Goldsmiths, University of London
Department of Computing

Dependence Communities in Source Code

James Alexander George Hamilton

A thesis submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

July 2013

Supervised by Dr Sebastian Danicic

Declaration

I, James Hamilton, 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.

Signature ____________________

Abstract

Dependence between components in natural systems is a well studied phenomenon in the form of biological and social networks. The concept of community structure arises from the analysis of social networks and has successfully been applied to complex networks in other fields such as biology, physics and computing.

We provide empirical evidence that dependence between statements in source code gives rise to community structure. This leads to the introduction of the concept of *dependence communities* in software and we provide evidence that they reflect the semantic concerns of a program.

Current definitions of sliced-based cohesion and coupling metrics are not defined for procedures which do not have clearly defined output variables and definitions of output variable vary from study-to-study. We solve these problems by introducing corresponding new, more efficient forms of slice-based metrics in terms of maximal slices. We show that there is a strong correlation between these new metrics and the old metrics computed using output variables.

We conduct an investigation into *dependence clusters* which are closely related to dependence communities. We undertake an empirical study using definitions of dependence clusters from previous studies and show that, while programs do contain large dependence clusters, over 75% of these are not 'true' dependence clusters.

We bring together the main elements of the thesis in a study of software quality, investigating their interrelated nature. We show that procedures that are members of multiple communities have a low cohesion, programs with higher coupling have larger dependence communities, programs with large dependence clusters also have large dependence communities and programs with high modularity have low coupling.

Dependence communities and maximal-slice-based metrics have a huge number of potential applications including program comprehension, maintenance, debugging, refactoring, testing and software protection.

Acknowledgements

First and foremost I'd like to thank my main supervisor, Sebastian Danicic, for all his help and guidance over the course of my studies. He has been a brilliant supervisor and I have benefitted greatly from his broad knowledge and the many discussions that helped shape this thesis. I would also like to thank his wife, Aurora, for her delicious meals.

I would also like to thank my second supervisor, Mark Bishop, for all his help.

Many thanks to all those who I have had discussions about the subject of the thesis including, but not limited to (and in no particular order), Dave Binkley, John Howroyd, Jens Krinke, Syed Islam, Richard Barraclough and Lahcen Ouarbya.

I would like to thank Goldsmiths and the Engineering and Physical Sciences Research Council for providing funding without which I would not have been able to continue my studies to this level.

I would like to thank GrammaTech for providing a copy of CodeSurfer upon which the empirical studies in this thesis are based.

Finally, I would like to thank my family, friends and girlfriend for all their support and encouragement over the years.

To Jusna

Publications

Refereed Conference Papers

2012 **Dependence Communities in Source Code** (James Hamilton, Sebastian Danicic), *In Proceedings of the IEEE 28th International Conference on Software Maintenance, Early Research Achievements Track*, IEEE, 2012.

2011 **A Survey of Static Software Watermarking** (James Hamilton, Sebastian Danicic), *In Proceedings of the World Congress on Internet Security 2011*, IEEE, 2011.

2010 **An Evaluation of Static Java Bytecode Watermarking** (James Hamilton, Sebastian Danicic), *In Proceedings of the International Conference on Computer Science and Applications (ICCSA10), The World Congress on Engineering and Computer Science (WCECS10)*, 2010. Winner of The Best Student Paper Award.

2009 **An Evaluation of Current Java Bytecode Decompilers** (James Hamilton, Sebastian Danicic), *In Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, IEEE, volume 0, 2009.

Refereed Articles

Under Review **Maximal-Slice-Based Cohesion and Coupling Metrics** (James Hamilton, Sebastian Danicic), *Journal of Software: Evolution and Process (JSEP) SPECIAL ISSUE on Source Code Analysis and Manipulation*, John Wiley & Sons, Ltd.

2011 **An Evaluation of the Resilience of Static Java Bytecode Watermarks Against Distortive Attacks** (James Hamilton, Sebastian Danicic), *In IAENG International Journal of Computer Science*, volume 38, 2011.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Dependence between software system components is a relation that determines how components depend on and are affected by each other. Dependence analysis underpins many activities in software engineering such as software maintenance [124], testing [154], reverse engineering [71], program understanding [194], refactoring [15], software protection [39] and plagiarism detection [121].

The concept of community structure arises from the analysis of social networks in sociology [183], where dependence occurs in the form of relationships between people. Community structure can be found in many real world graphs other than social networks [155]. Informally, a graph is said to have community structure if it can be grouped into sets of nodes that are densely connected internally to each other, but sparsely connected to other groups [68].

Software systems can be modelled as complex networks where nodes are typically components of a system and edges represent dependencies or interactions between components [187]. Software graphs have been shown to have the non-trivial topological features of complex networks [114, 139, 156, 181], such as scale-free geometry [30, 42, 92, 96, 114, 123, 178, 181, 188] and community structure [153, 180, 191].

The task of managing large software projects is becoming more challenging due to the ever increasing size and complexity of industrial strength software systems. Software clustering approaches can have an important role in the task of understanding large, complex software systems by automatically decomposing them into smaller, easier-to-manage subsystems. Previous research in the area has focused on the clustering of high-level components in a software system to recover a modular structure or re-modularise software. The Bunch tool [129, 133], for example, works on a Module Dependency Graph (MDG) which includes high-level system components such as Java classes or C files that are connected due to dependence. The purpose of the tool is to help maintain and understand existing software by clustering related modules together.

We believe that for a clustering technique to be useful in software engineering it must provide what we call *Semantic Separation*, i.e. the clusters must, in some sense, partition the system into its different *functionalities.*

Dependence between statements can be modelled as a complex network built of program *slices* [186]; community structure has not previously been studied at this statement-level. The study of statement-level community structure has a huge number of potential applications including program comprehension, maintenance, debugging, software metrics, refactoring, testing and software protection. A strict form of statement-level community within a software network has previously been studied in the form of *dependence clusters* [16].

Software metrics provide another view of the dependence in a program and have long been used in an attempt to quantify code quality, for example for quantifying software development resources [55], refactoring [170], improving software [117] and finding bugs [152]. A software metric is a quantitative measure of some aspect of code, ranging from crude measures such as lines of code to measures of relationships between software components such as cohesion and coupling [192]. Slice-based metrics are underpinned by program slicing which provides the dependence analysis upon which values can be assigned to quantify the quality of source-code.

Cohesion and coupling metrics are closely related to the concept of communities in software networks. Low coupling between two components would suggest that those components are in two separate communities; while high cohesion between components would suggest that the components are part of the same community.

1.1 Contributions

The major contributions of this thesis involve the study of software dependence by modelling software as complex networks. Along with the System Dependence Graph (SDG) [56], we use the notion of a Backward Slice Graph (BSG). A BSG is a graph of a program's backward slices, such that a is connected to b if $b \in Slice(a)$.

A graph-based representation of software allows us to apply existing graph-theoretic techniques commonly used in other fields, such as community detection. We introduce the concept of *dependence communities*, redefine slice-based metrics in terms of *maximal slices* and undertake an investigation into *dependence clusters*.

The contributions of this thesis are:

1. The concept of *dependence communities* in software. We apply a well known community detection algorithm to BSGs. This algorithm is based on modularity maximisation and has previously been applied successfully to large networks in other fields such as physics, biology and sociology. We provide empirical evidence that dependence between statements in software gives rise to community structure. We give examples which suggest that the dependence communities reflect the semantic concerns of a program.

2. A new, efficient form of slice-based metrics based on maximal slices. Previously, slice-based software metrics have been defined using the ambiguous concept of output variables. It is difficult to define output variables and previous studies have used different definitions. We take a different approach – using maximal slices as a basis for calculating slice-based metrics. We show that there is a strong correlation between slice-based metrics as previously computed and our new metrics. Furthermore, we introduce an approximation which, not only is more efficient than calculating maximal-slice-based metrics but is also more efficient than calculating output variable based metrics.

3. An investigation into *dependence clusters* and their approximations used in previous studies. Continuing our graph-based approach in this thesis, we redefine dependence clusters as maximal cliques within a BSG, and think of them as stricter forms of dependence communities. We conduct an empirical study using definitions of dependence clusters from previous studies. The results provide further evidence of the existence of large dependence clusters in software. We also show, however, that over 75% of the dependence clusters found are not, in fact, 'true' dependence clusters.

4. A study of sofware quality. We discuss the inter-connected nature of the three main topics of this thesis. Are they all different aspects of the same quality measurement? Do programs with large dependence clusters also have large dependence communities? We show that procedures that are members of multiple communities have a low cohesion, programs with higher coupling have larger dependence communities and clusters, programs with large dependence clusters also have large dependence communities and programs with high modularity have low coupling.

1.2 Structure of the Thesis

- Chapter 2 (Background) introduces the background of the subjects of this thesis; namely, previous work in modelling software as complex networks, community detection, program slicing, dependence clusters and software metrics.
- Chapter 3 (Dependence Communities) provides empirical evidence which shows that dependence communities exist in software networks. We apply a well known community detection algorithm to BSGs and show that dependence communities tend to reflect the functional concerns of a program.
- Chapter 4 (Maximal-Slice-Based Cohesion and Coupling Metrics) introduces maximal-slice-based metrics as an alternative to output variable based metrics. We show that there is a strong correlation between slice-based metrics as previously computed and our new metrics. We introduce an efficient approximation to our new maximal-slice-based metrics which is also faster than computing output variable based metrics.
- Chapter 5 (Dependence Clusters) confirms the results of previous dependence cluster studies by providing evidence of the existence of large dependence clusters in the set of programs used in chapters 3 and 4. We show that over 75% of dependence clusters found in our set of programs, calculated as defined in studies, are not 'true' dependence clusters.
- Chapter 6 (A Study Of Software Quality) discusses the main themes of this thesis and their relationships, as measures of software quality.
- Chapter 7 (Conclusion) reviews the findings of this thesis.
- Chapter 8 (Future Work) provides directions for future work including a road map for future work with dependence communities.

CHAPTER 2

Background

Graph structure is useful for representing a wide variety of different systems, such as social [59], biological [68], technological and information networks [57]. For example, a social network contains people as nodes and the relationships between people, such as *friend*, or *colleague*, as the edges. A graph can also be called a *network* [144], which is typically used in the fields of social network analysis and community analysis of graphs.

2.1 Complex Networks

Historically the study of networks has been mainly in the domain of the branch of mathematics known as *graph theory* [25]. In addition to the mathematical domain, the study of networks has been important in the area of social sciences; the study of social networks began in the 1930s with work by a psychiatrist named Jacob Moreno with a set of studies which are considered the first true social network studies [144]. The studies included diagrams known as *sociograms* which would now be called *social networks.*

The pattern of connections between components in systems can be represented as a network where nodes represent the components and edges the interactions between them. The nodes in a social network represent people while the edges describe many different interactions between people, for example *friend*, *colleague*, *acquaintance* etc. and the definition of an edge depends on the questions being asked. For example, we may be interested in a network of friends or a network of acquaintances but not both. Edges in the network can be also directed – although Bob may consider Alice his friend, Alice may not consider Bob her friend.

Empirical studies of real-world networks revealed unexpected topological features which existing graph models failed to replicate. Since 1998 the study of networks has had a resurgence after the seminal papers of Watts and Strogatz [184] on small world networks and Barabási and Albert [9] on scale-free behaviour in networks. After the publication of these papers much of the research in complex networks has been conducted within the physics communities, with applications of the results finding uses in many areas from biology [5] to sociology [120] to computing [156].

A complex network is a network with non-trivial topological features such as having the *small world* property [165, 184], a scale-free degree distribution [4, 9] and exhibiting community structure [143]. These features do not appear in random networks, such as in the Gilbert [66] or Rényi-Erdös models [53], and real-world networks are typically complex networks [9, 11, 25, 141].

Figure 2.1, page 23 shows a real-world network detailing the interactions between proteins in yeast [29]; compare this to the random Gilbert [66] network* in fig. 2.2, page 23. A clear difference can be noted between the two graphs – the *yeast* graph has several nodes with a high degree (the large nodes; known as *hubs*) but most have a small degree. In the random graph the nodes are topologically equivalent

*Gilbert's random network model is of the form $G(n,p)$ where n is the number of nodes in the network and each potential edge is chosen to be included or excluded, independently of other edges, based on the probability p.

indicating a homogeneous underlying structure; there is an equal probability of there being a link between two nodes. The degree distribution of the random graph is Poissonian while many real-world networks tend to have a power-law distribution [9].

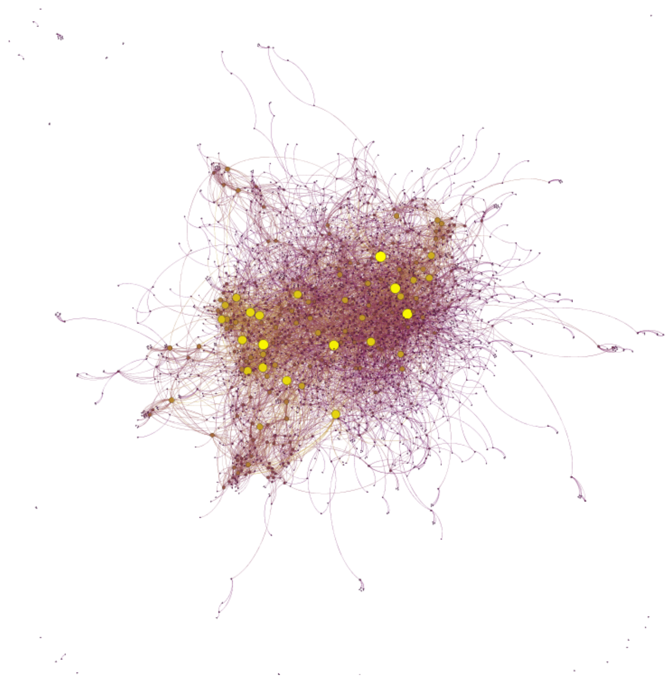


Figure 2.1: The protein-protein interaction network in yeast [29]

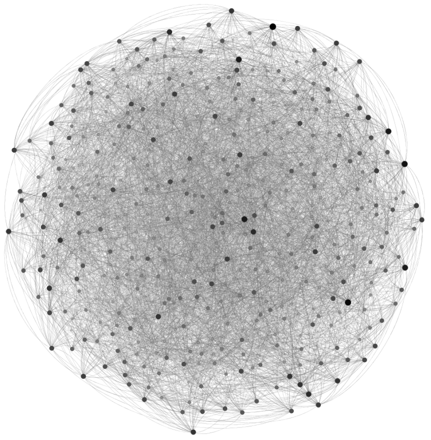


Figure 2.2: Random Gilbert graph $G(n, p)$ with $n = 500$ and $p = 0.05$

2.1.1 Community Structure

The concept of community structure arises from the analysis of social networks in sociology [183]. Community structure can be found in many real world graphs other than social networks [155]. Informally, a graph is said to have community structure if it can be grouped into sets of nodes that are densely connected internally to each other, but sparsely connected to other groups [68]. A community can also be described as a sub-graph which is more tightly connected than average, i.e. cohesive [54].

The strongest definition is that all pairs of nodes in a community must be connected to each other; this is known as a *clique*. The term is derived from social network analysis where it was coined to describe a group of mutual friends [125]. Figure 2.3 shows a graph with 23 1-node cliques, 42 2-node cliques, 19 3-node cliques and 2 4-node cliques.

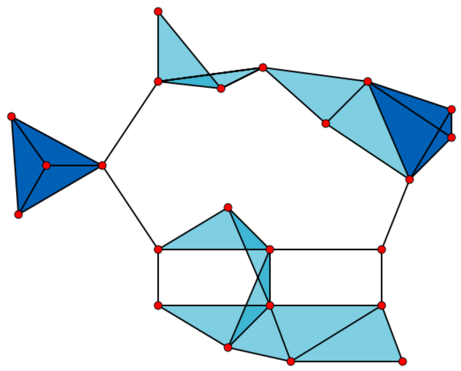


Figure 2.3: A graph with various size cliques.

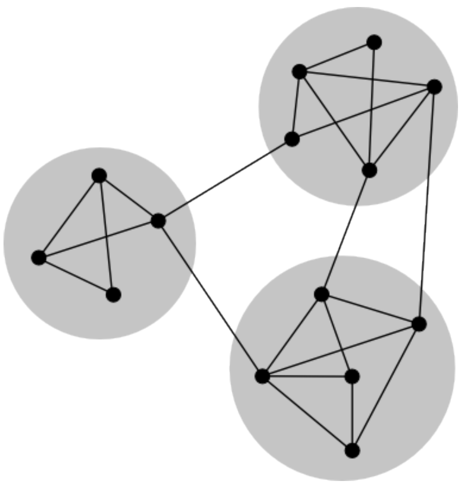


Figure 2.4: Graph with highlighted communities, $Q = 0.489$.

Beyond cliques, there are weak and strong definitions of community [158]. A strong community has more connections within the community than with the rest of the network. In a weak community the sum of the degrees within the community is larger than the sum of the degrees towards the rest of the network. Figure 2.4 shows 3 strong communities – each community has more connections within the community than with the rest of the network.

2.1.1.1 Modularity

Modularity is a measure of the quality of a particular division of a network [142, 143] that can be used to detect community structure, calculated as:

$$Q = \text{(fraction of edges that fall within communities in the given graph)} - \text{(expected number of edges within those communities in the null model)} \tag{2.1}$$

The value of the modularity lies in the range $[-1, 1]$. A modularity greater than 0 means that the graph does exhibit community structure; the higher the modularity the stronger the communities. It is positive if the number of edges within communities exceeds the number expected on the basis of chance. An unpartitioned graph (a graph of 1 community) always has a modularity of 0 and a graph where each node is in it's own community always has a negative modularity.

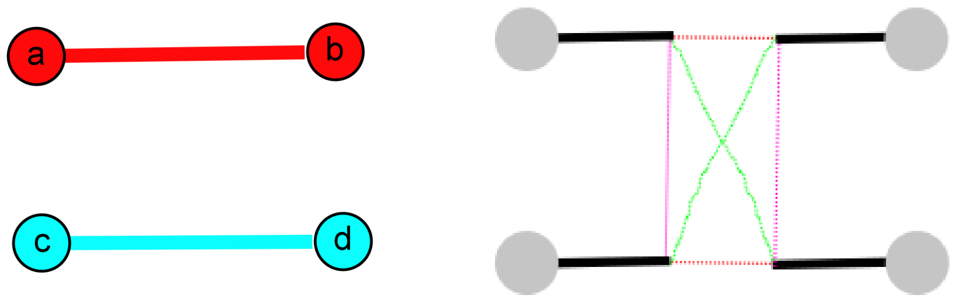
Modularity, of a weighted undirected graph, is defined as [57]

$$Q = \frac{1}{2m}\sum_{i,j}\Big[A_{ij} - E_{ij}\Big]\delta(c_i, c_j) \tag{2.2}$$

where A_{ij} is the weight of the edge incident to i and j, $k_i = \sum_j A_{ij}$ is the sum of the weights of the edges incident to node i, c_i is the community to which node i is assigned, $\delta(u, v)$ is the Kronecker δ-function so the value is 1 if i and j are in the same community and 0 otherwise and $m = \frac{1}{2}\sum_{i,j} A_{ij}$. E_{ij} is the expected number of edges between i and j in a chosen *null model*.

The null model used is based on the configuration model [179] which is a randomised realisation of a particular graph that retains the original degree distribution. Given a graph where each node i has degree k_i, each edge is divided into two halves called *stubs*. Each stub is randomly connected to one of the other l_{n-1} stubs resulting in a random graph with the same degree distribution. The total number of stubs l_n is

$\sum_{k=1}^{n} k_i = 2m$. Figure 2.5b depicts the 3 possible rewirings of the simple graph in figure 2.5a.



