N. Armstrong and C. Trudeau. Evaluating architectural extractors. In *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 30–39, Honolulu, Hawaii, 1998. IEEE Comput. Soc.

Nadim Asif, Faisal Shahzad, Najia Saher, and Waseem Nazar. Clustering the Source Code. *WSEAS Transactions on Computers*, 8(12):1835–1844, 2009.

Marla J. Baker and Stephen G. Eick. Space-filling software visualization. *Journal of Visual Languages & Computing*, 6(2):119–133, 1995.

Tibor Bakota, Péter Hegedus, Peter Kortvelyesi, Rudolf Ferenc, and Tibor Gyimóthy. A probabilistic software quality model. In *27th IEEE International Conference on Software Maintenance (ICSM)*, pages 243–252. IEEE, 2011.

T. Ball and S.G. Eick. Visualizing program slices. *Proceedings of 1994 IEEE Symposium on Visual Languages*, pages 288–295, 1994.

Thomas Ball and Susan Horwitz. Slicing Programs with Arbitrary Control Flow. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 206–222, 1993.

Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *Proceedings of the Sixth Joint Eurographics – IEEE TCVG Conference on Visualization*, pages 261–266, 2004.

Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. In *International AAAI Conference on Weblogs and Social Media*. AAAI Press, 2009.

Fabian Beck. Improving Software Clustering with Evolutionary Data. Technical report, Universität Trier, 2009.

L. A. Belady and C. J. Evangelisti. System partitioning and its measure. *Journal of Systems and Software*, 2(1):23–29, February 1981.

Árpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Toth, Tibor Gyimóthy, and Václav Rajlich. Computation of static execute after relation with applications to software maintenance. In 23^{rd} *IEEE International Conference*

on Software Maintenance, pages 295–304. IEEE Computer Society Press, October 2007.

Arpad Beszedes, Lajos Schrettner, Béla Csaba, Tamás Gergely, Judit Jász, and Tibor Gyimóthy. Empirical investigation of sea-based dependence cluster properties. In *13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–10. IEEE, 2013.

Dirk Beyer. CCVisu: automatic visual software decomposition. In *Companion of the 30th International Conference on Software Engineering*, pages 967–968. ACM Press, 2008.

D. Binkley. Source code analysis: A road map. In *Future of Software Engineering*, pages 104–119. IEEE Computer Society Press, 2007.

Dave Binkley. Dependence cluster causes. In *Scalable Program Analysis*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Germany.

David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *IEEE International Conference on Software Maintenance*, pages 44–53. IEEE Computer Society Press, 2003.

David Binkley and Mark Harman. Forward slices are smaller than backward slices. In *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 15–24, 2005a.

David Binkley and Mark Harman. Locating dependence clusters and dependence pollution. In 21^{st} *IEEE International Conference on Software Maintenance*, pages 177–186. IEEE Computer Society Press, 2005b.

David Binkley and Mark Harman. Identifying 'linchpin vertices' that cause large dependence clusters. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 89–98, 2009.

David Binkley, Nicolas Gold, Mark Harman, Zheng Li, and Kiarash Mahdavi. An empirical study of the relationship between the concepts expressed in source code and dependence. *Journal of Systems and Software*, 81(12):2287–2298, 2008a.

David Binkley, Nicolas Gold, Mark Harman, Zheng Li, Kiarash Mahdavi, and Joachim Wegener. Dependence anti patterns. In 4^{th} *International ERCIM Workshop on Software Evolution and Evolvability (Evol'08)*, pages 25–34, 2008b.

David Binkley, Mark Harman, Youssef Hassoun, Syed Islam, and Zheng Li. Assessing the impact of global variables on program dependence and dependence clusters. *Journal of Systems and Software*, 83(1):96–107, 2010.

David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Zheng Li. Efficient identification of linchpin vertices in dependence clusters. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 35 (2):1–35, July 2013a.

David Binkley, Nicolas Gold, Mark Harman, Jens Krinke, and Shin Yoo. Observation-based slicing. Technical report, Department of Computer Science, University College London, 2013b.

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

David Wendell Binkley, Mark Harman, and Jens Krinke. Empirical study of optimization techniques for massive slicing. *ACM Transactions on Programming Languages and Systems*, 30:3:1–3:33, 2007.

Sue Black, Steve Counsell, Tracy Hall, and Paul Wernick. Using program slicing to identify faults in software. In *Beyond Program Slicing*, number 05451 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

Sue Black, Steve Counsell, Tracy Hall, and David Bowes. Fault analysis in OSS based on program slicing metrics. In *EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 3–10. IEEE Computer Society Press, 2009.