(a) $Q = 0.5$ (b) 3 Possible Rewirings

Figure 2.5: A simple graph with 2 communities

In the null model a node can be attached to any other node, by connecting two stubs together. The probability p_i of picking at random a stub attached to node i is $\frac{k_i}{2m}$ since there are k stubs attached to i out of a total of $2m$ stubs. The probability of a connection between i and j is $p_i p_j = \frac{k_i k_j}{4m^2}$ since edges are placed independently of each other. Therefore the expected number of edges E_{ij} between i and j is $2mp_i p_j = \frac{k_i k_j}{2m}$ [57].

Modularity, can therefore be written as [145]

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{2.3}$$

We can calculate the modularity for the graph of adjacency matrix 2.4 and figure 2.6. If there is only one community then $Q = 0$. If each node is placed in it's own community then $Q = -0.130$. The optimal value, $Q = 0.389$, is achieved when there are two communities, $c_1 = \{a, b, c, d\}$ and $c_2 = \{e, f, g, h\}$.

Figure 2.7 depicts variations of the community structure of fig. 2.4, page 24 along with the modularity of each variation.

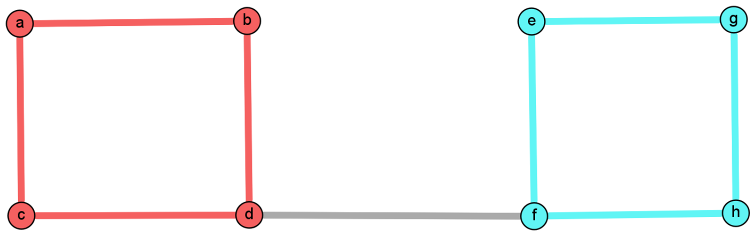
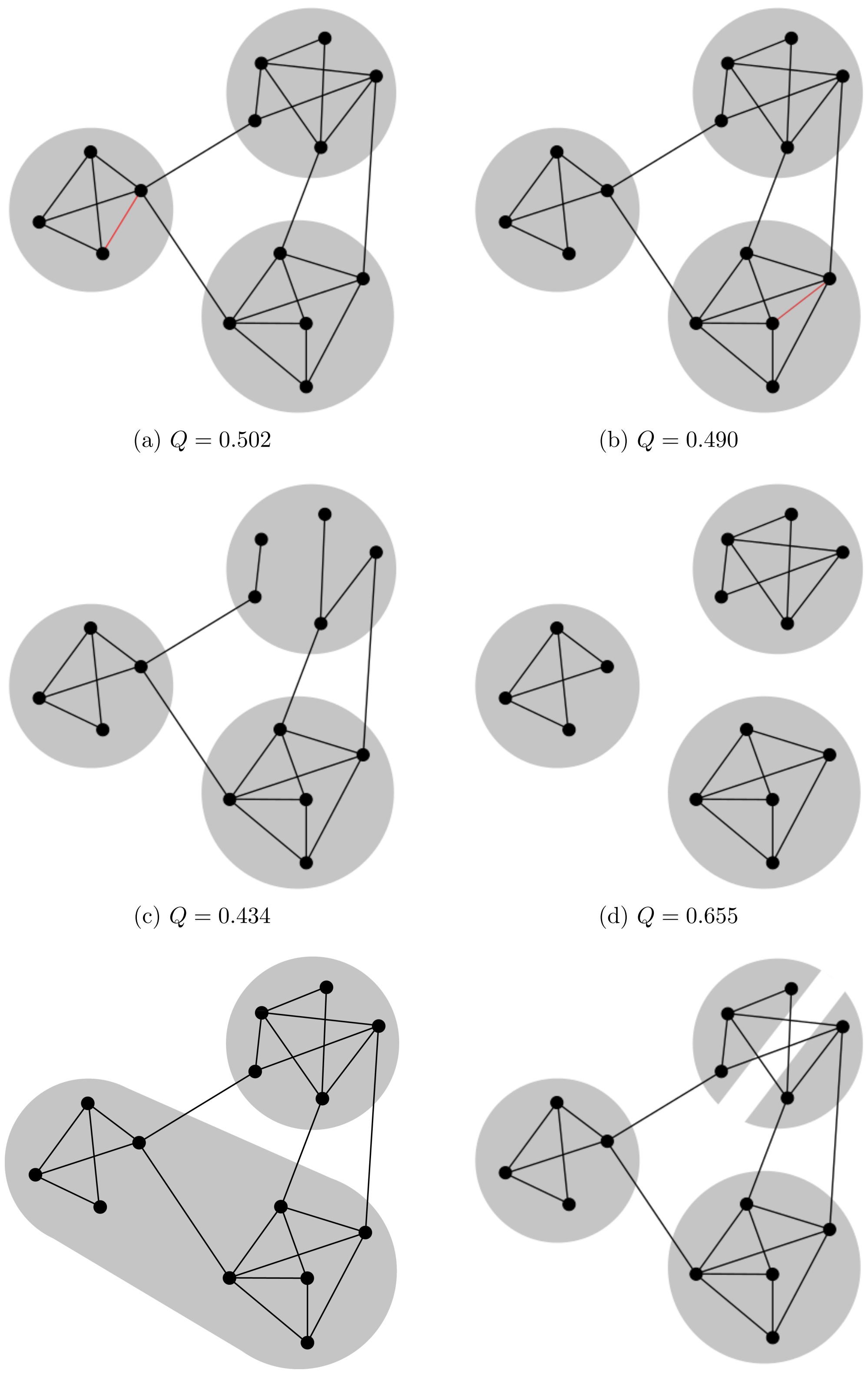


Figure 2.6: Example community structure

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \tag{2.4}$$



(a) $Q = 0.502$

(b) $Q = 0.490$

(c) $Q = 0.434$

(d) $Q = 0.655$

(e) $Q = 0.332$

(f) $Q = 0.385$

Figure 2.7: Variations of the graph/community structure of fig. 2.4

2.1.1.2 Community Detection

The *Louvain method* [24] is a fast algorithm for detecting communities in large networks based upon modularity maximisation. The algorithm combines neighbouring nodes until a local maximum of modularity is reached and then creates a new network of communities; these two steps are repeated until there is no further increase in modularity. This creates a hierarchical decomposition of the network - at the lowest level all nodes are in their own community, and at the highest level nodes are in communities which gives the highest gain in modularity. This technique is simple, fast and has good accuracy. It has been tested on networks with millions of nodes/edges which is particularly important as software systems can be large. The algorithm is extremely fast, and has a linear run-time on large, typical graphs – the majority of the run-time is taken up by the first few iterations, after which the number of communities dramatically decreases.

Blondel et al. [24] compared the algorithm to other modularity maximisation algorithms and found the new algorithm to be faster and able to handle much larger graphs. For example, the modularity of a graph containing 118 million nodes and 1 billion edges was computed by the *Louvain method* in 152 minutes.

Modularity optimisation, in general, has the so-called resolution limit problem where the modularity optimisation fails to identify communities smaller than a certain scale [58]. Using the *Louvain method* the problem is only partially relevant because the algorithm provides a decomposition of the network into communities for different levels of organisation [24]; thus, in the resulting hierarchical community structure levels in between the highest and lowest may contain communities of interest.

The *Louvain method* is divided into two phases that are repeated iteratively. Starting with a network with N nodes we first put each node into it's own community, giving N communities of size 1. Then each neighbour j of node i is considered and the gain in modularity after moving i to the community of j is evaluated. node i is placed into the community that gives the greatest increase in modularity but only if the increase is a positive value. If there is no positive increase obtained by place i into the community of any of it's neighbours then i remains in it's original community. This process is repeated and applied sequentially for all nodes in a randomised order until no further improvement in modularity is possible.

The efficiency of the algorithm is due to the fact that the gain in modularity ΔQ can be easily computed by:

$$\Delta Q = \left[\frac{\Sigma_{in} + k_{i,in}}{2m} - \left(\frac{\Sigma_{tot} + k_i}{2m}\right)^2\right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right]$$

where Σ_{in} is the sum of the weights of the links inside community C, Σ_{tot} is the sum of the weights of the edges incident to nodes in community C, k_i is the sum of the weights of the edges incident to node i (equivalent to the degree in an unweighted network), $k_{i,in}$ is the sum of the weights of edges from i to nodes in community C and $2m$ is the sum of the weights of all the edges in the network (the total number of edges in an unweighted network).

The second phase of the algorithm involves building a new network whose nodes are the communities discovered in the first phase. The weights of the edges between two communities are the sum of the weights of the edges between the nodes in the corresponding communities. After the nodes have been combined into communities the first phase can be re-applied to the new network.

The two phases are iterated until there are no more changes and a maximum modularity is achieved. Blondel et al. [24] notes that the algorithm "is reminiscent of the self-similar nature of complex networks [167] and naturally incorporates a notion of hierarchy".

2.1.1.2.1 Example

Figure 2.8, page 32 depicts the process of detecting the two communities in fig. 2.6, page 27. In the first iteration of the algorithm each node is in it's own community, so there are 8 nodes and 8 communities. The modularity of the graph in this state is $Q = -0.1296$. We can compute the modularity contribution for each community C_i as

$$\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2$$

where Σ_{in} is the number of strictly internal links in community i, Σ_{tot} is the total number of links internally and towards the rest of the network in community i and m is the number of links in the network. In fig. 2.8 the numbers near a node indicate Σ_{in} and Σ_{tot}. For example, the modularity contribution of C_0 is $0-(\frac{2}{2\times 9})^2 \approx -0.0123456$.

The first phase of the algorithm is places the nodes in communities that gives a local maximum for modularity. We iterate through the nodes in the network, removing the node from it's current community and placing it in a neighbouring community that gives the maximum increase in modularity (or keep it's current community if $\Delta Q \leq 0$).

Beginning with moving v_a from C_0 to C_1 results in an increase in modularity from -0.1296 to -0.0432 (alternatively, we could move v_a into C_2 for the same increase in modularity); now v_a and v_b are in the same community C_1. We now have the choice of moving v_c into C_1, C_3 or remaining in C_2 – moving v_c into C_1 gives a modularity of 0.0185 and into C_3 gives a modularity of 0.0309, therefore we move v_c into C_3.

We now have $C_0 = \emptyset$, $C_1 = \{v_a, v_b\}$, $C_2 = \emptyset$, $C_3 = \{v_c, v_d\}$, $C_4 = \{v_e\}$, $C_5 = \{v_f\}$, $C_6 = \{v_g\}$, $C_7 = \{v_h\}$. The node v_d will stay in it's current community and the nodes on the other side of the graph will similarly form 2 communities, resulting in $C_4 = \emptyset$, $C_5 = \{v_e, v_f\}$, $C_6 = \emptyset$ and $C_7 = \{v_g, v_h\}$.

The first iteration is now complete with no more increases in modularity possible; the modularity of this graph is now $Q = 0.19136$.

The next step is to produce a new graph in which the communities are nodes and the weighted edges between the nodes represent the sum of the weight of the edges between the nodes in the original graph. The new graph contains the four communities from the original graph – the weight between C_0 and C_1 is 4 because there were two edges between communities C_1 and C_3 in the original graph; the weight between C_1 and C_3 is 2 and the weight between C_2 and C_3 is also 4. The values of Σ_{in} and Σ_{tot} are also updated to reflect the sum of the original values.

We now iterate through the nodes in the new graph, combining those which increase modularity. This results in two final communities, $C_0 = \{v_a, v_b, v_c, v_d\}$ and $C_1 = \{v_e, v_f, v_g, v_h\}$, with a modularity of $Q = 0.3\overline{888}$ as depicted in c of fig. 2.8. As can be seen from fig. 2.8 there are 3 levels in the hierarchical decomposition of this network.

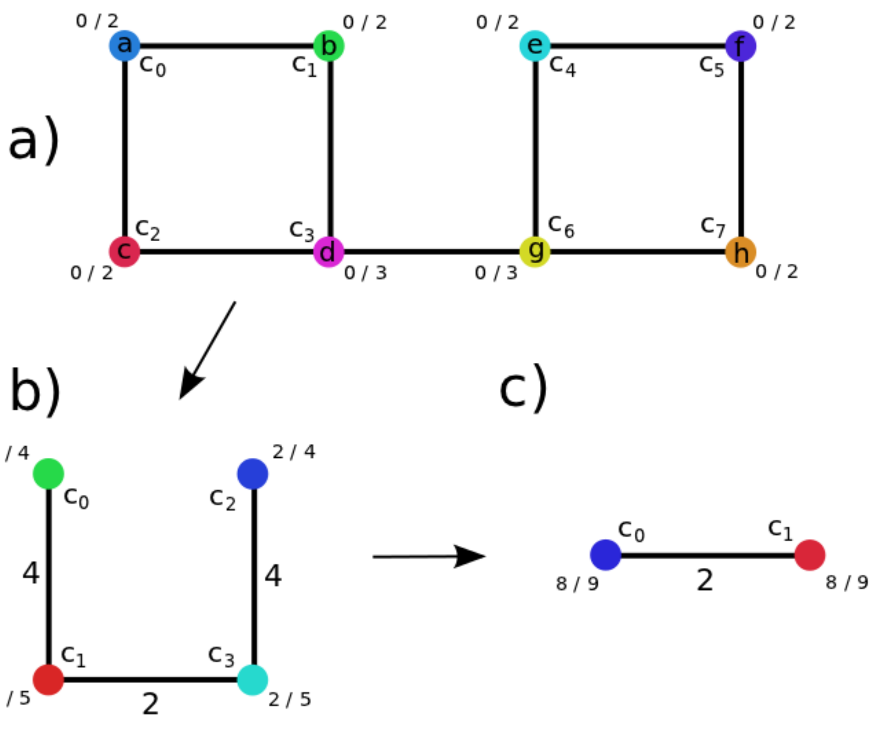


Figure 2.8: Community detection with the *Louvain method*

2.2 Software as Complex Networks

Software systems can be modelled as complex networks where nodes are typically components of a system and edges represent dependencies or interactions between components [187]. There a many different types of relationships in software and previous studies have mainly focused on four types of software network –

Class dependency graphs

In object-oriented programming nodes can represent classes and edges represent the dependencies between classes, such as A *extends* B, or A *uses* B.

Object dependency graphs

In object-oriented programming object graphs contain object instances as nodes that are linked to each other by one object either owning or containing another object or holding a reference to another object.

Call graphs

In a call graph nodes are procedures and edges represent procedure calls, such as f calls g.

Package dependency graphs

At a higher level package dependency graphs represent software packages as nodes and edges representing dependencies between packages.

Software graphs have been shown to have the non-trivial topological features of complex networks [114, 139, 156, 181]. For example, conventionally, object-oriented software is thought of as being built from many small components, like bricks, that when put together form the whole; however, previous studies have found that such software is built from objects that are scale-free and quite unlike bricks [156].

Software graphs that have not previously been studied as complex networks include System Dependence Graphs (SDGs) and Backward Slice Graphs (BSGs) (see section 2.2.2, page 35 for an introduction to SDGs and section 3.2, page 71 for definitions of Backward Slice Graphs (BSGs)).

2.2.1 Communities in Software Networks

The problem of clustering software components has long been studied [82], for example with the use of *hill climbing* [128] and *genetic* algorithms [48] applied to software modules. Previous research in the area has focused on the clustering of high-level components in a software system to recover a modular structure or re-modularise software. The Bunch tool [129, 133], for example, works on a Module Dependency Graph (MDG) which includes high-level system components such as Java classes or C files that are connected due to dependence. The purpose of the tool is to help maintain and understand existing software by clustering related modules together.

After a resurgence in interest in complex networks in recent years, graph theoretic techniques pioneered in areas such as physics and biology have been applied to software; one such area is the development of fast, efficient community detection algorithms and the measure of community structure in complex networks.

Previous studies have shown that software graphs exhibit community structure.

Yakovlev [191] studied Java class dependency graphs where nodes in the graphs were grouped using different clustering methods: grouping by namespace, grouping into clusters using a community detection algorithm and grouping into clusters using rules. Discrepancies between the clusterings revealed candidates for refactoring.

Šubelj and Bajec [180] studied Java class dependency graphs and found that they exhibited significant community structure, as found in other complex networks. The results showed that the communities did not exactly correspond to the software packages. The authors conclude by suggesting applications of the work to software engineering such as revealing the high level modular packaging of software.

Paymal et al. [153] conducted a similar study on the Java software *JHotDraw*; this study, however, considered 6 versions of the software in order to understand how JHotDraw code evolved over time. The results showed that the top two communities contained most of the nodes in each version and there was significant overlap between corresponding communities across consecutive versions. This stability over time suggests that the design was stable, however, as the communities cut across packages a restructuring could be needed.

2.2.2 System Dependence Graph

An SDG [73, 91] is an inter-connected collection of Procedure Dependence Graphs (PDGs) [56]. The nodes of a PDG represent the statements and predicates of a procedure; the edges of a PDG represent the intra-procedural control and data dependencies between statements and predicates. An SDG connects a collection of PDGs by inter-procedural control and data dependence edges.

In addition to the nodes representing statements and predicates each PDG contains explicit entry & exit nodes, variable declarations, nodes representing parameters and return values, and *pseudo*-nodes representing globals modified by a function. A PDG contains `formal-in` nodes representing the parameters to the procedure and `formal-out` nodes representing the variables returned by the procedure.

There is an intra-procedural control-dependence edge between a control node and a second node if the condition controls whether the second point will be executed. If the control node is the condition of a loop the node will have a self-loop as the condition in one iteration influences whether the condition will be executed again. There is an intra-procedural data-dependence edge between two nodes if the first may assign a value to a variable that may be used by the second.

A function call is represented by a `call-site` node and there is an inter-procedural control-dependence edge from each call point to the corresponding entry point of the callee. There is also a control-dependence edge from a procedure's entry node to each of the top-level statements and conditions in that procedure; this is due to the fact that any of the top-level statements are only reachable by the execution of the function.

There is an intra-procedural control-dependence edge between a procedure's entry node and it's formal parameters. There is also an intra-procedural control-dependence edge between each call-site and the actual parameters associated with that call. There is 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 a PDG's `formal-out` nodes and the associated `actual-out` nodes in the calling procedure.

The use of global variables are represented by *pseudo*-nodes in PDGs. For each global variable used or modified by a procedure there is a `global-formal-in` node and for each variable modified by a procedure there is a `global-formal-out` node. 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` nodes and the associated `global-actual-out` nodes in the calling procedure.

2.2.2.1 Summary Edges

There is a lack of transitive dependence information at call sites. This means that a node after calling a procedure may depend on a variable used before calling the procedure but there would be no indication of this in the graph because only direct dependencies are included in a PDG. To solve this issue an SDG contains summary edges [7, 91, 164] which represent the transitive data dependence at call sites.

A summary edge starts at an `actual-in` node and ends at the corresponding `actual-out` node. The summary edge indicates that the `actual-in` node may affect the values of variables at the `actual-out` node. The summary edges allow us to know which variables after a procedure call depend on variables before a procedure call.

2.2.2.2 Example SDG

Figure 2.9 shows the SDG for the simple program in listing 2.1. This program calculates the sum of the numbers 1 to 10 using a procedure, *add*, to do the addition instead of using the primitive + directly; this allows us to discuss an SDG containing two PDGs.

The entry point to the program is the main function's `entry` node; from this node all other nodes are reachable. Control dependence is shown as green edges and data dependence is shown as pink edges. There is a control dependence from the entry node to the top-level nodes: the `global-formal-in` and `global-formal-out` for *sum*, the declaration of the local variables *sum* and *i*, the body of the procedure and the `exit` node.

The main procedure returns an integer which is represented by the `formal-out` node. The use of a global variable for the *sum* variable introduces pseudo-nodes in the form of the `global-formal-in` and `global-formal-out` – these nodes do not actually exist in the real program but they model the behaviour of the global variable. The `global-formal-in` is a copy of the *sum* variable which is copied into the local variable also called *sum*. The local variable is declared in the procedure. The `global-formal-out` node has a data dependence on the local variable and on the `global-formal-in` node.

The body of the procedure consists of a while loop, which contains two function calls, Listing 2.1: Simple C Program

```
int sum = 0;

int add(int a, int b) {
  return a + b;
}

int main(void) {
  int i;
  i = 1;
  while(i<11) {
    sum = add(sum, i);
    i = add(i, 1);
  }
  printf(sum);
  printf(i);
}
```

and print calls to print out the results. The body has a control dependence on the entry node and there is a control dependence from the body to the top-level nodes in the body. The `control-point` node is the predicate controlling the execution of the loop; thus there is a control dependence from it to the nodes contained within the loop.

The `control-point` has a data dependence on the declaration of the variable i and the assignments to i because the value of i determines whether or not the loop executes.

There are two expressions within the loop: exp_1 *sum = add()* and exp_2 *i = add()*; these expressions contain procedure calls which are represented in the SDG as `call-site` nodes. The two calls are to the same procedure, *add*, which takes two parameters. The parameters for exp_1 are *sum* and i and the parameters for exp_2 are i and the constant 1. These parameters are represented by `actual-in` nodes in the calling procedure; the `actual-ins` have a control dependence on the associated `call-site` and a data dependence on the nodes which may affect the variable used at the `actual-in`.

The `actual-in` nodes are connected by a *summary edge* to the corresponding `actual-out` nodes because the value of the variables after the procedure call depend on the values of the variables before the procedure call.

There is a control dependence between each `call-site` and the *add* procedure's `entry` node because calling a procedure is the only way that the procedure will be executed. The body of the *add* procedure simply sums the two parameters and then the result is returned.

The two parameters are represented by `formal-in` nodes – one for each parameter. The values of the actual parameters (represented by the `actual-in` nodes attached to each call site) are copied to the formal parameters of the procedure. There is a data dependence from the actual parameters to the formal parameters.

The `formal-out` of the *add* procedure represents the value of the sum of the two parameters; each function in an SDG has a `formal-out` node and zero or more `global-formal-out` nodes. The `formal-out` node has a data dependence on the `return` node and a control dependence on the `entry` node.

The two `actual-out` nodes in the main procedure have a control dependence on the `formal-out` node of the *add* procedure. Each `actual-out` represents the return value of a single call to a procedure.

At the end of the program the results are printed by calling the *printf* procedure. Each `call-site` is associated with the `actual-ins` containing the values to be output to the screen. The `actual-ins` have a control dependence on the associated `call-sites`.

The main procedure modifies the global variable *sum* therefore there is a `global-formal-out` which represents the global variable modified by the procedure.

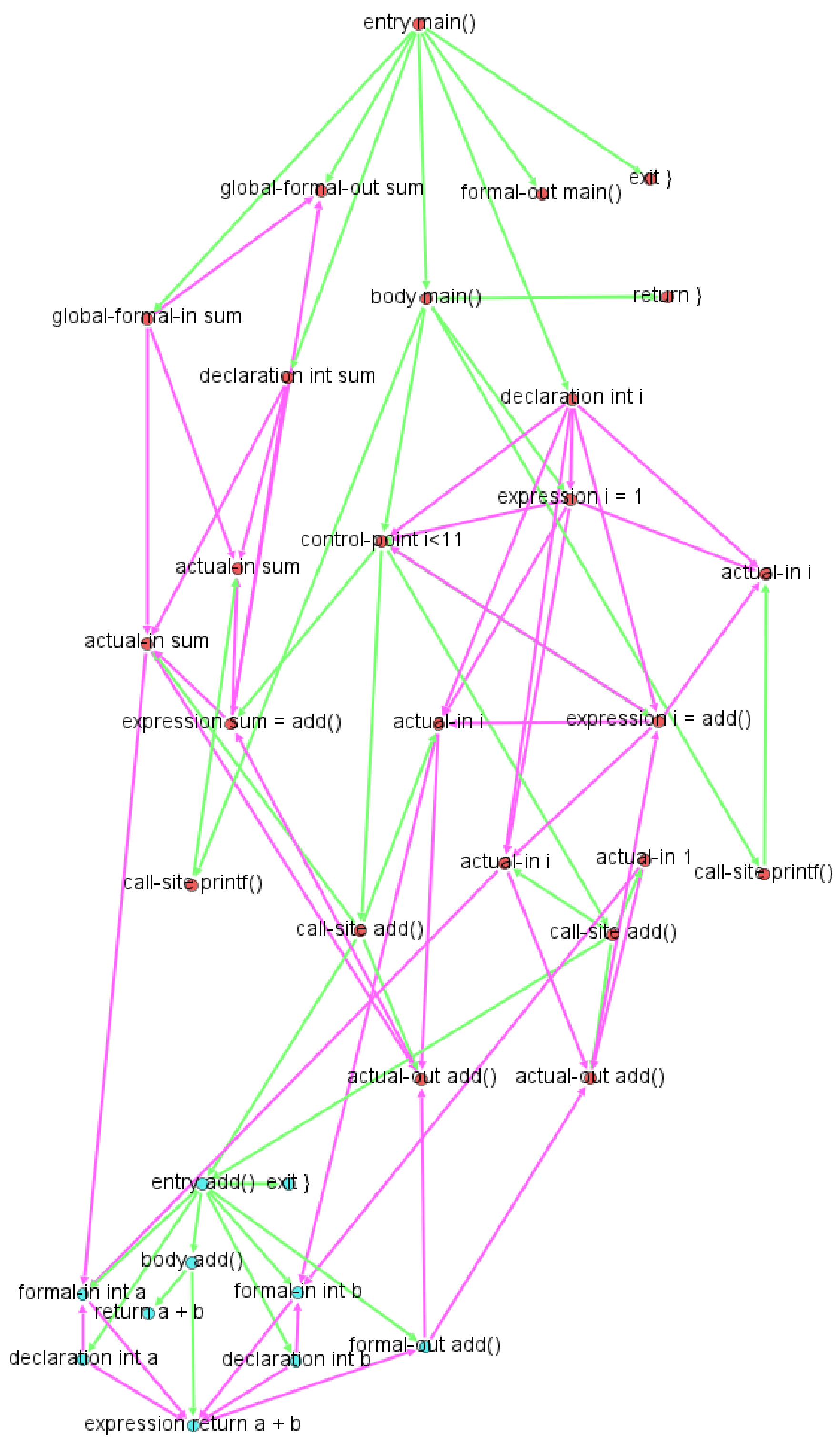


Figure 2.9: System Dependence Graph for the simple program in listing 2.1

2.3 Program Slicing

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.* Program slicing simplifies the analysis of a program by producing a semantically equivalent subset of the program, with respect to some *slicing criterion.*

Program slicing was introduced by Weiser in his PhD Thesis [186]. Informally, a program P is sliced with respect to a *slicing criterion* which is a pair (V, i) where V is a set of program variables and i is a program point. The slice s of P is obtained by removing statements from p such that s is semantically equivalent to P with respect to the slicing criterion (V, i).

Slicing has been applied to structured programs as well as programs with *arbitrary control flow* (i.e. programs that use *goto* statements) [2, 8, 35, 83, 112, 116], concurrent programming languages [32–34, 65, 88, 106–108, 140, 160, 161, 174, 195] and object-oriented languages [6, 12, 31, 51, 110, 118, 134, 135, 147, 147, 159–162, 169, 173, 174, 182, 195, 196].

A slice can be either *executable* or *non-executable.* A *non-executable* slice is simply the set of statements that may affect the value of a variable x at a program point i. For applications such as program understanding a *non-executable* slice is usually sufficient.

Program slices are either *backward* or *forward* slices [84, 189] – a backward slice consists of the program points that may *affect* a criterion whereas a forward slice consists of the program points that may be *affected by* a criterion.

Weiser originally defined *static* slicing where slices are computed to agree with all initial states of a program. Korel and Laski [102] introduced the concept of *dynamic* slicing [3, 12, 26, 98, 101–105, 135–138, 171, 182, 193, 196] where a slice is computed using information from a trace of an execution of the program with a particular initial state.

Program slices were originally computed as a solution to a data-flow problem using a program's Control Flow Graph (CFG) [185]. The nodes of a CFG [89] represent basic blocks in a program[†] and the edges represent jumps between basic blocks.

Later, program slicing algorithms were defined as a solution to a reachability problem on a program's PDG [56, 151].

Ottenstein and Ottenstein [151] only considered slicing on single procedures as a PDG only represents dependencies between statements in a single procedure. Horwitz et al. [91] introduced the SDG which is an extension of the PDG and introduced a traversing algorithm based on the SDG to compute inter-procedural slices.

Horwitz et al.'s algorithm involves two steps: (1) the SDG is firstly augmented with summary edges, which represent transitive dependencies due to procedure calls; then (2) slices can be computed using the augmented SDG with a reachability analysis. The construction of the SDG and the slicing together require time polynomial in the size of the program. The cost of the first step - computing summary edges - dominates the cost of the second step and a more efficient algorithm for computing summary edges was proposed by Reps et al. [164].

[†]a basic block is a consecutive sequence of instructions that begin with a jump target, end with a jump and do not contain jumps or jump targets in between

2.3.1 Slice Example

Consider program P_1 in listing 2.2 and the two slices s_1 and s_2 of P_1 in listings 2.3 and 2.4. The two slices are semantically equivalent to P_1 with respect to different slicing criteria. The slice s_1 is semantically equivalent with respect to the variable *sum* and the program point given by line 11; s_2 is semantically equivalent with respect to the variable *product* and the program point given by line 12.

The slices contain any portion of the code which may affect the slicing criterion – therefore in each slice we see the variables on which there is a data dependence and the loop on which there is a control dependence. These slices are *executable backward* slices of program P_1. Executing s_1 will result in the same value for *sum* as produced by P_1 and executing s_2 will result in the same value for *product* as produced by P_1. This is a simple example but applies to large programs where slices can provide a convenient way to analyse portions of code.

Listing 2.2: Sum and Product Program

```
 1 int main() {
 2   const int N = 10;
 3   int sum = 0;
 4   int product = 1;
 5   int i = 1;
 6   while(i < N) {
 7     sum = sum + i;
 8     product = product * i;
 9     i = i + 1;
10   }
11   printf("%d\n", sum);
12   printf("%d\n", product);
13 }
```

Listing 2.3: Slice of listing 2.2 with respect to the variable sum at line 11

```
int main() {
  const int N = 10;
  int sum = 0;
  int i = 0;
  while(i < N) {
    sum = sum + i;
    i = i + 1;
  }
  printf("%d\n", sum);
}
```

Listing 2.4: Slice of listing 2.2 with respect to the variable product at line 12

```
int main() {
  const int N = 10;
  int product = 1;
  int i = 0;
  while(i < N) {
    product = product * i;
    i = i + 1;
  }
  printf("%d\n", product);
}
```

2.3.2 Program Slicing Tools

2.3.2.1 CodeSurfer

CodeSurfer [72] is "a code browser that understands pointers, indirect function calls, and whole-program effects." It can be used to compute precise inter-procedural slices and has been used in many previous studies [14, 16, 36, 44, 49, 50, 94, 95, 100, 127, 131, 132, 146, 152] to analyse C and C++ code. The algorithm used by CodeSurfer computes precise slices by "tracking dependencies transmitted through the program only along paths that reflect the fact that when a procedure call finishes, control returns to the site of the most recently invoked call" [7, 73].

2.3.2.2 Indus

Indus [163] is a program slicing tool for the Java object-oriented programming language. Indus computes slices using the Jimple [175] intermediate language provided by the Soot Toolkit [52, 176, 177].

2.4 Dependence Clusters

A dependence cluster is a maximal set of program statements all of which are mutually dependent [16]. Large dependence clusters may hinder software maintenance as a change in any of the members in a dependence cluster could affect any other member. Although some dependence clusters may naturally occur in a program, unwanted dependence clusters may be the result of bad programming practice which could be refactored away. Binkley and Harman [16] use the term 'dependence pollution' for such unwanted and avoidable dependence clusters, and large dependence clusters are an example of a 'dependence anti-pattern' [21]. Global variables have been shown to be a cause of 'dependence pollution' [22]. Binkley and Harman define dependence clusters in terms of CFG nodes and SDG nodes [87].

Listing 2.5 is a (statement-level) dependence cluster because any statement may affect any other statement.

Listing 2.5: Example Dependence Cluster

```
while(i < 10)
     if(A[i] > 0)
          i = i + 1;
```

Definition 2.1 (Dependence Cluster [16]) *A dependence cluster is a set of nodes,* $\{N_1, \ldots, N_m\}$ $(m > 1)$*, of the Control Flow Graph (CFG) such that for all* i*,* $1 \leq i \leq m$ *and for all* $j, 1 \leq j \leq m$ N_i *depends on* N_j*.*

Harman et al. [87] later re-define dependence clusters in terms of Mutually Dependent Sets (MDSs) and a dependence cluster as a maximal MDS[‡].

[‡]the original definition [16] was not maximal

Definition 2.2 (Mutually Dependent Set [87]) *A Mutually Dependent Set is a set of statements,* $\{s_1, \ldots, s_m\}$ $(m > 1)$*, such that for all* i, j*,* $1 \leq i, j \leq m$ s_i *depends on* s_j*.*

Definition 2.3 (Dependence Cluster [87]) *A dependence cluster is an MDS not properly contained within any other MDS.*

The focus of work into dependence clusters has been rooted in the control and data dependence that exists between nodes in an SDG. Harman et al. [87] define dependence clusters in terms of a program's slices computed on the SDG. Definition 2.4 satisfies definition 2.2 as all nodes in a slice-based MDS depend upon all others in the MDS (including themselves).

Definition 2.4 (Slice-Based Mutually Dependent Set [87]) *A slice-based MDS is a set of SDG nodes* $\{n_1, \ldots, n_m\}$ $(m > 1)$*, such that for all* i, j*,* $1 \leq i, j \leq m$*,* $n_i \in Slice(n_j)$*.*

Definition 2.5 (Slice-Based Dependence Cluster [87]) *A slice-based dependence cluster is a slice-based MDS not properly contained within any other slice-based MDS.*

Binkley and Harman [16] calculate (slice-based) dependence clusters in empirical studies, using CodeSurfer, by saying that nodes are in a dependence cluster if and only if they have the same slice; Binkley and Harman define dependence clusters in this way because two nodes that depend on each other must have the same slice - since a slice contains its own slicing criterion then where two criteria have the same slice each criterion must be in the slice of the other and therefore must depend on each other[§]. For clarity, in this thesis, we refer to these as Type 2 dependence clusters.

Definition 2.6 (Type 2 Slice-Based Dependence Cluster)
A Type 2 slice-based dependence cluster is a set of SDG nodes $\{n_1, \ldots, n_m\}$ $(m > 1)$, *such that for all* i, j, $1 \leq i, j \leq m$, $Slice(n_i) = Slice(n_j)$.

In order to reduce the computational effort required to obtain such dependence clusters an approximation to Type 2 dependence clusters was introduced by Binkley and Harman, where nodes are in a dependence cluster if and only if their slices are the same size. Finding Type 2 dependence clusters would require $\mathcal{O}(n^3)$ time whereas computing Type 1 dependence clusters would require $\mathcal{O}(n^2)$ time (excluding the slicing phase) where n is the number of SDG nodes.

Their conjecture was based upon the assumption that sufficiently large slices that are the same size are likely to be the same slice. This is a conservative approximation to Type 2 dependence clusters as any cluster identified may contain real clusters and no real clusters will fail to be identified. For clarity, in this thesis, we refer to this approximation to Type 2 dependence clusters as Type 1 dependence clusters.

Definition 2.7 (Type 1 Slice-Based Dependence Cluster)
A Type 1 slice-based dependence cluster is a set of SDG nodes $\{n_1, \ldots, n_m\}$ $(m > 1)$, *such that for all* i, j, $1 \leq i, j \leq m$, $|Slice(n_i)| = |Slice(n_j)|$.

A verification study was conducted to provide evidence to support the Type 1 approximation. The study analysed 36 out of the 45 programs used in the full dependence cluster empirical study. It was found that 99.55% of slices of the same size, from the 36 programs, differed by only up to 1%. The difference was measured in

[§]this definition satisfies definition 2.4 but not definition 2.5, and is therefore an approximation to 'true' dependence clusters; more on this in chapter 5 (Dependence Clusters)

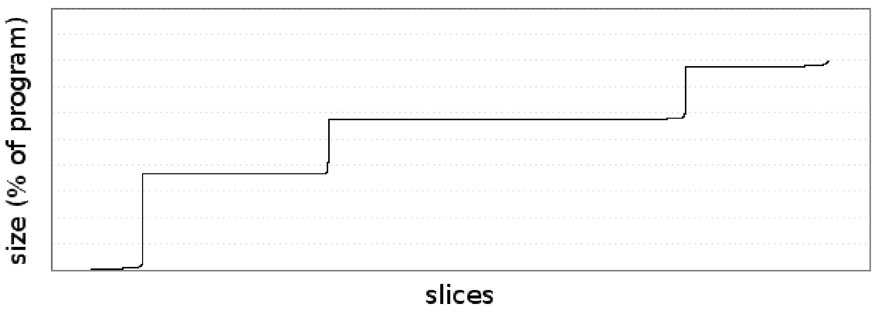


Figure 2.10: Example Monotonic Slice-size Graph

terms of the Jaccard similarity between slices therefore most slices that have the same size required very few nodes to be removed before they could achieve 100% similarity. This evidence suggested to Harman et al. that 'same size' is a good proxy for 'same slice'.

The approximation led to the introduction of the Monotonic Slice-size Graph (MSG), which is a graph of the function of slice size, plotted for monotonically increasing size. This is a useful visualisation for locating dependence clusters in a program – a plateau (created by a set of slices that are the same size) indicates the presence of a dependence cluster. The size of the plateau indicates the size of the dependence cluster and the height shows how much of the program is covered by the dependence cluster.

Definition 2.8 (Monotonic Slice-size Graph [16]) *An MSG is a graph of the function of slice size, plotted for monotonically increasing size.*

The lack of 'sharpness', for example slices that differ by 1 node will appear on the same plateau, in the visualisation is suggested to be an advantage as the dominant features of the dependence cluster landscape are more readily identified [17, 87]. Figure 2.10 shows an example MSG that contains several possible dependence clusters.

Islam et al. [95] introduced the concept of *coherent dependence clusters* that allow a fine-grained analysis of the relationships between clusters of dependence in a program. A coherent dependence cluster "is a set of statements all of which are mutually dependent on each other, the same set of statements depend on them, and they all depend on the same set of statements" [95].

Definition 2.9 (Coherent MDS/Dependence Cluster [95]) *A Coherent MDS is a set of statements S, such that $\forall x, y \in S$: 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.*

Coherent dependence clusters are formulated in terms of backward and forward slices:

Definition 2.10 (Coherent-Slice MDS/Dependence Cluster) *A Coherent-Slice MDS is a set of statements, S, such that*

$$\forall x, y \in S : BSlice(x) = BSlice(y) \wedge FSlice(x) = FSlice(y)$$

A Coherent-Slice Cluster is a Coherent-Slice MDS contained within no other Coherent-Slice MDS.

Islam et al. [95] introduced two new visualisations: Monotone Cluster-Size Graph (MCG) and Slice/Cluster-Size Graph (SCG), where cluster sizes are plotted in monotonically increasing order, to provide a greater precision than MSGs.

An empirical study was conducted using a set of 8 open-source programs to discover if coherent dependence clusters occur in real-world programs. The results showed that 6 out of the 8 programs contain coherent dependence clusters that encompass more than 25% of the program and 3 that encompass more than 50%. The clusters in the program *bc* were manually inspected and it was discovered that the clusters mapped to the functionalities of the program; for example, one cluster encompassed the 'calculator' code, another the parser code.