Susan E. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(4):263–279, 2001.

Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

Immanuel M Bomze, Marco Budinich, Panos M Pardalos, and Marcello Pelillo. The maximum clique problem. In *Handbook of combinatorial optimization*, pages 1–74. Springer US, 1999.

E. Burd, M. Munro, and C. Wezeman. Extracting reusable modules from legacy code: considering the issues of module granularity. In *Proceedings of the 4th Working Conference on Reverse Engineering*, pages 189–196. IEEE Comput. Soc. Press, 1996.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224, 2008.

Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40(11 and 12):595–607, 1998.

Yih-Farn Chen, Michael Nishimoto, and C. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

Yih-Farn Chen, Glenn Fowler, Eleftherios Koutsofios, and Ryan Wallach. Ciao: a graphical navigator for software and document repositories. In *Proceedings of International Conference on Software Maintenance*, pages 66–75. IEEE Comput. Soc. Press, 1995.

S.C. Choi and Walt Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, 7(1):66–71, 1990.

A. Cimitile, A. De Lucia, G.A. Di Lucca, and A.R. Fasolino. Identifying objects in legacy systems. In *Proceedings of the Fifth International Workshop on Program Comprehension*, pages 138–147. IEEE Comput. Soc. Press, 1997.

B. Csaba, L. Schrettner, A. Beszedes, J. Jasz, P. Hegedus, and T. Gyimothy. Relating clusterization measures and software quality. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 345–348, 2013.

John Davey and Elizabeth Burd. Evaluating the suitability of data clustering for software remodularization. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 268, Washington, DC, USA, 2000. IEEE Computer Society.

Stephan Diehl. Software visualization. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 718–719, New York, NY, USA, 2005. ACM Press.

Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

D. Doval, Spiros Mancoridis, and Brian S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of the Ninth International Workshop Software Technology and Engineering Practice*, pages 73–81. IEEE Comput. Soc, 1999.

Stephane Ducasse and Damien Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.

S.C. Eick, J.L. Steffen, and E.E. Sumner. Seesoft – A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18 (11):957–968, 1992.

T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, March 2003.

Amr Elssamadisy and Gregory Schalliol. Recognizing and responding to "bad smells" in extreme programming. In *International Conference on Software Engineering*, pages 617–622. ACM Press, 2002.

Michael D. Ernst. Static and dynamic analysis: synergy and duality. In *ICSE Workshop on Dynamic Analysis*, pages 6–9. ACM, 2003.

B. S. Everitt. *Cluster analysis*. Heinemann Educational for Social Science Research Council, London, 1974.

Manuel Fahndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 85–96. ACM Press, 1998.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.

Jon Froehlich and Paul Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 387–396, Washington, DC, USA, 2004. IEEE Computer Society.

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

Keith Brian Gallagher. Some notes on interprocedural program slicing. In *Fourth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 36–42. IEEE Comput. Soc, 2004.

G. C. Gannod and B. H. C. Cheng. A formal automated approach for reverse engineering programs with pointers. In *ASE '97: Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, page 219, Washington, DC, USA, 1997. IEEE Computer Society.

Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

S. Grant, J.R. Cordy, and D. Skillicorn. Automated concept location using independent component analysis. In *15th Working Conference on Reverse Engineering*, pages 138–142, 2008.

Rajiv Gupta, Mary Lou Soffa, and John Howard. Hybrid slicing: integrating dynamic information with static analysis. *ACM Transactions on Software Engineering Methodology*, 6(4):370–397, October 1997.

Ákos Hajnal and István Forgács. A demand-driven approach to slicing legacy COBOL systems. *Journal of Software: Evolution and Process*, 24(1):67–82, 2011.

James Hamilton and Sebastian Danicic. Dependence communities in source code. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 579–582, 2012.

Clemens Hammacher. *Design and Implementation of an Efficient Dynamic Slicer for Java*. PhD thesis, Dept. of Computer Science, Saarland University, 2008.

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

Mark Harman, David Wendell Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, October 2003.

Mark Harman, Stephen Swift, and Kiarash Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In *Proceedings of the 2005 conference on Genetic and evolutionary computation – GECCO '05*, page 1029, New York, New York, USA, 2005. ACM Press.