Harman et al. [16, 87] conducted the first large scale study of dependence clusters in programs, with a study of 45 programs totalling just over 1.2 million lines of code. The study considered Type 1 dependence clusters computed using backward and forward slicing. The study computed MSGs for the 45 programs to provide visual evidence of dependence clusters in programs.

The study defined a large dependence cluster as one which consumes more than 10% of the program. Large dependence clusters were found to be surprisingly common with 40 out of 45 programs containing clusters that consume 10% or more of the program. Several programs contained dependence clusters consuming up to 80% of the program.

The causes of large dependence clusters were briefly considered with a large scale investigation left for future work. For each node in a program's SDG, its effect on large dependence clusters was assessed by recomputing the slices without traversing the node. This provided a way of finding nodes which caused large dependence clusters; for most nodes this caused no effect but several nodes had large effects. In one example, a single node caused a large dependence cluster; this node turned out to be a switch statement that created a mutual recursion which involved most of the functionality of the program.

Binkley and Harman [18] conducted a study into the wider causes of dependence clusters; they identified the 'linchpin' nodes in the SDG that cause large dependence clusters. The search for dependence clusters was conducted by computing the MSG while ignoring the specific nodes and edges. The area under the MSG was computed with the components removed; the nodes and edges that produced the biggest drop were noted – many of the nodes that caused at least a 20% drop in area were control points.

The effect of global variables on dependence clusters was further considered by Islam [93] in his MSc thesis and later extended in a larger study by Binkley, Harman, Hassoun, Islam, and Li [22]. Binkley et al. conducted an empirical study of 21 programs totalling just over 50,000 lines of code and considered 849 global variables to determine the effect that they have on dependence clusters. The results of the study showed that over half of the programs include global variables that cause large dependence clusters and contain a single global variable that is solely responsible for a dependence cluster.

2.5 Slice-Based Cohesion and Coupling Metrics

Software metrics have long been used in an attempt to quantify code quality, for example for quantifying software development resources [55], refactoring [170], improving software [117] and finding bugs [152]. A software metric is a quantitative measure of some aspect of code, ranging from crude measures such as Lines of Code to measures of relationships between software components such as cohesion and coupling [192].

Cohesion metrics attempt to quantify the inter-relatedness of code in a program module. A highly cohesive module suggests that its statements are highly interrelated and perform one job; whereas a poorly cohesive module may be performing multiple jobs.

Coupling metrics attempt to quantify the inter-relatedness of modules in a program. High coupling between two modules suggests that there is a large dependence between the two modules; removing or changing one module would greatly affect the other; whereas low coupling between two modules suggests that they are performing separate tasks without a large dependence on each other.

Ideally, program modules should have a high cohesion and a low coupling corresponding to a modular design - where each task is separated into individual modules - which reduces complexity and allows programmers to work on modules in isolation [168].

2.5.1 Cohesion

Sliced-based cohesion metrics attempt to determine the cohesiveness of a program module by calculating the intersection of slices. Using slices to calculate cohesion metrics works well due to the relatedness definition of cohesion. If statements in a module are related it is likely that their slices will share many statements. On the other hand, if there are multiple tasks taking place in a module it is likely that the intersection will be small or empty.

Weiser [185] defined several metrics, which were later found to be related to cohesion [122], that have since been further studied, refined and added to by others [80, 149, 150]:

tightness

the ratio of the size of the slice intersection to the total module length.

minimum coverage

the ratio of the smallest slice to the total module length.

coverage

the ratio of the average slice size to the total module length.

maximum coverage

the ratio of the largest slice size to the total module length.

overlap

the average ratio of the intersection of all slices to slice size.

parallelism

the number of slices which have few statements in common.

clustering

the degree to which slices are reflected in the original code layout.

Tightness, coverage, overlap, parallelism and clustering were originally defined by Weiser while minimum coverage and maximum coverage were later added by Ott and Thuss [150]. Longworth dismissed clustering as a measure of cohesion as it is “an aspect of arrangement only” [122] and Meyers and Binkley suggest that parallelism has “not proven useful in software reconstruction” [132]; for these reasons previous research has focused on Tightness, MinCoverage, Coverage, MaxCoverage and Overlap.

Early work on slice-based metrics focused on understanding the relationship between slices and the metrics suggested by Weiser. Longworth [122] investigated several of the slice based metrics defined by Weiser and found that they appeared to be related to cohesion. Longworth studied a number of small programs comparing the values of metrics and the expected cohesion and concluded that coverage is a good indicator of low or high cohesion levels and overlap and tightness can be used to confirm the level suggested by coverage.

In Weiser’s thesis slices where computed for each variable at all output statements resulting in a large number of slices per program; these slices were clustered together based on their size to reduce this number. Longworth also took slices for each variable in a program but the slice was taken from the end of a program instead of at each variable location (later named *end-slices* [115]); the number of slices was also reduced by removing those with a size less than 10% of a module and combining those which are at least 90% identical.

Ott and Thuss [150] took a different approach in their study of slice based cohesion metrics: rather than removing slices or using some combination of slices, output variables¶ were used as slicing criteria. These variables included “parameters and globals modified, variables that are written by the module, and the return value of a function” [150]. Other studies have used different definitions of output variable [74], which are enumerated in table 2.1, page 55 and slice-based metrics are undefined for modules with no output variables [20, 115]. These problems have been highlighted

¶called ‘principle variables’ in some literature.

by studies attempting to replicate published studies [28] and a study of 'Key Statements' which was based on Ott and Thuss's definition of output variables [20].

Paper(s)	**Definition of output variable**
The Relationship Between Slices And Module Cohesion, Ott and Thuss, 1989	"v is a reference parameter (or any variable which retains its assigned value after the module has completed execution). If M is the main program module, v is any data object passed to the module by the operating system, e.g., the standard output file. or more outputs." [149]
Rule-based approach to computing module cohesion, Lakhotia, 1993	"A variable is termed output variable for a given procedure if it is modified within it and is either a reference parameter or declared outside the scope of that procedure." [115]
Measuring functional cohesion, Bieman and Ott, 1994 Program slices as an abstraction for cohesion measurement, Ott and Bieman, 1998	"An 'output' is any single value explicitly output to a file (or user output), an output parameter, or an assignment to a global variable." [13, 148]
An examination of the behavior of slice based cohesion measures, Karstu, 1994	"variable parameters, global variables. Also included in outputs would be any direct output from the module to files or devices. . . the definition of output includes only variables and constants" [99]
Cohesion Metrics, Harman et al., 1995	"we consider only those global variables and reference parameters whose values are *modified* by the program component" [80]
Slice-based measurement of function coupling, Harman et al., 1997	"the only variables of interest are those output at the end of a function"
Code Extraction Algorithms which Unify Slicing and Concept Assignment, Harman et al., 2002	"A variable v in the set of statements S is a principle variable iff it is either: global and assigned in S, call-by-reference and assigned in S, the parameter to an output statement in S." [86]
A longitudinal and comparative study of slice-based metrics, Meyers and Binkley, 2004 Slice-based cohesion metrics and software intervention, Meyers and Binkley, 2004 An empirical study of slice-based cohesion and coupling metrics, Meyers and Binkley, 2007	"a function's return value or those globals modified by the function" [130–132]
Bug Classification Using Program Slicing Metrics, Pan et al., 2006	"An output variable of a function can be the function's return value or a non-local variable modified in the function" [152]
Statement-Level Cohesion Metrics and their Visualization, Krinke, 2007	"variables that are used after the module M is left. For example, these include the return value or global variables." [109]
Evaluating Key Statements Analysis, Binkley et al., 2008	"v is a global variable assigned in f or v is used in an output statement (e.g., print or write)in f" . . . "The set of global variables potentially modified by a call to a function f, is easily extracted because each such global variable has a special *global-formal-out* node in the sub-graph representing f. The set $\mathcal{PV}_O$ is also easily determined from the SDG as each output variable appears in an *actual-in* node (i.e.,is used as an actual parameter). The set $\mathcal{PV}_O$ includes those variables found in the *actual-in* vertices of calls to output functions such as **printf** and **write**." [20]
Program Slice Metrics and Their Potential Role in DSL Design, Counsell et al., 2009	"We analyzed multiple versions of the system and sliced its functions in three separate ways (i.e., input, output and global variables). . . Formal-ins (Input parameters for the function), Formal outs (The set of return variables) Global variables (The set of variables which are used by the module)" [46]
An Analysis of the "Inconclusive' Change Report Category in OSS Assisted by a Program Slicing Metric, Counsell et al., 2010	"We denote a set of variables used by a function K as V_K and V_z as the subset of V_K representing output (return), input, global and printf variables (i.e. variables used in printf statements)" [45]
Program Slicing-Based Cohesion Measurement: The Challenges of Replicating Studies Using Metrics, Bowes et al., 2011	"These variables include: 1. Formal Ins: Input parameters for the function specified in the module declaration. 2. Formal Outs: Return variables. 3. Globals: Variables used by or affected by the module. 4. Printf: Variables which appear as Formal Outs in the list of parameters in an output statement. . . We combined these variable settings to collect data in 30 different ways for each of the five cohesion metrics." [28]

Table 2.1: Varying definitions of output variable

Green et al. [74] compared 11 studies which used output variables and came to the conclusion that there are four main types of output variables:

1. the return value of a function.
2. global variables modified by a function.
3. parameters passed by reference to, and modified by, a function.
4. any variables printed, written to a file, or otherwise output to the external environment.

Out of the 11 studies that were investigated 7 different combinations of output variable definitions found. Most of the previous studies in the area of slice-based metrics have used CodeSurfer; the last category is the most difficult to capture with CodeSurfer (as it is the most difficult to define). In the most recent large-scale study by Meyers and Binkley [131] the set of output variables contained a function's return value and those globals modified by the function.

Ott and Thuss [150] formalised cohesion metrics in terms of the slices obtained from the set of output variables and the size of a module. The set of variables used by the module M is V_M and the subset of V_M that are output variables is V_O. The slice SL_i is the slice given by the slicing criteria v_i and SL_{int} is the intersection of all slices in a module:

$$\mathrm{SL}_{int} = \bigcap_{i=1}^{|V_O|} \mathrm{SL}_i$$

The length of a module M is given by $\mathrm{length}(M)$ which is calculated as the number of Variable-Referent Executable Statements (VRESs) – these are statements which reference the variables in V_M.

The 6 cohesion metrics were defined as:

$$\mathit{Tightness}(M) = \frac{|\mathrm{SL}_{int}|}{\mathrm{length}(M)}$$

$$\mathit{MinCoverage}(M) = \frac{1}{\mathrm{length}(M)} \min_i |\mathrm{SL}_i|$$

$$\mathit{Coverage}(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|\mathrm{SL}_i|}{\mathrm{length}(M)}$$

$$\mathit{MaxCoverage}(M) = \frac{1}{\mathrm{length}(M)} \max_i |\mathrm{SL}_i|$$

$$\mathit{Overlap}(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|\mathrm{SL}_{int}|}{\mathrm{SL}_i}$$

$$\mathit{Parallelism}(M) = |\{\mathrm{SL}_i \text{ such that } |\mathrm{SL}_i \bigcap \mathrm{SL}_j| \leq \tau \text{ for all } j \neq i\}|$$

The values of *Tightness*, *MinCoverage*, *Coverage*, *MaxCoverage* are ratios of a particular slice size (or set of slices) to the length of the module and are thus related such that $\mathit{Tightness}(M) \leq \mathit{MinCoverage}(M) \leq \mathit{Coverage}(M) \leq \mathit{MaxCoverage}(M)$; the value of *Overlap* is not mathematically bound to these. The values of *Tightness*, *MinCoverage*, *Coverage*, *MaxCoverage* and *Overlap* all lie within the range 0 to 1 but *Parallelism* will range from 0 upwards, depending on τ, with higher values indicating that the module contains multiple unrelated or only slightly related tasks.

Ott and Thuss [150] compute the values of metrics using *metric-slices* which are the union of the forward and backward slices of a module. The motivation for this is that when computing metrics the interest is in the *uses* and *used by* relationships in a module. In order to validate the metrics, the values of the metrics where examined using small example programs to see if they produced the desired results.

Karstu [99] studied the relationship between cohesion and revisions made to software over time with findings that suggested highly cohesive modules are less likely to need changes.

Bieman and Ott [13, 148] introduced *data-slices* which modify the concept of *metric-slices* to use data tokens (variable and constant definitions and references) rather than statements. The motivation for using data tokens as the basis of slices was to ensure that all changes of interest in a module (i.e. any change that affects the cohesiveness of the module) will cause a change in at least one slice. Bieman and Ott define two types of data token – *glue* tokens are those that are common to more than one slice in a procedure and *super-glue* tokens are those that are common to all slices in a procedure. The distribution of *glue* and *super-glue* tokens indicates how tightly bound the slices are and the *stickiness* (or *adhesiveness*) between tokens is then used as a basis for defining cohesion metrics. Two metrics were defined: *strong functional cohesion* is the ratio of *super-glue* tokens to the total number of tokens in a module; and *weak functional cohesion* is the ratio of *glue* tokens to the total number of tokens in a module.

Krinke [109] introduced a statement-level cohesion metric which is able to identify areas of low cohesion within procedures, compared to other cohesion metrics which identify module-level cohesion. The statement-level cohesion is the sum of the slice sizes in which a statement appears as a proportion of the sum of all slice sizes in a module.

Table 2.2, page 60 shows an example calculation of cohesion metrics by slicing with respect to the output variables. The 3 variables *sum*, *product* and *count* are all printed and are therefore output variables in this function. The slices are depicted in the *slice profile* next to the program code – a bar in the column indicates that the line is included in the slice; the columns show, in order from left-to-right, slices on *sum*, *product* and *count*. The intersection of all 3 slices is just 2 which indicates that the program may be performing more than one task; this is reflected in the low Tightness of 0.154. The mid-value Coverage indicates that the slices in the module cover about half of the module, on average. The low value for Overlap indicates that there is a low percentage of the module that is common to all slices.

```
void main() {

    const int N = 10;             |   |   |
    int sum = 0;                  |
    int product = 1;                  |
    int count = 0;                        |
    int i = 1;                    |   |

    while(i < N) {                |   |   |
        sum = sum + i;            |
        product = product * i;        |
        count = count + 1                 |
        i = i + 1;                |   |
    }

    printf("%d\n", sum);          |
    printf("%d\n", product);          |
    printf("%d\n", count);                |
}
```

$$
\begin{aligned}
Tightness(main) &= \frac{2}{13} = 0.154 \\
MinCoverage(main) &= \frac{5}{13} = 0.385 \\
Coverage(main) &= \frac{1}{3}\left(\frac{7}{13} + \frac{7}{13} + \frac{5}{13}\right) = 0.487 \\
MaxCoverage(main) &= \frac{7}{13} = 0.538 \\
Overlap &= \frac{1}{3}\left(\frac{2}{7} + \frac{2}{7} + \frac{2}{5}\right) = 0.324
\end{aligned}
$$

Table 2.2: Example calculation of line-based cohesion metrics

2.5.2 Coupling

Slice-based coupling was first introduced by Harman et al. [85] who provided a slice-based version of Henry and Kafura's information flow based coupling [90]. The motivation for using slices to calculate coupling was based on the idea that slicing provides a better measure of dependence between procedures than information flow. Slice-based coupling gives rise to the notion of 'bandwidth' of dependence between procedures. The number of statements in a slice given by a criteria in a procedure f that appear in procedure g reflects how much f depends on g. The number of statements from the slice in f that appear in g indicates the 'bandwidth' of the information flow. The coupling between two functions is given as the normalised values of the flow in each direction.

The flow between two modules f and g is given by

$$FF^{P}_{(f,g)} = \frac{\#\Big((\bigcup_{v \in p(g)} SS(p,[v],E(g))) | N(f) \Big)}{N(f)}$$

where $SS(p,[v],E(g))$ is the slice on program p with respect to the set of output variables $[v]$ at the end of module g, $p(g)$ is the set of output variables of g and $N(f)$ denotes number of CFG nodes in module f.

Green et al. [74] simplify the notation by introducing a new symbol $N(f \rightarrow g)$ to mean the number of CFG nodes in a module f which are in the union of slices on the output variables of the module g, thus flow can be written as

$$FF^{P}_{(f,g)} = \frac{N(f \rightarrow g)}{N(f)}$$

Coupling between two modules f and g is then defined as

$$Coupling(f,g) = \frac{FF^{P}_{(f,g)} \times N(f) + FF^{P}_{(g,f)} \times N(g)}{N(f) + N(g)} = \frac{N(f \rightarrow g) + N(g \rightarrow f)}{N(f) + N(g)}$$

Meyers and Binkley [131] define coupling for a single module as a weighted average of its coupling with each of the other modules in a program:

$$Coupling(f) = \frac{\sum_i \text{Coupling}(f, g_i) \times N(g_i)}{\sum_i |g_i|}$$

Table 2.3, page 63 shows an example of the calculation of coupling by slicing interprocedurally with respect to the output variables in each procedure. The output variables in *add* and *addOne* are the function's return value while in *main* they are the two printed variables *sum* and *i*. The slice profile for each slice is shown, from left-to-right, as the union of the slices w.r.t to *sum* and *i* in *main*, the slice w.r.t to the return value in *add* and finally the slice w.r.t to the return value in *addOne*. We are interested in the size of the slice that appears not in the originating procedure but the other procedures in the program. A bar or underscore in a column indicates that the line is included in the slice but an underscore indicates membership of a slice in the originating procedure. The coupling between *add* and *addOne* is 1 because slicing in either procedure includes the whole of the other.

```
int add(int a, int b) {
     return a + b;                |    _    |
}

int addOne(int a) {
     return add(a, 1);            |    |    _
}

void main() {
     int sum = 0;                 _    |
     int i = 1;                   _    |    |

     while(i<11) {                _    |    |
        sum = add(sum, i);        _    |
        i = addOne(i);            _    |    |
     }

     printf("%d\n", sum);         _
     printf("%d\n", i);           _
}
```

$$
\begin{aligned}
N(main \to add) &= 1 & N(main \to addOne) &= 1\\
N(add \to main) &= 5 & N(addOne \to main) &= 3\\
N(add \to addOne) &= 1 & N(addOne \to add) &= 1\\
N(main) &= 7 & N(add) &= 1\\
N(addOne) &= 1 & &
\end{aligned}
$$

$$
\begin{aligned}
Coupling(main, add) &= \frac{1+5}{7+1} = \frac{3}{4} = 0.75\\
Coupling(main, addOne) &= \frac{1+3}{7+1} = \frac{1}{2} = 0.5\\
Coupling(add, addOne) &= \frac{1+1}{1+1} = 1\\
Coupling(main) &= \frac{\frac{3}{4} \times 1 + \frac{1}{2} \times 1}{1+1} = \frac{5}{8} = 0.625\\
Coupling(add) &= \frac{1+\frac{3}{4}}{1+7} = 0.21875\\
Coupling(addOne) &= \frac{1+\frac{1}{2}}{1+7} = 0.1875
\end{aligned}
$$

Table 2.3: Example calculation of line-based coupling

2.5.3 Empirical Studies

Meyers and Binkley [130, 131, 132] conducted the most extensive study of slice-based cohesion and coupling metrics to-date. The studies used the CodeSurfer software to calculate slices from which the values of slice-based metrics could be calculated. Meyers and Binkley calculate Tightness, MinCoverage, Coverage, MaxCoverage, Overlap and Parallelism metrics defined by Ott and Thuss [150] but do not use *metric-slices* instead using traditional slices generated by the Horwitz et al. [91] slicing algorithm. Meyers and Binkley suggest "a metric slice is likely to be larger than a traditional slice with a corresponding drop in usefulness" [131].

Meyers and Binkley analysed a range of 63 programs containing a total of 22,651 procedures including open-source and commercial software. The studies provide base-line values for cohesion and coupling metrics, providing a useful measure against which others can compare metric values.

Meyers and Binkley performed a head-to-head comparison of slice-based metrics, finding a very strong correlation between tightness and mincoverage, and strong correlations between mincoverage and overlap; tightness and overlap; and maxcoverage and coverage. The study also considered the relationship between slice-based metrics and Lines of Code finding no relationship between the two.

Two longitudinal studies were performed which showed that slice-based metrics can be used to quantify the deterioration of software during development. The study found that the value of cohesion tends to fall between minor releases of software but recovers with a major release. This provided empirical evidence from real-world software of the link suggested by Karstu [99] between on-going software maintenance and cohesion.

Bowes et al. [27] studied the effect of the choice of output variable definitions on the values of slice-based cohesion metrics. The study found that the values of slice-based metrics are sensitive to multiple factors including the level of abstraction at which slices are taken (system or file levels) and the inclusion or exclusion of 'printf' statements, formal-ins, formal-outs, and global variables. The study investigated the GNU Barcode program used in Meyers and Binkley's study to compare the values of Tightness, Overlap, MinCoverage, Coverage and MaxCoverage using various combinations output variable definitions. The results showed that the values drop significantly when slicing files individually rather than as a whole project, especially the Tightness value. MaxCoverage was found to be the least sensitive metric. The exclusion of global variables as output variables increased the values of metrics significantly but the inclusion or exclusion of 'printfs', formal-ins or formal-outs made little difference.

Black et al. [23] investigated the 19 revisions of the GNU Barcode program using slice-based Tightness and Overlap along with fault data extracted from online report logs. The values of the metrics in procedures with at least one reported fault were compared against fault-free procedures to determine whether any differences were observed and to determine whether the metrics might allow prediction of faulty modules. The results suggested that low values of Tightness seem to indicate fault-prone procedures whereas the values of Overlap were not significantly different. Counsell et al. [45] extended the study to consider a category excluded from the analysis which comprised of 221 'inconclusive' log reports where it was not clear whether or not a fault occurred. The extended study presented a methodology to allow conclusions to be drawn from the 'inconclusive' category. The results suggested that the 'inconclusive' category is more strongly related to the fault-prone category than the non-fault-prone category from the earlier study.

Bowes et al. [28] repeated the longitudinal study of GNU Barcode by Meyers and Binkley in order to further validate the metrics and provide a replication of the study which adds to the corpus of metrics data available to researchers and software maintainers. The study considered 49 versions of the program which contained 65 procedures. A total of 30 distinct combinations of output variable definition were used in order to attempt to produce the values reported in the original study. The closest match to the original study was obtained when slicing files individually with only the inputs to 'printf' as output variables. However, these values were still significantly different from the original study and the new study failed to reach the same conclusions as the original. Bowes et al. were unable to conclude why their values were so different from Meyers and Binkley and suggest that researchers must present their work with sufficient detail for replication to be possible in future studies.

CHAPTER 3

Dependence Communities

The concept of community structure arises from the analysis of social networks in sociology. Community structure can be found in many real world graphs other than social networks. Do Backward Slice Graphs (BSGs) have community structure?

We provide empirical evidence that dependence between statements in software gives rise to community structure; this leads to the introduction of the concept of dependence communities in software. We give examples where the dependence communities approximate to the semantic concerns of a program.

The results of our empirical study provide compelling evidence that there is great potential for using dependence communities as an alternative software clustering methodology, applicable to a number of areas of software engineering including re-engineering and re-factoring, maintenance, comprehension and metrics.

3.1 Introduction

Community structure has been shown to exist in software networks, for example class dependency networks [153, 180, 191]. In this chapter we apply the *Louvain method* [24], used in fields such as social network analysis, to Backward Slice Graphs (BSGs) to answer the question: do BSGs have community structure?

We believe that for a clustering technique to be useful in software engineering it must provide what we call *Semantic Separation*, i.e. the clusters must in, some sense, partition the system into its different *functionalities*. We manually inspect several programs to answer the question: what do dependence communities in software mean? Our hypothesis is that statements that are related due to control and data dependence will form 'communities' and that such areas will approximate to the semantic concerns of the program.

The *Louvain method* is a fast algorithm for detecting communities in large networks based upon modularity maximisation (see section 2.1.1.2, page 29). The algorithm combines neighbouring nodes until a local maximum of modularity is reached and then creates a new network of communities; these two steps are repeated until there is no further increase in modularity. This creates a hierarchical decomposition of the network - at the lowest level all nodes are in their own community, and at the highest level nodes are in communities which gives the highest gain in modularity. This technique is simple, fast and has good accuracy.

Dependence communities are closely related to the previously studied *dependence clusters*. A dependence cluster is a stricter form of community where every node within a community is connected to every other node in that community (see chapter 5, page 114).

3.1.1 The Program Sample

The studies in this thesis are based on the analysis of 44 open-source C programs; details of the programs are listed in figure table 3.1. The programs studied are a collection of open-source software that cover a range of application domains such as games, small and large utilities and operating system components. The smallest program has 71 Analysed Lines of Code (ALoC)[*], while the largest has 76,369. The total ALoC for the set of 44 programs is 464,621 and the average ALoC per program is 10,559.57.

The 44 SDGs were computed using CodeSurfer. The smallest SDG, in terms of nodes, has 499 nodes and the largest has 2,954,718 nodes. In terms of edges, the smallest has 1,173 edges and the largest has 12,800,065 edges. The total number of nodes for the set of 44 program SDGs is 13,949,332 and the average number of nodes per SDGs is 39,222.73. The total number of edges for the set of 44 program SDGs is 61,878,708 and the average number of edges per SDGs is 1,406,334.25.

The program with the smallest number of slices had 172 slices and the program with the largest number of slices had 305,037 slices. The total number of slices computed for the set of 44 programs was 1,725,800 and the average number of slices per program is 317,030.28.

There are a total of 20,588 non-empty procedures[†] in the 44 programs and an average of 469.73 procedures per program.

[*]ALoC are the number of lines of code obtained by counting the unique number of lines of code attached to System Dependence Graph (SDG) nodes generated by CodeSurfer.

[†]some procedures are empty, for example *silent_error_logger* in *gettext-0.18*

Name	**ALoC**	**Procedures**	**SDG nodes**	**SDG Edges**	**Slices**	**Description**
a2ps-4.14	17,518	1,034	271,685	1,566,984	57,844	a2ps is an Any to PostScript filter. It is able to process and pretty print many types of file.
acct-6.5.5	2,691	130	15,596	47,834	6,324	The GNU Accounting Utilities provide login and process accounting utilities for GNU/Linux and other systems.
acm-5.1	922	48	3,462	11,155	2,029	A distributed aerial combat simulator that runs on the X Windows System
adns-1.3	5,865	394	102,855	347,055	20,895	Advanced, easy to use, asynchronous-capable DNS client library and utilities.
aeneas-1.2	803	45	17,016	61,044	8,315	Package for the design and simulation of submicron semiconductor devices.
anubis-4.1	6,618	418	68,510	247,986	19,510	An SMTP message submission daemon.
archimedes-1.2.0	766	44	17,665	68,781	7,957	Package for semiconductor device simulations.
barcode-0.98	2,111	73	14,457	48,006	5,452	A tool to convert text strings to printed barcodes.
bc-1.06	5,249	216	41,025	310,319	11,037	An arbitrary precision numeric processing language
cflow-1.3	6,226	325	57,209	213,966	17,135	A program that analyzes a collection of C source files and prints a graph, charting control flow within the program.
combine-0.3.4	4,527	101	37,185	127,748	15,585	The program matches files on arbitrary keys using a hash table, and reports on the matches (or lack thereof).
cppi-1.15	1,949	122	12,904	46,162	5,471	Indents C preprocessor directives to reflect their nesting, among other regularizations.
diction-1.11	865	46	7,791	23,723	3,283	Diction identifies wordy and commonly misused phrases.
diffutils-3.2	9,042	413	48,210	163,360	21,621	A package of several programs related to finding differences between files.
ed-1.5	1,883	133	30,779	110,126	7,274	A line-oriented text editor.
empire-4.3.28	40,704	1,546	2,277,653	9,242,821	156,776	A real time, multiplayer, Internet-based game, featuring military, diplomatic, and economic goals.
enscript-1.6.5	9,896	249	107,684	539,478	33,681	Converts ASCII files to PostScript, HTML, or RTF and stores generated output to a file or sends it directly to the printer
findutils-4.4.2	17,206	887	157,518	534,727	52,843	The basic directory searching utilities of the GNU operating system.
garpd-0.2.0	308	22	2,239	6,113	1,034	Broadcasts Gratuitous ARPs for a list of MAC address ¡-¿ IP address mappings on specified interfaces at regular intervals.
gettext-0.18	76,369	2,998	2,954,718	12,800,065	305,037	A set of tools that provides a framework to help other GNU packages produce multi-lingual messages
gforth-0.7.0	7,457	116	26,851	4,456,673	13,907	A fast and portable implementation of the ANS Forth language.
global-6.0	22,964	955	774,361	3,916,873	86,136	A source code tagging system that works the same way across diverse environments.
gnats-4.1.0	13,425	583	130,337	551,887	35,295	A set of tools for tracking bugs reported by users to a central site.
gnubik-2.4	2,936	211	13,542	31,977	9,417	An interactive, graphical, single player implementation of the Rubik's cube.
gnuchess-6.0.1	12,064	653	103,525	336,793	28,072	A free chess-playing program.
gnuedma-0.18.1	12,485	746	1,334,533	3,921,091	63,560	An Object Oriented Component-Based development environment to build modular and evolving applications as well as highly reusable components.
gnuit-4.9.5	11,522	520	181,333	785,979	29,792	A set of interactive text-mode tools, closely integrated with the shell.
gnujump-1.0.5	4,979	243	42,620	136,880	15,345	A clone of the simple yet addictive game Xjump, adding new features like multiplaying, unlimited FPS, smooth floor falling, themable graphics, sounds, replays.
gnurobots-1.2.0	1,230	85	5,551	13,306	3,773	A game/diversion where you construct a program for a little robot, then watch him explore a world.
gnushogi-1.3	4,417	177	24,079	79,091	9,597	An implementation of version of chess played in Japan.
gperf-3.0.4	3,625	199	15,712	45,420	8,976	A perfect hash function generator.
inetutils-1.8	31,941	1,451	835,082	2,928,870	142,631	A collection of common network programs.
lame-3.99.1	16,337	762	107,876	410,316	39,736	An educational tool to be used for learning about MP3 encoding.
ntp-4.2.6p5-RC1	41,232	1,631	1,177,368	5,501,985	167,394	A reference implementation of the Network Time Protocol.
pure-ftpd-1.0.32	7,142	275	65,674	240,992	18,591	A free (BSD), secure, production-quality and standard-conformant FTP server.
rsync-3.0.9	21,099	775	2,145,312	9,327,181	142,962	A program to synchronize file trees across local disks, directories or across a network.
sed-4.2	6,527	340	75,658	276,319	24,377	A stream editor, used to perform basic text transformations on an input stream.
tar-1.23	22,002	1,185	535,824	1,974,361	97,435	Provides the ability to create tar archives, as well as various other kinds of manipulation
time-1.7	382	14	1,822	5,290	826	The 'time' command runs another program, then displays information about the resources used by that program, collected by the system while the program was running.
userv-1.0.3	3,629	264	69,978	231,217	14,184	An enhanced version of the emacsclient program for GNU Emacs.
wc	71	8	499	1,173	172	Counts lines, words, and characters in one or more input files.
wdiff-0.5	658	32	3,242	9,326	1,301	A front end to diff for comparing files on a word per word basis.
which-2.20	774	41	5,138	15,787	2,189	A utility that is used to find which executable (or alias or shell function) is executed when entered on the shell prompt.
zlib-1.2.5	4,205	158	27,254	162,468	11,029	A free, general-purpose, legally unencumbered, lossless data-compression library.
Sum	464,621	20,668	13,949,332	61,878,708	1,725,800	
Average	10,559.57	469.73	317,030.28	1,406,334.25	39,222.73	

Table 3.1: The 44 subject programs used in the studies in this thesis

3.2 Backward Slice Graphs (BSGs)

Before we can look for communities of statements in software, we need a suitable graph in which to look for such communities. We want edges to represent dependencies between statements.

Program slicing [186] is a technique which computes a set of program statements, known as a *backward slice*, that may affect a point of interest known as the *slicing criterion*. The backward slice of a given statement s contains all the other statements upon which s depends. A natural choice was, therefore, to connect node v_1 to v_2 if v_1 is in the backward slice of v_2.

We call such a graph a Backward Slice Graph (BSG). Communities in the BSG will, thus, be sets of statements with *strong* inter-dependencies. We call such sets *dependence communities.*

We also considered looking for communities in the SDG [91]. Here nodes are only connected if there is a *direct* data or control dependence. Transitive de-

Program	**BSG**		
	nodes	**Edges**	**Density**
a2ps-4.14	57,844	1,258,072,909	0.376
acct-6.5.5	6,324	5,499,790	0.138
acm-5.1	2,029	295,025	0.0717
adns-1.3	20,895	177,781,311	0.407
aeneas-1.2	8,315	20,658,335	0.299
anubis-4.1	19,510	149,296,932	0.392
archimedes-1.2.0	7,957	16,881,616	0.267
barcode-0.98	5,452	11,732,564	0.395
bc-1.06	11,037	62,116,117	0.510
cflow-1.3	17,135	101,597,540	0.346
combine-0.3.4	15,585	47,417,876	0.195
cppi-1.15	5,471	6,661,891	0.223
diction-1.11	3,283	3,334,634	0.309
diffutils-3.2	21,621	76,902,352	0.165
ed-1.5	7,274	37,419,735	0.707
empire-4.3.28	156,776	16,112,669,388	0.656
enscript-1.6.5	33,681	287,940,746	0.254
findutils-4.4.2	52,843	1,191,798,635	0.427
garpd-0.2.0	1,034	269,612	0.252
gettext-0.18	305,037	9,961,065,414	0.107
gforth-0.7.0	13,907	52,707,360	0.273
global-6.0	86,136	2,500,096,080	0.337
gnats-4.1.0	35,295	666,919,670	0.535
gnubik-2.4	9,417	805,652	0.00908
gnuchess-6.0.1	28,072	85,105,593	0.108
gnuedma-0.18.1	63,560	1,141,664,053	0.283
gnuit-4.9.5	29,792	356,982,854	0.402
gnujump-1.0.5	15,345	77,794,411	0.330
gnurobots-1.2.0	3,773	169,759	0.0119
gnushogi-1.3	9,597	34,527,979	0.375
gperf-3.0.4	8,976	13,394,816	0.166
inetutils-1.8	142,631	4,265,435,922	0.210
lame-3.99.1	39,736	580,705,172	0.368
ntp-4.2.6p5-RC1	167,394	8,053,207,729	0.287
pure-ftpd-1.0.32	18,591	130,914,642	0.379
rsync-3.0.9	142,962	16,449,155,234	0.805
sed-4.2	24,377	374,571,561	0.630
tar-1.23	97,435	4,946,266,842	0.521
time-1.7	826	127,281	0.187
userv-1.0.3	14,184	97,915,761	0.487
wc	172	9,296	0.314
wdiff-0.5	1,301	385,953	0.228
which-2.20	2,189	2,513,834	0.525
zlib-1.2.5	11,029	65,570,162	0.539

Table 3.2: Slice graph statistics for the 44 programs.

pendencies are not connected. Intuitively, transitive dependencies are as important as direct ones (their effect is just as important) and so we rejected this approach. To back up our intuition, we applied the *Louvain method* directly to the SDG in a number of cases and found there was a strong correlation between the communities and the separate procedures in the programs. The reason for this is high ratio of intra-procedural edges to inter-procedural edges in the SDG. Figure 8.1 shows the SDG and the BSG of the *wc* program.

We exclude *pseudo*-nodes generated by CodeSurfer and retain the nodes that represent source-code statements (with the exception of `global-formal-out`[‡]). This allows us to tie the results back to the source-code of a program as well as greatly reducing the number slices required to be computed.

Definition 3.1 (Backward Slice Graph (BSG))

A Backward Slice Graph (BSG) G consists of a set of SDG nodes V and a set of edges E of the form $e_{i,j}$ *where* $\forall e_{i,j} \in E, v_j \in BackwardSlice(v_i)$

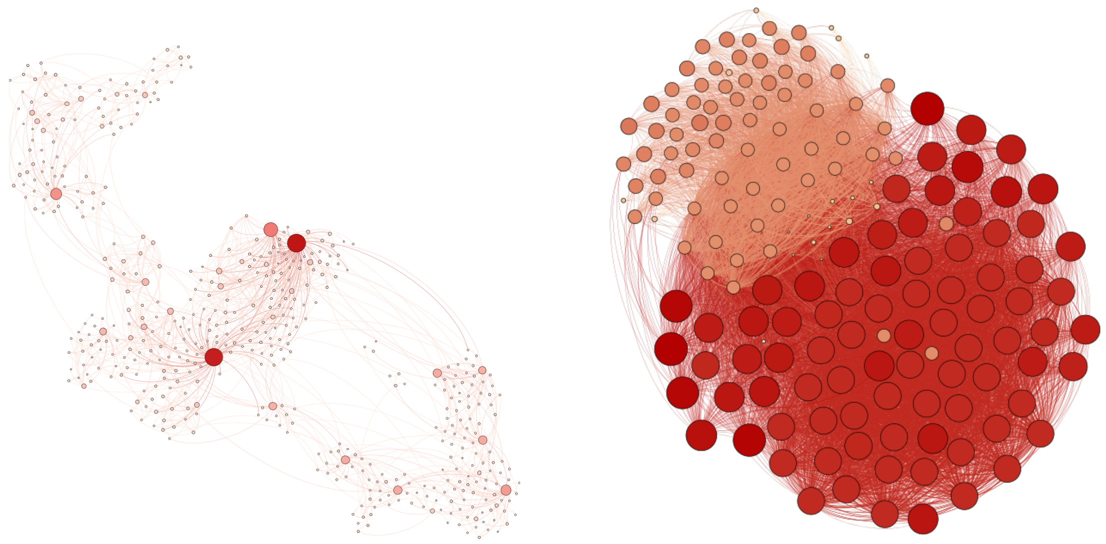

(a) System Dependence Graph (SDG) (b) Backward Slice Graph (BSG)

Figure 3.1: The *wc* program as two complex networks

[‡]these nodes are used for calculating metrics in chapter 4

3.3 Dependence Communities in Backward Slice Graphs

In this section, we show the result of applying the *Louvain method* to the BSG of a simple program. In this small example, we found the results far from arbitrary: the algorithm appeared to partition the graph into communities each of which approximated to different 'semantic' concerns of the program. This initial promising result was the evidence which led us to investigate these communities more extensively.

The *sumproduct* program is shown in listing 3.1. The communities found in the program are depicted in fig. 3.2. There are 3 communities detected in this program: the 'sum' community, the 'product' community, and the 'support' community.

The 'support' community consists of the parts of the program which are not directly involved in calculating the sum or product, such as the procedure body, exit, return, and the calls to *printf*. This community is separate because there are fewer dependencies between itself and the sum/product code.

The 'product' community consists of the parts of the program which compute the product, including the initialisation of the product variable, the updating of the product variable and the `actual-in` to the *printf* call.

The 'sum' community consists of the parts of the program which compute the sum but also the loop code and the counter N. Communities cannot overlap because the algorithm partitions the graph therefore the loop code, which should intuitively be in both communities, must be placed in one or the other; the choice of the sum community is arbitrary.