Mark Harman, Keith Gallagher, David Binkley, Nicolas Gold, and Jens Krinke. Dependence clusters in source code. *ACM Transactions on Programming Languages and Systems*, 32(1):1–33, 2009.

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.

Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

D.H. Hutchens and V.R. Basili. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering*, SE-11(8):749–757, August 1985.

Syed Islam, Jens Krinke, and David Binkley. Dependence cluster visualization. In *SoftVis'10: 5th ACM/IEEE Symposium on Software Visualization*, pages 93–102. ACM Press, 2010a.

Syed Islam, Jens Krinke, David Binkley, and Mark Harman. Coherent dependence clusters. In *PASTE '10: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 53–60. ACM Press, 2010b.

Syed Islam, Jens Krinke, David Binkley, and Mark Harman. Coherent clusters in source code. *Journal of Systems and Software*, 88(0):1 – 24, 2014.

D Jackson, KM Somers, and HH Harvey. Similarity coefficients: Measures of co-occurrence and association or simply measures of occurrence? *American Naturalist*, 133(3):436–453, 1989.

A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, September 1999.

J. Jasz, A. Beszedes, T. Gyimothy, and V. Rajlich. Static execute after/before as a replacement of traditional software dependencies. In *IEEE International Conference on Software Maintenance*, pages 137–146, 2008.

J. Jasz, L. Schrettner, A. Beszedes, C. Osztrogonac, and T. Gyimothy. Impact analysis using static execute after in webkit. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 95–104, 2012.

Ganeshan Jayaraman, Venkatesh Ranganath, and John Hatcliff. Kaveri: Delivering the Indus Java program slicer to Eclipse. *Fundamental Approaches to Software Engineering*, 3442:269–272, 2005.

Yue Jia and Mark Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Testing: Academic & Industrial Conference - Practice and Research Techniques*, pages 94–98. IEEE Computer Society, 2008.

Tao Jiang, Nicolas Gold, Mark Harman, and Zheng Li. Locating dependence structures using search-based slicing. *Information and Software Technology*, 50(12):1189–1209, 2008.

James A Jones, Mary Jean Harrold, and John T Stasko. Visualization for fault localization. In *Proceedings of the Workshop on Software Visualization, 23rd International Conference on Software Engineering*, May 2001.

Shariar Kazem, Ali Asghar Pourhaji And Lotfi. A Modified Genetic Algorithm for Software Clustering Problem. In *AIC'06: Proceedings of the 6th WSEAS International Conference on Applied Informatics and Communications*, volume 2006, pages 306–311, Elounda, Greece, 2006. World Scientific and Engineering Academy and Society (WSEAS).

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

Rainer Koschke. Atomic architectural component recovery for program understanding and evolution. In *International Conference on Software Maintenance*, pages 478–481, 2002.

Rainer Koschke and Thomas Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Intl. Workshop on Program Comprehension*, page 201. IEEE Computer Society, 2000.

Jens Krinke. Static slicing of threaded programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 35–42, 1998.

Jens Krinke. Evaluating context-sensitive slicing and chopping. In *IEEE International Conference on Software Maintenance*, pages 22–31. IEEE Computer Society Press, 2002.

Jens Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of the 9th European Software Engineering Conference*, pages 178–187. ACM Press, 2003.

Jens Krinke. Effects of context on program slicing. *Journal of Systems and Software*, 79(9):1249–1260, 2006.

Adrian Kuhn, Stephane Ducasse, and Tudor Girba. Enriching Reverse Engineering with Semantic Clustering. In *12th Working Conference on Reverse Engineering (WCRE'05)*, pages 133–142. IEEE, 2005.

Thomas Kunz. Developing a measure for process cluster evaluation. Technical Report TI-2/93, Institut fur Theoretische Informatik, Fackbereich Informatik, Technische Hochschule Darmstadt, 1993.

Arun Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, 36(3):211–231, March 1997.

Arun Lakhotia and John M. Gravley. Toward experimental evaluation of subsystem classification recovery techniques. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 262–269, Toronto, Ontario, Canada, 1995. IEEE Comput. Soc. Press.

Kiran Lakhotia, Mark Harman, and Hamilton Gross. Austin: A tool for search based software testing for the C language and its evaluation on deployed automotive systems. In *2nd International Symposium on Search Based Software Engineering*, pages 101–110, 2010.

Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *25th International Conference on Software Engineering*, pages 308–318, 2003.

Steffen Lehnert. A review of software change impact analysis. Technical report, Ilmenau University of Technology, 2011.