The modularity of this partition of the graph is 0.227 - a positive value indicates that the graph has community structure.

Listing 3.1: Sum Product Program with 3 communities highlighted in different colours

```
int main() {

  const int N = 10;
  int sum = 0;
  int product = 1;
  int i = 1;

  while(i < N) {
    sum = sum + i;
    product = product * i;
    i = i + 1;
  }

  printf("%d\n" , product );
  printf("%d\n" , sum );
}
```

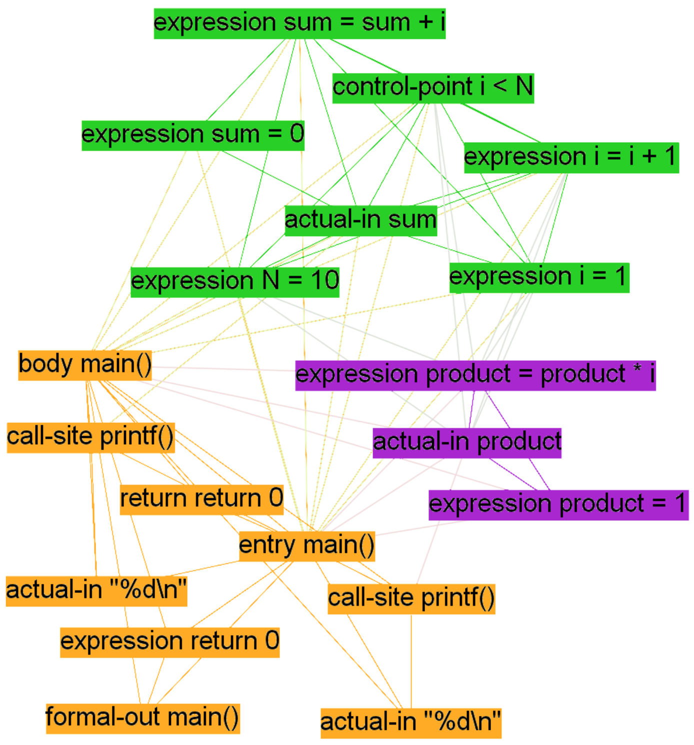


Figure 3.2: Communities detected in the sumproduct program (listing 3.1), $Q = 0.227$

3.4 Empirical Study

The study in this section is based on the analysis of 44 open-source programs. The programs studied are a collection of open-source software that cover a range of application domains including games, small and large utilities and operating system components.

3.4.1 Research Questions

Do Backward Slice Graphs (BSGs) have community structure?

Community structure has been shown to exist in module-level software networks, for example class dependency networks [153, 180, 191]. Additionally, a strict form of clustering at the statement-level has previously been studied in the form of *dependence clusters.* Our hypothesis is that dependence at the statement-level may give rise to community structure, revealing less strict groupings of statements in software compared to dependence clusters. A positive result would have applications in all areas of software engineering where system decomposition is important, including software comprehension & maintenance, reverse engineering & restructuring, software evolution and information recovery.

We show that the 44 BSGs generated from the set of programs in the empirical study have community structure, as evidenced by the positive modularity obtained by partitioning them using the *Louvain method.*

3.4.2 Method

In our empirical study, we used CodeSurfer to compute backward slices for SDG nodes in a program, to generate BSGs for the set of 44 programs.

We then applied the *Louvain* algorithm to the BSG of each of the 44 programs. As well as measuring the modularity we calculated the number of dependence communities, the size of the smallest, the size of the largest and the average size of communities measured as a percentage of program size.

3.4.3 Results

An important result of the study was that all 44 programs have a positive modularity which suggests that dependence in software does, indeed, exhibit community structure at the statement level. Details of the communities found in the 44 programs are listed in table 3.3, page 78 and fig. 3.3 plots the modularity for each of the programs.

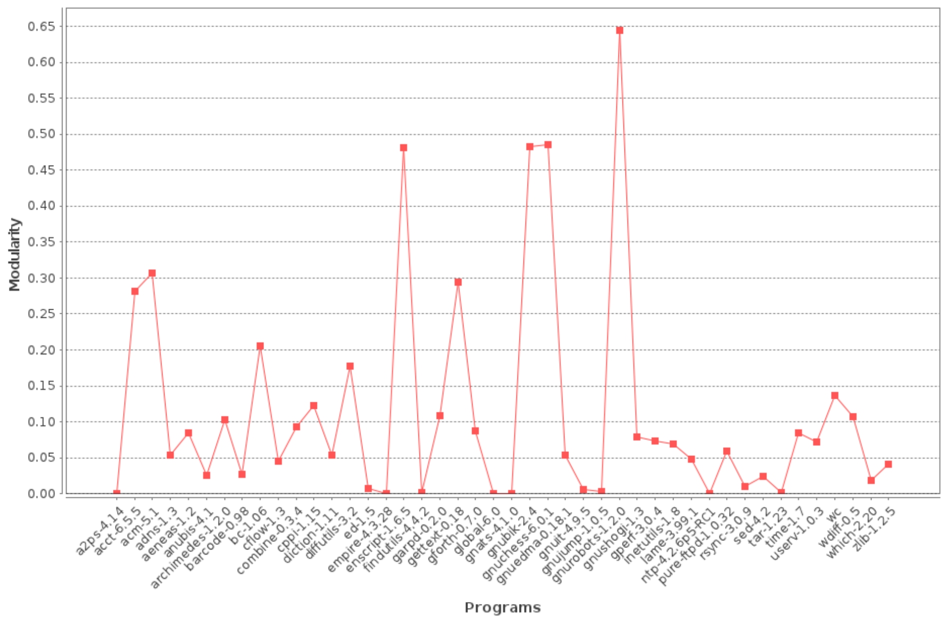


Figure 3.3: Modularity of the 44 Backward Slice Graphs.

The highest modularity is 0.644 for *gnurobots-1.2.0*, the lowest is 7.47×10^{-5} for *global-6.0* and the average modularity for the 44 programs is 0.12.

The program *ed-1.5*, although not the smallest, has only 2 communities – one consuming about 10% and the other 90% of the program. This suggests that there is a large section of code with a high dependence between statements. This result is similar to the discovery of a large dependence cluster in the same program by Binkley and Harman [16].

The largest community, on average, consumes 53.41% of a program; this echoes the results of Binkley and Harman who found that programs contain large dependence clusters [18]. Two programs (*global-6.0* and *gnujump-1.05*) have a dependence community that is $> 90\%$ of the program code. The larger program of the two has a community that is 97.8% of the program and 57 other communities; and the smaller has a community that is 92% of the program and only 11 other communities.

As program size increases (in terms of ALoC) so does the number of communities detected; the Pearson correlation coefficient is $R = 0.82$ with p-value < 0.000001. This could be due to larger programs performing more sub-tasks as part of the overall function of the program. The largest program, *gettext-0.18*, contains 264 dependence communities with a modularity of 0.294. The average number of dependence communities in this set of programs is 32.91.

Program	**Dependence Communities**				
	Number	**Smallest**	**Largest**	**Average**	**Modularity**
a2ps-4.14	102	0.00346%	93.2%	0.980%	0.00101
acct-6.5.5	11	0.221%	28.2%	9.09%	0.282
acm-5.1	25	0.197%	32.4%	4.00%	0.306
adns-1.3	10	0.0239%	42.2%	10.0%	0.0535
aeneas-1.2	3	22.2%	43.8%	33.3%	0.0844
anubis-4.1	25	0.0256%	48.9%	4.00%	0.0257
archimedes-1.2.0	11	0.126%	43.2%	9.09%	0.103
barcode-0.98	4	0.514%	73.8%	25.0%	0.0277
bc-1.06	11	0.0634%	50.8%	9.09%	0.206
cflow-1.3	23	0.0117%	83.9%	4.35%	0.0454
combine-0.3.4	16	0.0321%	58.6%	6.25%	0.0935
cppi-1.15	24	0.0914%	46.6%	4.17%	0.123
diction-1.11	7	0.0609%	51.3%	14.3%	0.0539
diffutils-3.2	51	0.0278%	54.1%	1.96%	0.178
ed-1.5	2	10.3%	89.7%	50.0%	0.00692
empire-4.3.28	13	0.00383%	98.8%	7.69%	0.000247
enscript-1.6.5	27	0.00891%	54.6%	3.70%	0.481
findutils-4.4.2	40	0.00568%	91.5%	2.50%	0.00207
garpd-0.2.0	5	1.93%	43.2%	20.0%	0.108
gettext-0.18	264	0.00131%	53.1%	0.379%	0.294
gforth-0.7.0	10	0.0503%	61.3%	10.0%	0.0872
global-6.0	56	0.00348%	97.5%	1.79%	7.47e-05
gnats-4.1.0	25	0.0113%	97.2%	4.00%	0.000332
gnubik-2.4	39	0.0531%	16.2%	2.56%	0.483
gnuchess-6.0.1	78	0.0142%	35.3%	1.28%	0.486
gnuedma-0.18.1	60	0.00315%	64.7%	1.67%	0.0543
gnuit-4.9.5	32	0.0101%	85.6%	3.12%	0.00661
gnujump-1.0.5	12	0.0391%	92.0%	8.33%	0.00277
gnurobots-1.2.0	25	0.106%	18.9%	4.00%	0.644
gnushogi-1.3	13	0.0208%	66.4%	7.69%	0.0796
gperf-3.0.4	21	0.0334%	48.8%	4.76%	0.0730
inetutils-1.8	71	0.00210%	81.3%	1.41%	0.0694
lame-3.99.1	34	0.0126%	56.7%	2.94%	0.0482
ntp-4.2.6p5-RC1	70	0.00179%	97.4%	1.43%	8.38e-05
pure-ftpd-1.0.32	12	0.0377%	72.5%	8.33%	0.0595
rsync-3.0.9	33	0.00210%	55.1%	3.03%	0.00965
sed-4.2	11	0.0164%	53.5%	9.09%	0.0248
tar-1.23	58	0.00308%	95.8%	1.72%	0.00109
time-1.7	5	9.56%	37.7%	20.0%	0.0853
userv-1.0.3	6	0.0423%	79.4%	16.7%	0.0715
wc	2	48.8%	51.2%	50.0%	0.136
wdiff-0.5	4	15.1%	32.1%	25.0%	0.107
which-2.20	5	0.411%	67.6%	20.0%	0.0181
zlib-1.2.5	26	0.0272%	44.3%	3.85%	0.0407

Table 3.3: Dependence Community Statistics

3.5 The Semantic Nature of Dependence Communities

The empirical study found that dependence communities do exist in software but what do these mean? Recall that a community is a sub-graph with a higher than expected number of internal connections. In our situation this means sets of statements in a program that have high dependence between them. It seems plausible that such collections of statements will be part of the same functional behaviour of the program.

In this section, we investigate this by manually inspecting a number of programs and their dependence communities to see if, like the *sumproduct* example, the communities closely approximated the separate semantic concerns of the program.

3.5.1 GNU wc

In the GNU *wc* program - that counts lines, characters and words in a text file - there are two communities detected; this can be seen graphically in fig. 3.4, page 80. The two communities are, broadly speaking, the 'counting community' and the 'input/output community'.

The 'counting community' consists of the parts of the program which deal with counting the values of lines, characters and words in a file; this includes nodes that iterate through the characters in a file, nodes that increment counters, and nodes that deal with checking if a string is a word.

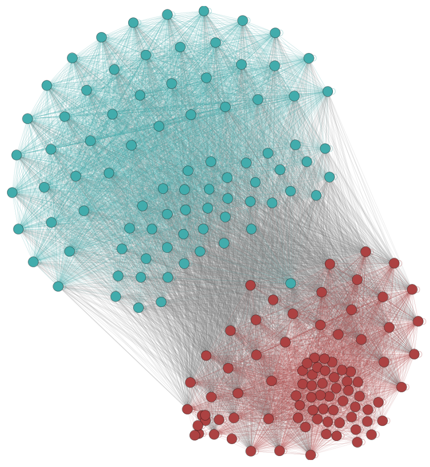


Figure 3.4: Communities detected in the *wc* program

The 'input/output' community contains nodes which deal with the opening of the file, printing of error messages and printing of the results of the 'counting community'.

It is clear from the diagram that there are two distinct communities - the advantage of using a force-directed layout algorithm is that the nodes that are more tightly connected are placed close together; this is exactly the same idea behind community detection. Nodes that have more edges between them than with the rest of the graph will be put into a community.

3.5.2 GNU bc

The program *bc* is an "arbitrary precision numeric processing language" [69] which is a utility included in the POSIX standard. The program parses input from the user, translates it into bytecode and executes the bytecode. In the program there are two main communities detected – the parser and the calculator. The communities can be seen graphically in fig. 3.5 where each section of the ring depicts a community as a percentage of program so. These two communities combined make up 96% of the program; the parser community is 51% of the program and the calculator community is 45%.

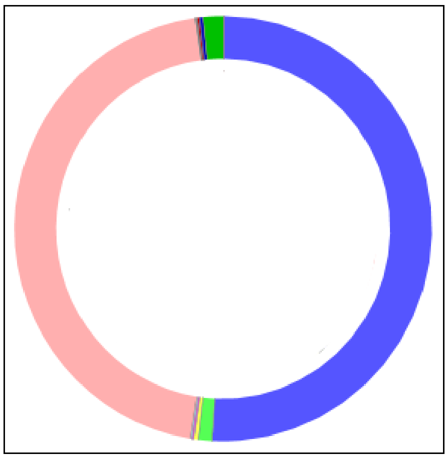


Figure 3.5: GNU bc. Communities as a percentage of program size.

3.5.3 GNU Chess

GNU Chess [70] is another program that exhibits clearly defined communities which correspond to the modules of the program. The program is composed of three loosely-coupled modules: the fronted, adapter and engine; the adapter sits in between the front-end and the engine. The three main communities detected in the BSG correspond to the three components of the software. The

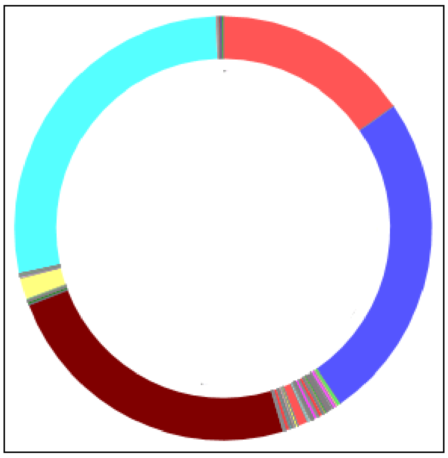


Figure 3.6: GNU Chess. Communities as a percentage of program size.

developers of the software intended the components to be loosely-coupled which has resulted in the distinct communities. If coupling was high there would be more edges in the BSG between components and it would therefore be less likely that they would be separate communities. This indicates that community detection could potentially be applied to the problem of software metrics for the calculation of coupling between modules in software. If the program has a community that expands across functional areas it may indicate that the program has high coupling; on the other hand, if the detected communities closely match the functional areas of a program we can be confident that the modules are loosely-coupled. The communities can be seen graphically in fig. 3.6 where each section of the ring depicts a community as a percentage of program so.

3.5.4 GNU Robots

GNU robots is a program with low coupling between procedures. The user interface is written in C which interacts with an external Scheme program. This causes the coupling between procedures to be very low because many procedures communicate with the external program rather than with each other. In turn, this causes a large number of communities as there are many areas of highly dependent code with few edges between them. The communities can be seen graphically in fig. 3.7 where each section of the ring depicts a community as a percentage of program so.

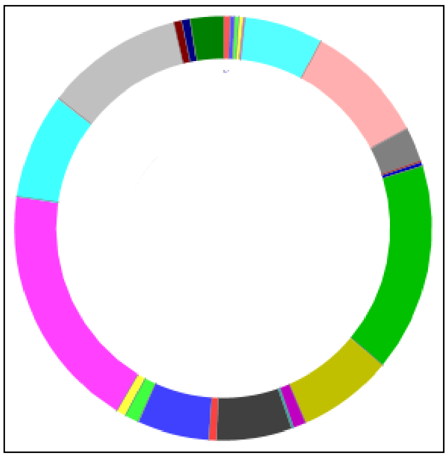


Figure 3.7: GNU Robots. Communities as a percentage of program size.no

3.5.5 Watermark Communities

Software watermarking involves embedding a unique identifier within a piece of software, to discourage the copying of software [67, 75, 77]. Watermarking does not prevent copyright infringement but instead discourages it by providing a means to identify the creator of a piece of software and/or the origin of copied software. A hidden watermark can be extracted, at a later date, to prove ownership.

Watermark code can be protected by *opaque predicates* [40] which attempt to force the original program code to depend on the watermark and the watermark code to depend on the original code. However, it is difficult to make the watermark fully integrated into the original program. Preliminary work suggests that community detection could be used to uncover hidden watermarks as the additional watermarking code tends to reside in its own communities.

For example, fig. 3.8a shows the BSG of a Java program that has been injected with a dynamic watermark [41]; fig. 3.8b shows the communities detected using the *Louvain method* in which the majority of the watermark code is in its own community.

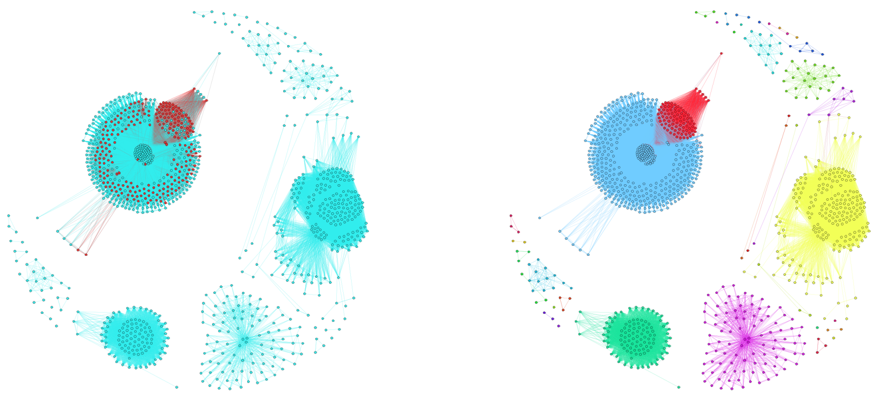

(a) Highlighted Watermark (b) Highlighted Communities

Figure 3.8: A Java program's BSG with highlighted watermark (red) and detected communities

3.6 Conclusion

This work is the first to investigate community structure at the statement-level in software. We introduced the concept of a *dependence community* and have shown that BSGs have community structure. We manually inspected a number of programs and found that the communities seem to approximate to the semantic concerns of those programs.

There is little point in breaking a piece of software into smaller components if these components do not in some way reflect different *functionalities*. We, therefore, believe that good semantic separation is the key to the usefulness of partitioning techniques in software engineering.

Of course no automated approach can perform perfect semantic separation. Our hypothesis is that dependence communities computed using community detection algorithms applied to program graphs can give *sufficiently good* semantic separation to be highly applicable in a number of areas of software engineering. This new approach has applications in all areas of software engineering where system decomposition is important, including software comprehension & maintenance, reverse engineering & restructuring, software evolution and information recovery.

We have described an empirical study of 44 open-source programs, analysing a total of 464,621 lines of code; the study applied the well-known *Louvain method* to program dependence graphs generated from a total of 1,725,800 backward slices.

The *Louvain* algorithm has previously been successfully used on large graphs of up to 118 million nodes and 1 billion edges. We have successfully applied the algorithm to graphs with up to 305,000 nodes and 16 billion edges. We showed that each of the 44 BSGs exhibited community structure, as evidenced by the positive modularity obtained by partitioning them using the *Louvain method*.

The modularity for the set of 44 programs in the empirical study varies, with some programs showing a stronger community structure than others. It is not yet clear what 'stronger community structure' means in terms of software dependence but the positive findings merit further work. We suspect that there is a connection between high modularity and how well a program is separated into its different semantic concerns.

We manually inspected a number of programs to answer the question "what do dependence communities in software mean?" The analysis revealed that dependence communities found in those programs seem to reflect the functional behaviour or semantics of the program; for example in the GNU *wc* program, we found two dependence communities: the *counting* community, which consists of the parts of the program which count the number of lines, characters and words in a file, and the *I/O* community which contains statements which opens the file, prints error messages and prints results.

Being able to break down software into meaningful sub-components is one of the major challenges in software engineering. Previous research in the area has focused on the clustering of high-level components in a software system to recover a modular structure or re-modularise software. The Bunch tool [129, 133], for example, works on a Module Dependency Graph (MDG) which includes high-level system components such as Java classes or C files that are connected due to dependence.

Our results provide compelling evidence that there is great potential for using dependence communities as an alternative software clustering methodology. Our results have shown that there is merit in further investigations of dependence communities and their application to various software engineering areas.

CHAPTER 4

Maximal-Slice-Based Cohesion and Coupling Metrics

Slice-based cohesion and coupling metrics attempt to quantify the inter-dependence between statements in program modules and between modules. Current definitions of sliced-based metrics are not defined for procedures which do not have clearly defined output variables and definitions of output variable vary from study-to-study.

We solve these problems by introducing corresponding new, more efficient forms of slice-based metrics in terms of maximal-slices. These metrics have the advantage that they do not require a definition of output variable and are also, therefore, defined for all procedures irrespective of whether or not output variables exist

We report on an empirical study which shows that where output variables exist, there is a strong correlation between the values of the metrics computed using both output-variables and maximal slices. We give a practical algorithm for computing sliced-based metrics in terms of *pseudo-maximal* slices which is also more efficient than calculating slice-based metrics using output variables.

4.1 Introduction

4.1.1 The Output Variable Problem

The calculation of slice-based cohesion and coupling metrics poses a problem: what are the slicing criteria? Previously, the concept of an output variable has been used. There are several problems with the current definitions of slice-based metrics which we call the 'The Output Variable Problem':

- It is difficult to define or capture output variables, especially of the type 'any variables printed, written to a file, or otherwise output to the external environment'.
- The definition of output variables varies from study to study.
- Not all functions have output variables (for some definition of output variables).
- output variables do not always capture the true cohesion of a module.

The choice of slicing criteria is a difficult problem to solve and various combinations of criteria may lead to different results for cohesion and coupling values. It turns out that the values of the metrics for modules can be highly sensitive to the choice of slices used in their computation. Many authors have grappled with the problem of which slices to include in order to come up with meaningful values for the metrics. We feel that the choices appear somewhat arbitrary and are in some sense trying to second-guess what the slices of interest are.

4.1.2 Definitions of Output Variable

Previous studies, and the study in this thesis, are based on a program's System Dependence Graph (SDG), generated by CodeSurfer.

Different studies have used different definitions for what 'counts' as an output variable. In practice, choosing what counts as output variables is equivalent to deciding which SDG nodes to slice with respect to.

Some [13, 80, 81, 99, 149] use a function's return value, global variables modified by a function and printed (or similarly output) variables (Definition 4.2). Others [130, 132, 152] use only a function's return value and globals modified by a function (Definition 4.3).

In addition to the first two definitions of output variable, for the purposes of our study, we define two further sets (Definitions 4.4 and 4.5) which include the `return` node of a function. These are included as it seems a more intuitive choice when thinking about slicing for slice-based cohesion and coupling metrics (i.e. rather than one slice, we have the same number of slices as return statements).

We do not attempt to exactly replicate the study by Meyers and Binkley [131] but merely provide a base-line to compare the maximal slice metrics against. Bowes et al. [27, 28] attempted to replicate the Meyers and Binkley study but failed to obtain the same results. In this section we define output variables based on descriptions in previous studies (see table 2.1, page 55 for definitions of output variable used in previous literature).

Definition 4.1 (Printed (or Similarily Output) Variable) *An input (*`actual-in`*) to an ANSI C* `stdio` *output function call, e.g.* `printf`*,* `puts`*,* `fwrite` *etc.*

Definition 4.2 (Output Variables 1 (OV1)) *$OV1$ is the set of SDG nodes matching the following conditions: v_i is a function's return value or the return value of a global variable modified by a function (formal-out and global-formal-out). Or v_i is an input (actual-in) to a C* `stdio` *output function call, e.g.* `printf`*.*

Definition 4.3 (Output Variables 2 (OV2)) *$OV2$ is the set of SDG nodes matching the following conditions: v_i is a function's return value or the return value of a global variable modified by a function (formal-out and global-formal-out).*

Definition 4.4 (Output Variables 3 (OV3)) *$OV3$ is the set of SDG nodes matching the following conditions: v_i is a function's* `return` *or* `normal-return` *node or the return value of a global variable modified by a function (global-formal-out). Or v_i is an input (actual-in) to a C* `stdio` *output function call, e.g.* `printf`*.*

Definition 4.5 (Output Variables 4 (OV4)) *$OV4$ is the set of SDG nodes matching the following conditions: v_i is a function's* `return` *or* `normal-return` *node or the return value of a global variable modified by a function (global-formal-out).*

4.1.2.1 The Effect of Different Definitions of Output Variable on Slice-Based Metrics

Previous authors have used many different definitions of output variable which has resulted in obtaining different results for slice-based metrics [27, 28]. Table 4.1 shows an example where the choice of output variable results in a large difference in slice-based cohesion metrics.

```
           int a, b, c, d,
                  e, f, g;

           int example() {
||||||| |      a = 0;
 |||||| |      b = a + 1;
  ||||| |      c = b + 1;
   |||| |      d = c + 1;
    ||| |      e = d + 1;
     || |      f = e + 1;
      | |      g = f + 1;

        |      return g;
           }

|          a_out
 |         b_out
  |        c_out
   |       d_out
    |      e_out
     |     f_out
      |    g_out
        |  formal-out
```

```
    int example() {
       int a, b, c, d,
              e, f, g;

|       a = 0;
|       b = a + 1;
|       c = b + 1;
|       d = c + 1;
|       e = d + 1;
|       f = e + 1;
|       g = f + 1;

|       return g;
    }

|   formal-out
```

$tightness(example) = \frac{1}{16}$	$tightness(example) = 1$

Table 4.1: The choice of output variable can greatly affect the value of cohesion.

The tightness for the first example is $\frac{1}{16}$ whereas for the second example is 1. This difference results from the definition of tightness where the size of the intersection of slices is the numerator and the size of the procedure the denominator. In the first example, the intersection only includes the first line; in the second example there is only one slice so the intersection is equal to the size of the module. In this example, it is clear that an arbitrary decision regarding the choice of output variable can severely affect the value of the metrics. Whether, or not, the variables used in the procedure are global or local has no bearing on the inter-relatedness of the statements within the procedure; in each case the statements are inter-related as much as the other.

In some cases the value of the metric may not reflect the expected value when analysed subjectively by a programmer, as is the case in table 4.1 if `global-formal-outs` are used as slicing criteria.

Consider the program in listing 4.1, which does nothing but print the value passed as a parameter to it. In this case, depending on the choice of output variables the value of cohesion may be undefined.

Listing 4.1: Simple C Procedure

```
void g(int x) {
      printf("%d\n", x);
}
```

If we use OV1 (definition 4.3), then in listing 4.1 we would then achieve the desired tightness value of 1 as we would have 1 output variable. However, now consider listing 4.2 – in this case we have a call to the GTK library *printf* procedure rather than the C library call; we are back to having an undefined metric. It seems unreasonable to be required to define all possible output procedures used in a program in order to calculate slice-based metrics.

Listing 4.2: Simple C Procedure 2

```
void g(int x) {
      g_printf("%d\n", x);
}
```

4.1.2.2 Non-existence of Output Variables

In a previous study [20] that used the OV1 definition it was found that 1,071 of 1,444 procedures have no output variables. So for these procedures the slice-based metrics are not defined at all. To verify this previous experiment, we conducted an empirical study using the four definitions of output variable to our set of 44 programs.

Using each definition of output variable, we counted the percentage of procedures in a number of programs which had output variables. The results of this experiment can be seen in table 4.2.

Type	**Mean percentage of procedures without output variables per program**	**Percentage of procedures without output variables in worst-case program**
OV1	5.23%	45.02%
OV2	7.6%	46.92%
OV3	4.87%	37.44%
OV4	7.02%	44.72%

Table 4.2: Percentage of procedures for which metrics are undefined using output variables, using each definition.

Although our results were not as extreme as the previous study, they confirmed that many procedures have no output variables and hence no values for slice-based metrics. Furthermore, our experiment showed that even by changing the definition of output variable the problem still persisted.

An alternative definition that avoids this problem is required. The next section proposes a solution: maximal-slice-based metrics.

4.2 Maximal-Slice-Based Metrics

We propose the use of maximal slices as the basis of the definition of slice-based cohesion and coupling metrics. In our definition there is no need to define the concept of an output variable. This avoids the problems discussed in the previous section. A maximal slice is simply a slice which is contained in no other slices[*].

The motivation for this is as follows: if a slice is not maximal it is, by definition, a subset of another slice and rather than performing a 'whole' task of its own, could be thought of as contributing towards the 'larger' task of the larger slice. We claim that the output variables are not of interest in themselves. They are simply used as slicing criteria to produce the slices on which the metrics are based. In our approach we use the set of maximal slices (corresponding to 'complete' tasks) in a module instead.

Disjoint maximal slices would suggest that the set of statements in each slice are performing completely separate tasks; for example, if we calculate maximal slices for a procedure which performs two distinct tasks we would find two disjoint maximal slices. Ideally, the intersection of maximal slices of a procedure should be large, corresponding to a highly inter-related set of program statements.

[*]Previously, maximal decomposition slices [60–64, 111, 172] have been used for software maintenance. A decomposition slice with respect to variable v is the union of slices taken at all points where v is output, and the last statement of the program. These are not the same as maximal slices.

4.2.1 Definitions

Definition 4.6 (Maximal Slice) *A maximal slice is a slice that is not a subset of any other slice. We define MaxSlices(M) to be the set of all maximal slices of module M.*

Definition 4.7 (Crucial Point) *A crucial point is a slicing criterion which produces a maximal slice.*

For cohesion metrics we use *local slices* [14] which are obtained by intersecting an inter-procedural slice with a particular procedure.

Definition 4.8 (The Set of all Local Slices in a Module) *Given a module M we define LocalSlices(M) to be the set of all local slices of M.*

$$\mathit{LocalSlices}(M) = \{M \cap k \mid k \in \mathit{Slices}(M)\}$$

Definition 4.9 (The Set of all Maximal Local Slices) *Given a module M we define MaxLocSlices(M) to be the set of all maximal local slices of M. Formally,*

$$s \in \mathit{MaxLocSlices}(M) \iff \begin{cases} s \in \mathit{LocalSlices}(M) \\ \mathit{and} \\ \forall k \in \mathit{LocalSlices}(M), \\ \qquad s \subseteq k \implies s = k. \end{cases}$$

Definition 4.10 (The Intersection of all Maximal Local Slices) *Given a module M we write $\mathcal{I}(M)$ for the intersection of all maximal slices. Formally,*

$$\mathcal{I}(M) = \bigcap_{s \in \mathit{MaxLocSlices}(M)} s$$

4.2.1.1 Maximal-slice-based Metrics

We now define our new maximal-sliced-based metrics. Importantly, these definitions agree with the previous definitions in the sense that both the old and the new are defined in terms of a set of slices. The only difference is that in the old, this set is the intersection of all slices $\left|\bigcap_{v \in V_O} \mathrm{SL}_v\right|$ corresponding to output variables whereas in the new it is the intersection of all *maximal local slices* (Definition 4.10).

Definition 4.11 (Maximal-slice-based Tightness)
The tightness of a module is the ratio of the intersection of all maximal local slices to the size of the module:

$$\mathit{tightness}(M) = \frac{|\mathcal{I}(M)|}{|M|}$$

Tightness is, thus, the ratio of the intersection of all maximal slices to the size of the module.

Definition 4.12 (Maximal-slice-based Minimum Coverage)

$$\mathit{mincoverage}(M) = \frac{\mathit{min}\{|s| : s \in \mathit{MaxLocSlices}(M)\}}{|M|}$$

Definition 4.13 (Maximal-slice-based Coverage)

The coverage is the ratio of the average maximal slice to the size of the module:

$$coverage(M) = \frac{\sum\limits_{s \in MaxLocSlices(M)} |s|}{|M||MaxLocSlices(M)|}$$

Definition 4.14 (Maximal-slice-based Maximum Coverage)

The maximum coverage is the ratio of size of the largest maximal local slice to the size of the module:

$$maxcoverage(M) = \frac{max\{|s| : s \in MaxLocSlices(M)\}}{|M|}$$

Definition 4.15 (Maximal-slice-based Overlap)

The overlap is the average ratio of the intersection of all maximal local slices to slice size:

$$overlap(M) = \frac{|\mathcal{I}(M)|}{|MaxLocSlices(M)|} \sum_{s \in MaxLocSlices(M)} \frac{1}{|s|}$$

Maximal-slice-based coupling is defined in terms of maximal-slice-based flow between two modules:

Definition 4.16 (Maximal-Slice-based Flow between two modules)

The maximal-slice-based flow between two modules M and N is written $M \hookrightarrow N$, is defined as

$$M \hookrightarrow N = N \cap \bigcup_{c \in crucial(M)} Slice(c)$$

where crucial(M) are the set of points in M which give rise to the maximal local slices of M and Slice(c) is the inter-procedural slice of the whole program with respect to c.

Definition 4.17 (The Maximal-slice-based coupling between two modules)

The maximal-slice-based coupling between two modules M and N is the normalised values of the maximal-slice-based flow in each direction.

$$\mathit{coupling}(M, N) = \frac{|M \hookrightarrow N| + |N \hookrightarrow M|}{|M| + |N|}$$

We generalise coupling of a module M to be the weighted average of the coupling between M and the other modules in the program:

Definition 4.18 (The Maximal-slice-based coupling of a Module)

$$\mathit{coupling}(M) = \frac{\sum\limits_{N\ \in\ \mathit{modules}(P)} |N|\ \mathit{coupling}(M, N)}{\sum\limits_{N\ \in\ \mathit{modules}(P)} |N|}$$

4.2.2 Examples

For computing cohesion, consider the example in listing 4.3 which contains 2 maximal slices: {9, 7, 6, 4, 3, 2} and {8, 7, 5, 4, 3, 1}. The crucial points are lines 9 and 8 as these give the maximal slices. Notice that the crucial points contain output variables. The length $|M|$ of the module is 9 and the intersection $\mathcal{I}(M)$ of the maximal slices is $\{7, 4, 3\}$, so the size of the intersection is 3.

Listing 4.3: Maximal-slice-based cohesion example

```
1 sum = 0;
2 product = 0;
3 i = 0;
4 while(i < 10) {
5   sum = sum + i;
6   product = product * i;
7   i = i + 1;}
8 print sum
9 print product
```

Now, using our definitions we can calculate cohesion metrics for this example:

$$\text{tightness}(M) = \frac{|\mathcal{I}(M)|}{|M|} = \frac{3}{9} = \frac{1}{3}$$

$$\text{mincoverage}(M) = \frac{\min\{|s| : s \in \text{MaxLocSlices}(M)\}}{|M|} = \frac{6}{9} = \frac{3}{3}$$

$$\text{coverage}(M) = \frac{\sum\limits_{s \in \text{MaxLocSlices}(M)} |s|}{|M||\text{MaxLocSlices}(M)|} = \frac{3+3}{9 \times 2} = \frac{1}{3}$$

$$\text{maxcoverage}(M) = \frac{\max\{|s| : s \in \text{MaxLocSlices}(M)\}}{|M|} = \frac{6}{9} = \frac{2}{3}$$

$$\text{overlap}(M) = \frac{|\mathcal{I}(M)|}{|\text{MaxLocSlices}(M)|} \sum_{s \in \text{MaxLocSlices}(M)} \frac{1}{|s|} = \frac{3}{2}(\frac{1}{6} + \frac{1}{6}) = \frac{1}{2}$$

For computing coupling, consider the example in listing 4.4. The coupling between f and g can be computed from the crucial points – the `return` statements on lines 3 and 9. The slice with line 9 as the criterion contains lines $\{9, 8, 7, 6, 3, 2, 1\}$ and the slice with line 3 as the criterion contains lines $\{3, 2, 1, 8, 6\}$.

Listing 4.4: Maximal-slice-based coupling example

```
1 int f(int x) {
2   x += x;
3   return x;
4 }
5
6 int g(int x) {
7   x +=
8     f(x);
9   return x;
10 }
```

The coupling between f and g is computed as the number of statements from maximal slices originating in g that are in f and the number of statements from maximal slices originating in f that are in g divided by the total number of statements in f and g. In this example, the coupling between f and g is $\frac{3+2}{3+4} = \frac{5}{7}$.

4.3 Empirical Study

In this section we compute slice-based cohesion and coupling metrics for the same set of 44 programs, used in chapter 3. In our study, we calculated the 'old' slice-based metrics using four definitions of output variable (see section 4.1.2, page 88), and the 'new' maximal-slice-based metrics (defined in section 4.2.1, page 94) for each of procedures which contained output variables.

4.3.1 Research Questions

Is there a correlation between the values of metrics computed using maximal slices and those computed using output variables?

> If we find a strong correlation between our new, maximal-slice-based metrics and the old, output variable based metrics the new metrics would be applicable for *all* procedures, including those without output variables. Additionally, this would remove the need to define slicing criteria to compute slice-based metrics.
>
> We show that there is a strong correlation between output-variable-based metrics and the new maximal-slice-based metrics, for procedures which contain output variables

4.3.2 Method

In order to compute maximal-slice-based metrics, we used CodeSurfer to compute backward slices for each SDG node, excluding pseudo-nodes generated by CodeSurfer (except `global-formal-outs`), but retaining the nodes that represent source-code statements. We calculated output variable based metrics using the subset of slices whose criterion corresponded to the output variable definitions. We then computed the maximal slices from the set of slices obtained from CodeSurfer.

4.3.3 Results

Figure 4.1 shows the weighted average metric values using maximal slices, ordered by tightness, showing the wide range of values for all metrics.

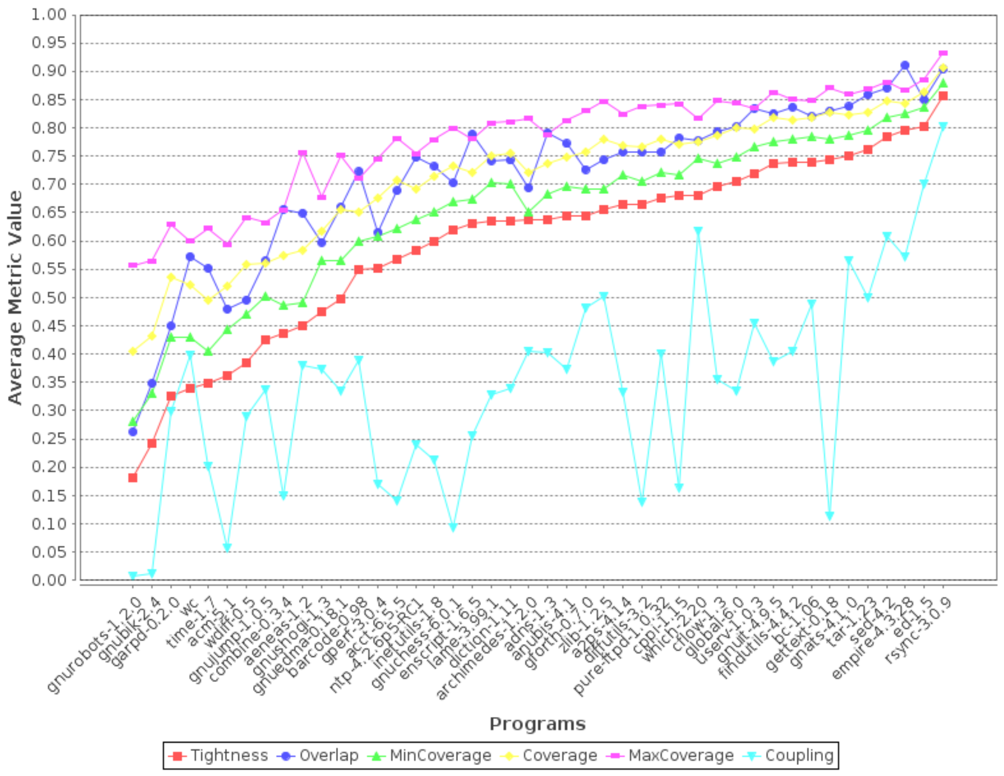


Figure 4.1: Weighted averages of the metrics calculated using maximal slices.