Zheng Li. Identifying high-level dependence structures using slice-based dependence analysis. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 457–460. IEEE, 2009.

Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *19th International conference on Software engineering*, pages 349–359. ACM Press, 1997.

Panos E. Livadas and Theodore Johnson. A new approach to finding objects in programs. *Journal of Software Maintenance: Research and Practice*, 6(5): 249–260, September 1994.

C Lung, Marzia Zaman, and Amit Nandi. Applications of clustering techniques to software partitioning, recovery and restructuring. *Journal of Systems and Software*, 73(2):227–244, October 2004.

Chung-Horng Lung, Xia Xu, Marzia Zaman, and Anand Srinivasan. Program restructuring using clustering techniques. *Journal of Systems and Software*, 79(9):1261–1279, September 2006.

Kiarash Mahdavi, Mark Harman, and Robert M. Hierons. A multiple hill climbing approach to software module clustering. In *International Conference on Software Maintenance, 2003*, pages 315–324. IEEE Comput. Soc, 2003.

Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 103–112, Toronto, Ontario, Canada, 2001. IEEE Comput. Soc.

Jonathan I. Maletic, Andrian Marcus, and Michael L. Collard. A task oriented view of software visualization. In *VISSOFT '02: Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, page 32, Washington, DC, USA, 2002. IEEE Computer Society.

Spiros Mancoridis and R.C. Holt. Recovering the structure of software systems using tube graph interconnection clustering. In *Proceedings of 12th International Conference on Software Maintenance*, pages 23–32. IEEE Comput. Soc. Press, 1996.

Spiros Mancoridis, Brian Mitchell, C. Rorres, Yih-Farn Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the 6th International Workshop on Program Comprehension*, volume 98, pages 45–52. IEEE Comput. Soc, 1998.

Spiros Mancoridis, Brian Mitchell, Yih-Farn Chen, and E.Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *IEEE International Conference on Software Maintenance*, pages 50–99, Oxford, England, 1999. IEEE.

Spiros Mancoridis, T.S. Souder, E.R. Gansner, and J.L. Korn. REportal: a Web-based portal site for reverse engineering. In *Proceedings of Eighth Working Conference on Reverse Engineering*, pages 221–230. IEEE Comput. Soc, 2001.

O. Maqbool and H. A. Babri. The weighted combined algorithm: A linkage algorithm for software clustering. In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, page 15, Washington, DC, USA, 2004. IEEE Computer Society.

O. Maqbool and H.A. Babri. Automated software clustering: An insight using cluster labels. *Journal of Systems and Software*, 79(11):1632 – 1648, 2006. Software Cybernetics.

Onaiza Maqbool and Haroon Babri. Hierarchical Clustering for Software Architecture Recovery. *IEEE Transactions on Software Engineering*, 33(11): 759–780, November 2007.

Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3D representations for software visualization. In *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*, New York, NY, USA, 2003. ACM Press.

Brian S. Mitchell. Clustering Software Systems to Identify Subsystem Structures. Technical report, Technische Universität Ilmenau, 2004.

Brian S. Mitchell and Spiros Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 744, Washington, DC, USA, 2001a. IEEE Computer Society.

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

Brian S. Mitchell and Spiros Mancoridis. On the evaluation of the Bunch search-based software modularization algorithm. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 12(1):77–93, June 2007.

B.S. Mitchell and S. Mancoridis. CRAFT: a framework for evaluating software clustering results in the absence of benchmark decompositions. In *Proceedings of Eighth Working Conference on Reverse Engineering*, pages 93–102, Stuttgart, Germany, 2001b. IEEE Comput. Soc.

Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

Hausi A. Muller and K. Klashinsky. Rigi: a system for programming-in-the-large. In *Proceedings of 11th International Conference on Software Engineering*, pages 80–86. IEEE Comput. Soc. Press, 1988.

Hausi A. Muller and J.S. Uhl. Composing subsystem structures using (k,2)-partite graphs. In *Conference on Software Maintenance*, pages 12–19, 1990.

Hausi A. Muller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse-engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

Gail C. Murphy, David Notkin, and Kevin Sullivan. Software Reflexion Models: Bridgin the Gap between Source and High-Level Models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28, New York, New York, USA, 1995. ACM Press.

A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue. Call-mark slicing: an efficient and economical way of reducing slice. In *International Conference on Software Engineering*, pages 422–431, 1999.

Chap-Liong Ong. *Class and object extraction from imperative code*. PhD thesis, University of Minnesota, Minneapolis, MN, USA, 1994.

Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in software development environments. *Proceedings of the ACM SIGSOFT-/SIGPLAN Software Engineering Symposium on Practical Software Development Environmt, SIGPLAN Notices*, 19(5):177–184, 1984.

Thomas Panas, Joanes Lundberg, and Welf Lowe. Reuse in reverse engineering. In *IWPC '04: Proceedings of the 12th IEEE International Workshop on Program Comprehension*, page 52, Washington, DC, USA, 2004. IEEE Computer Society.

Venkatesh Ranganath and John Hatcliff. Slicing concurrent Java programs using Indus and Kaveri. *International Journal on Software Tools for Technology Transfer*, 9(5–6):489–504, 2007.

S P Reiss. Bee/Hive: A software visualization back end. *Proceedings of ICSE 2001 Workshop on Software Visualization, Toronto*, pages 44–48, 2001a.

Steven P. Reiss. An overview of BLOOM. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 2–5, New York, NY, USA, 2001b. ACM Press.

Xiaoxia Ren, Barbara G. Ryder, Maximilian Störzer, and Frank Tip. Chianti: a change impact analysis tool for Java programs. In *27th International Conference on Software Engineering*, pages 664–665. ACM Press, 2005.

Xiaoxia Ren, Ophelia Chesley, and Barbara G. Ryder. Identifying failure causes in Java programs: An application of change impact analysis. *IEEE Transactions on Software Engineering*, 32(9):718–732, 2006.

Thomas Reps and T. Bricker. Semantics-based program integration: Illustrating interference in interfering versions of programs. In *Proceedings of the Second International Workshop on Software Configuration Management*, pages 46–55, Princeton, New Jersey, October 1989.

F. Ricca, P. Tonella, C. Girardi, and E. Pianta. An empirical study on keyword-based web site clustering. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 204–213. IEEE, 2004.

M.P. Robillard and G.C. Murphy. Automatically inferring concern code from program investigation activities. In *18th IEEE International Conference on Automated Software Engineering*, pages 225–234, 2003.

G. Robles, S. Koch, and J.M. Gonzalez-Barahona. Remote analysis and measurement of libre software systems by means of the CVSAnalY tool. In *Proceedings of Second International Workshop on Remote Analysis and Measurement of Software Systems*, pages 51–55. IEE, 2004.

H.C. Romesburg. *Cluster Analysis for Researchers*. Lifetime Learning Publications, Belmont, California, 1984.

M. Saeed, O. Maqbool, H. A. Babri, S. Z. Hassan, and S. M. Sarwar. Software clustering techniques and the use of combined algorithm. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 301, Washington, DC, USA, 2003. IEEE Computer Society.

Leo Savernik. Entwicklung eines automatischen Verfahrens zur Auflösung statischer zyklischer Abhängigkeiten in Softwaresystemen (in german). In *Software Engineering 2007 – Beiträge zu den Workshops*, volume 106 of *LNI*, pages 357–360. GI, 2007.

L. Schrettner, J. Jasz, T. Gergely, A. Beszedes, and T. Gyimothy. Impact analysis in the presence of dependence clusters using static execute after in webkit. In *12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 24–33, 2012.

Robert W. Schwanke. An Intelligent Tool For Re-engineering Software Modularity. In *13th international conference on Software engineering*, pages Pages: 83 – 92, Austin, Texas, United States, 1991. IEEE Computer Society Press.

Robert W. Schwanke and Michael A. Platoff. Cross references are features. In *Machine Learning: From Theory to Applications – Cooperative Research at Siemens and MIT*, pages 107–123, London, UK, 1993. Springer-Verlag.

Richard W. Selby and Victor R. Basili. Analyzing error-prone system structure. *IEEE Trans. Softw. Eng.*, 17(2):141–152, 1991.

Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 16–34. Springer Berlin Heidelberg, 1997.

Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 336, Washington, DC, USA, 1996. IEEE Computer Society.

M. Shtern and V. Tzerpos. A framework for the comparison of nested software decompositions. In *11th Working Conference on Reverse Engineering*, pages 284–292. IEEE Comput. Soc, 2004.

Josep Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surveys*, 44(3):12:1–12:41, June 2012.

Ian Sommerville. *Software engineering (5th ed.)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

Michael Stiff and Thomas Reps. Identifying modules via concept analysis. In *International Conference On Software Maintenance*, pages 170–179, Bari, 1997. IEEE Computer Society.

Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Muller. Rigi: a visualization environment for reverse engineering. In *Proceedings of the 19th international conference on Software engineering*, pages 606–607. ACM Press, 1997.

M.D. Storey, K. Wong, and H.A. Muller. How do program understanding tools affect how programmers understand programs? *Proceedings of the Fourth Working Conference on Reverse Engineering*, 36(2–3):183–207, 2000.

Attila Szegedi, Tamás Gergely, Árpád Beszédes, Tibor Gyimóthy, and Gabriella Tóth. Verifying the concept of union slices on Java programs. In 11^{th} *European Conference on Software Maintenance and Reengineering*, pages 233–242, 2007.

T. Takada, F. Ohata, and K. Inoue. Dependence-cache slicing: a program slicing method using lightweight dynamic information. In *10th International Workshop on Program Comprehension.*, pages 169–177, 2002.

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

P. Tonella, F. Ricca, E. Pianta, C. Girardi, G. Di Lucca, A. R. Fasolino, and P. Tramontana. Evaluation methods for web application clustering. In *5th International Workshop on Web Site Evolution*, 2003.

V. Tzerpos. Software Clustering Based on Dynamic Dependencies. *Ninth European Conference on Software Maintenance and Reengineering*, pages 124–133, 2005.

V. Tzerpos and R.C. Holt. Software botryology. Automatic clustering of software systems. In *Proceedings Ninth International Workshop on Database and Expert Systems Applications (Cat. No.98EX130)*, pages 811–818. IEEE Comput. Soc, 1998.

Vassilios Tzerpos and R.C. Holt. MoJo: a distance metric for software clusterings. In *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, pages 187–193. IEEE Comput. Soc, 1999.

Vassilios Tzerpos and R.C. Holt. ACDC: an algorithm for comprehension-driven clustering. In *Proceedings Seventh Working Conference on Reverse Engineering*, volume pages, pages 258–267, Brisbane, Australia, 2000a. IEEE Comput. Soc.

Vassilios Tzerpos and Richard C. Holt. On the stability of software clustering algorithms. In *IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension*, page 211, Washington, DC, USA, 2000b. IEEE Computer Society.

Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying Java bytecode for analyses and transformations, 1998.

Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 21st international conference on Software engineering – ICSE '99*, pages 246–255, New York, New York, USA, 1999. ACM Press.

Adam Vanya, Lennart Hofland, Steven Klusener, Piërre Van De Laar, and Hans Van Vliet. Assessing Software Archives with Evolutionary Clusters. In *2008 The 16th IEEE International Conference on Program Comprehension*, pages 192–201. IEEE, June 2008.

Lucian Voinea, Alex Telea, and Jarke J. van Wijk. CVSscan: visualization of code evolution. In *SoftVis '05: Proceedings of the 2005 ACM Symposium on Software Visualization*, pages 47–56, New York, NY, USA, 2005. ACM Press.

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

Mark Weiser. Program slicing. In 5^{th} *International Conference on Software Engineering*, pages 439–449, 1981.

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

Zhihua Wen and Vassilios Tzerpos. An effectiveness measure for software clustering algorithms. In *12th IEEE International Workshop on Program Comprehension*, pages 194–203. IEEE, 2004.

Zhihua Wen and Vassilios Tzerpos. Software clustering based on omnipresent object detection. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 269–278, Washington, DC, USA, 2005. IEEE Computer Society.

D. A. Wheeler. SLOC count user's guide. *http://www.dwheeler.com/sloccount/sloccount.html.*, 2004.

T.A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43. IEEE Comput. Soc, 1997.

N. Wilde, J.A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Conference on Software Maintenance*, pages 200–205, 1992.

Norman Wilde and Michael C. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

Claes Wohlin, Martin HÃűst, and Kennet Henningsson. Empirical research methods in software engineering. In *Empirical Methods and Studies in Software Engineering*, volume 2765 of *Lecture Notes in Computer Science*, pages 7–23. Springer Berlin Heidelberg, 2003.

Jingwei Wu, Ahmed E. Hassan, and Richard C. Holt. Comparison of clustering algorithms in the context of software evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 525–535, Washington, DC, USA, 2005. IEEE Computer Society.

Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30 (2):1–36, 2005.

Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 16(3):645–78, May 2005.

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

William Griswold Yoshikiyo, William G. Griswold, Y Yoshikiyo Kato, and Jimmy J. Yuan. Aspect Browser: Tool support for managing dispersed aspects. In *First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems—OOPSLA 99*, 1999.