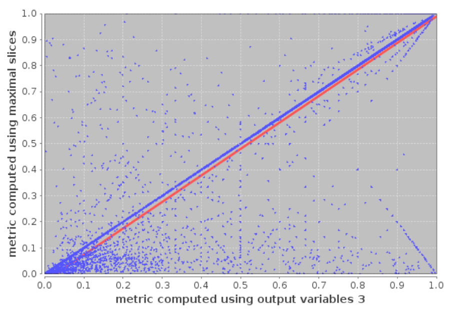
There is a strong correlation between the metric values obtained using maximal slices and those obtained using the four definitions of output variable, for procedures where metrics are defined; table 4.3 shows the Pearson correlation coefficient for maximal slice metrics and the four output variable definitions. The minimum correlation coefficient is 0.523 for overlap using maximal slices and OV2. The correlation coefficient for coupling using maximal slices and all output variable definitions is > 0.95 indicating that coupling is the least sensitive to the choice of slicing criteria.

Figure 4.2 plots the values for metrics calculated using maximal slices against the values calculated for output variables; the pair with the highest correlation coefficient is plotted for each metric.

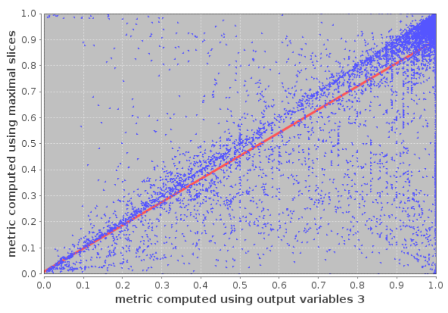
Each plot shows a clear, strong correlation between the values of metrics calculated using maximal slices and those calculated using output variables. Overlap has many high values for both maximal slice metrics and output variable metrics and thus has a large clustering of points in the top-right corner. Coupling has an extremely good correlation and most points are clustered around the diagonal.

Program	**Correlation**			
	MX/OV1	**MX/OV2**	**MX/OV3**	**MX/OV4**
Tightness	0.839	0.809	0.939	0.934
Overlap	0.678	0.603	0.719	0.651
MinCoverage	0.851	0.823	0.941	0.937
Coverage	0.929	0.911	0.949	0.945
MaxCoverage	0.964	0.959	0.850	0.840
Coupling	0.974	0.964	0.975	0.966

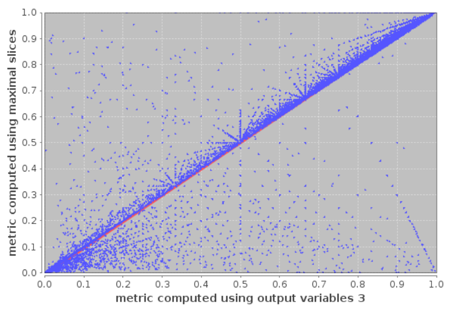
Table 4.3: Correlation between metrics calculated using maximal slices and output variables.



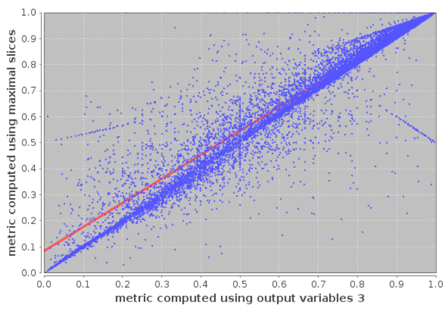
(a) tightness (0.939)



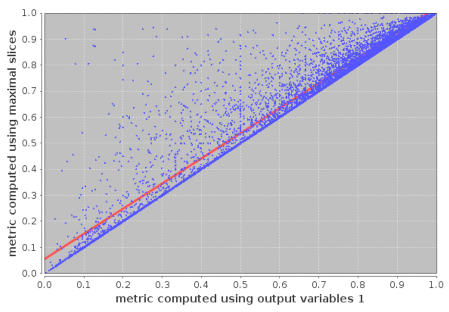
(b) overlap (0.719)



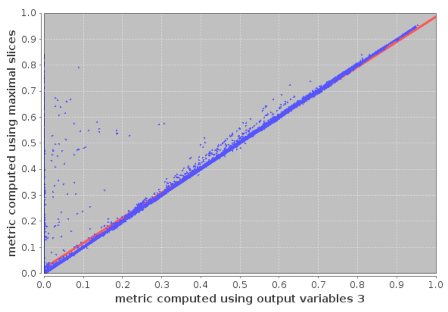
(c) mincoverage (0.941)



(d) coverage (0.949)



(e) maxcoverage (0.964)



(f) coupling (0.975)

Figure 4.2: Metrics calculated using maximal slices vs output variables

4.3.4 Analysis of Results

We have shown a strong correlation between the values of metrics obtained using maximal slices and output variables, for procedures that contain output variables. These results provide evidence that using maximal slices is a valid technique for computing slice-based cohesion and coupling metrics for all procedures, including those without output variables.

The majority of the 44 programs have some procedures for which metrics are undefined (see table 4.2, page 92). The highest number of procedures for which metrics are undefined is 45.50% of the procedures in *gnubik-2.4*. Only 8 programs have metrics defined for all procedures when using OV1 and OV3. This is due to the fact that not all procedures print or ouput values to a file, and not all procedures return a value. Some procedures could be said to print a value, such as in listing 4.5, page 104, but we must define which procedures print values. In this study the definition of printing a value is the use of the standard C output functions; if a program calls a different procedure to print a value or output to a file this will not be included in the set of output variables.

Listing 4.5: Procedures from which-2.20.

```
/* Print error message and exit with error status. */
static void errf (char *fmt, ...) {
  va_list ap;
  va_start (ap, fmt);
  error_print (0, fmt, ap);
  va_end (ap);
}
```

We have shown a strong correlation between the values of metrics obtained using maximal slices and output variables where metrics are defined but, from the scatter plots in fig. 4.2, page 103, it can be seen that there are cases where these differ. The requirement to define output variables and the use of different definitions of output variable may lead to different slicing criteria. The choice of slicing criteria is crucial in obtaining intuitive values for cohesion and coupling; making arbitrary choices about the inclusion or exclusion of certain nodes is not ideal.

Metrics using output variables, in many cases, produce values higher than one would expect for cohesion because if a function contains no printed variables or modified globals there will be only one slice (`formal-out`) in the module resulting in a high intersection and high tightness. Maximal slices tend to give a lower value of tightness due to the greater number of maximal slices in a module; for example, in *tar* there are 3 separate tests which form part of *uc_width* – testing for 'for non-spacing or control character', 'for double-width character' and 'ancient CJK encodings'; the tightness for this procedure is low with maximal slices, which seems to more accurately reflect cohesion than a very high value given by output variables.

Most procedures do not have printed (or similarly output) variables which leaves only `formal-outs` and `global-formal-outs` as the output variables; such slices will tend to include the whole procedure in the slice as they are the last statement in the procedure. Using output variable metrics, in a scenario where there are many modified globals, would result in a slice for each `global-formal-out` and potentially a high intersection value; whereas maximal-slice-based metrics will only use the locally maximal slices, potentially giving a different result. Additionally, procedures can have only 1 formal-out which means many procedures could have a high cohesion even though internally they many be performing many sub-tasks related to the calculation of the final value.

If a procedure uses few global variables (which is assumed to be a good thing) and does not print (or similarly output any values) then using *output variables* to calculate cohesion metrics will automatically give high values, with little relation to the cohesion of the internal code (other than all the code contributes to the result of the procedure). Procedures with multiple `returns` will have a lower tightness value when using maximal slices compared with using just `formal-outs` as the output variables. Metrics calculated using maximal slices take into account other slices in within the procedure, giving a more detailed analysis of the cohesion of the procedure.

4.4 Efficiency of Maximal-Slice-Based Metrics

Clearly, calculating maximal-slice-based metrics is less efficient than computing slice-based metrics using output variables. This is for two reasons:

1. In order to compute maximal-slice base metrics for a procedure, we have to slice at every point in the procedure to find which of the slices are maximal.
2. Having computed all the slices, finding the maximal ones takes considerable computational effort. The problem of finding maximal subsets or *extremal sets* as they are called has been studied [157]. Until recently the best algorithms for doing this, as is the algorithm used in our study, were $O(N^2/log(N))$ where N is total number of elements in all the sets.

Figure 4.3, page 107 shows the run times for calculating the maximal slices for each of the 44; many were computed in seconds, while the longest took 13 minutes.

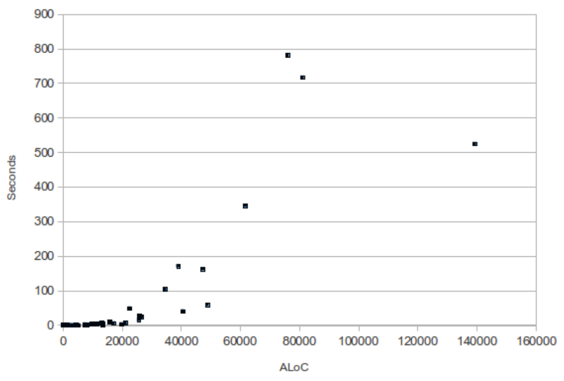


Figure 4.3: ALoC vs Runtime, for calculating maximal-slice-based metrics

In our empirical study, the situation is worse than it needs to be: we produce slices using CodeSurfer which we regard as a 'black box'. It is unlikely that such a naive approach would scale up.

4.5 An Efficient Algorithm Using Pseudo-Maximal Slices

We take advantage of the *near transitivity* of data and control dependence [47] to develop a much more efficient approach to computing extremely accurate approximations to maximal-slice-based metrics. By near-transitivity we mean that if a is in the slice of b and b is in the slice of c then a is *almost always* in the slice of c.

If we assume transitivity we can find maximal slices very quickly, since if slicing at point p gives us the set S then slicing at any of the points in S cannot result in a set bigger than S. So we can discard all elements of S (apart from p) in the search for maximal slices. If we start looking at points at the end of the procedure first as this is where the slicing criteria for maximal slices are most likely to be, we very quickly eliminate other points in our procedure for further consideration. Algorithm 1 is used to find all the *pseudo-crucial* points of a function f.

Algorithm 1 Pseudo-Maximal Slices

```
$L \leftarrow$ all the slice points in function $f$ (in order)
$T \leftarrow \emptyset$
while $L \neq \emptyset$ do
    $p \leftarrow last(L)$
    $K \leftarrow slice(p) \cap f$          ▷ K is the local slice
    $L \leftarrow L - K$                    ▷ remove all elements in K from L
    $T \leftarrow (T - K) \cup \{p\}$
end while
```

After executing this algorithm, `T` will be the set of *pseudo-crucial* points for f i.e. the set of slicing criteria that give rise to the *pseudo*-maximal slices of f. We call the slices returned by the algorithms *pseudo*-maximal as in some cases they will not be maximal. For the purposes of calculating metrics this appears not to matter.

4.5.1 Research Questions

Is there a correlation between *pseudo*-maximal-slice-based metrics and maximal-slice-based metrics?

If there is a correlation between *pseudo*-maximal-slice-based metrics and maximal-slice-based metrics these can be used in place of maximal-slice-based metrics.

We show that there is a very strong correlation between *pseudo*-maximal-slice-based metrics and maximal-slice-based metrics; this provides strong evidence that we can use *pseudo*-maximal-slice-based metrics in place of maximal-slice-based metrics.

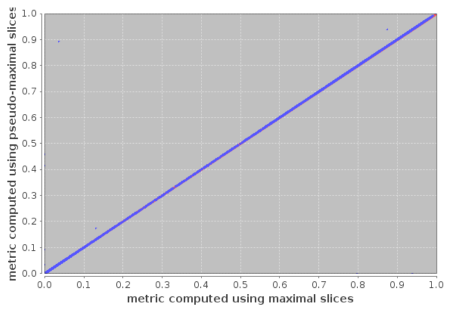
How many slices are required to compute *pseudo*-maximal-slice-based metrics compared to maximal-slice-based metrics and output variable based metrics?

The major bottle-neck in computing slice-based metrics is the number of slices that need to be computed. We hypothesis that a far few number of slices will need to be computed compared to maximal-slice-based metrics; if there is a correlation this will allow the efficient computation of accurate slice-based metrics.

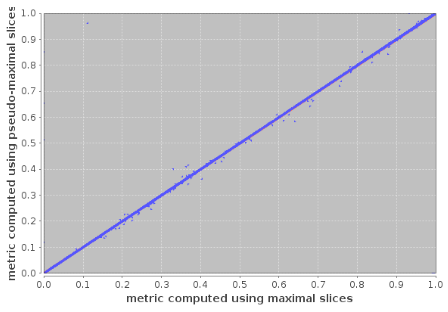
We show that there are far fewer slices to compute using emphpseudo-maximal-slice-based metrics compared to using maximal-slice-based metrics. We also show that there are fewer slices required to be computed than when using output-variable based metrics.

4.5.2 Results

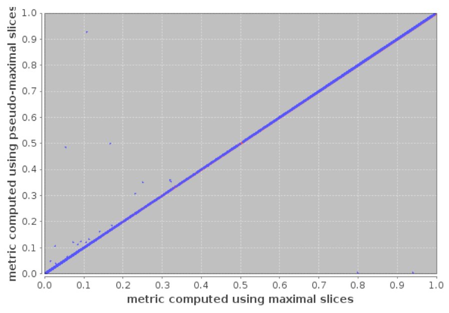
Figure 4.4 shows plots for each slice-based metric, comparing the values computed using pseudo-maximal slices versus those computed using maximal slices. There is a perfect correlation (a Pearson correlation coefficient of 1). What is more, the metrics were identical in all but a few cases. The *pseudo*-maximal slice approach correlates identically to the maximal-slice approach with all the output variable approaches (OV1–OV4).



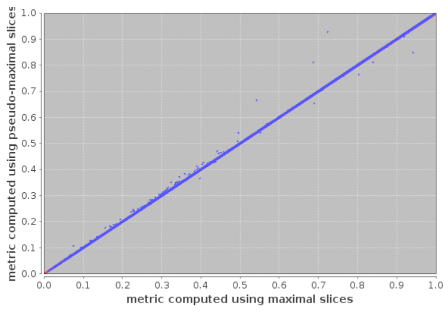
(a) tightness



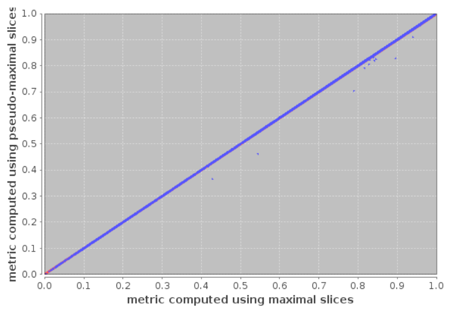
(b) overlap



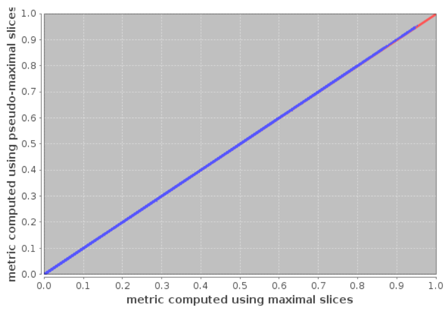
(c) mincoverage



(d) coverage



(e) maxcoverage



(f) coupling

Figure 4.4: Metrics calculated by maximal slices vs *pseudo*-maximal slices

Using the maximal slice approach, we have to slice at every single program point.

Using the output variable approach we have to slice only at points corresponding to output variables. The *pseudo*-maximal-slice algorithm requires far fewer slices to be computed, when compared to using maximal slices (see table 4.4).

It also turns out that the *pseudo*-maximal-slice algorithm required fewer slices to be computed than with the standard output variable approaches.

Method used for calculating slice-based metrics	Total number of slices calculated in order to compute all the slice-based metrics for every procedure in the study
Maximal slices	1,725,800
Pseudo-maximal slices	487,453
OV1	723,031
OV2	670,953
OV3	743,758
OV4	691,680

Table 4.4: The number of slices computed to calculate the slice-based metrics for all the procedures

4.6 Conclusion

We have introduced new forms of slice-based metrics based on maximal-slices, as a solution to 'The Output Variable Problem'. These metrics, unlike previous definitions, are defined for *all* procedures in a system and are strongly-correlated with the previous output-variable-based metrics.

There are several problems with the previous definitions (see section 2.5, page 52) of slice-based metrics which we call the 'The Output Variable Problem':

– It is difficult to define or capture output variables, especially of the type 'any variables printed, written to a file, or otherwise output to the external environment'.

– The definition of output variables varies from study to study.

– Not all functions have output variables (for some definition of output variables).

– output variables do not always capture the true cohesion of a module.

We undertook an empirical study where we calculated the cohesion and coupling values for 20,588 procedures in 44 programs using both the old output-variable-based definitions and our new maximal-slice-based definitions. We used 4 definitions of output variable which were based on descriptions in previous studies.

In the empirical study we found that about 5% of the 20,588 procedures considered did not have output variables at all, and so the slice-based metrics for these procedures is not defined.

Our results showed that there is a strong correlation between output-variable-based metrics and the new maximal-slice-based metrics, for procedures which contain output variables. This is strong evidence that the new maximal-slice-based metrics are a valid generalisation to all programs of the previous more restrictive sliced-based metrics.

The use of maximal slices is appropriate because a maximal slice captures an 'interesting' part of a program. Intersecting maximal slices represent inter-related program statements, while disjoint maximal slices suggest multiple tasks are being performed.

A limitation to the use of maximal-slice-based metrics is the greater number of slices required to be computed for the computation of the metrics. Unlike output-variable-based metrics slices must be computed for every slicing criterion, in order to be able to calculate the maximal slices. The total number of slices computed for the set of 44 programs was 1,725,800 and the number of output variables in the set of programs was between 670,953 and 743,758 (depending on which output variable definition is used). Therefore, approximately 1 million more slices had to be computed for the calculation of maximal-slice-based metrics, compared with output variable based metrics.

We took advantage of the near transitivity of data and control dependence [47] to develop a much more efficient approach to computing extremely accurate approximations to maximal-slice-based metrics. These were computed very quickly and led to metrics which were almost identical to their maximal-slice-based counterparts. Additionally, these *pseudo*-maximal-slice-based metrics required fewer slices to be computed than previous output variable based metrics.

CHAPTER 5

Dependence Clusters

A dependence cluster is a maximal set of program statements all of which are mutually dependent. Large dependence clusters may hinder software maintenance as a change in any of the members in a dependence cluster could affect any other member.

We conduct an empirical study using definitions of dependence clusters from previous studies. The results provide further evidence of the existence of large dependence clusters in software.

We also show, however, that over 75% of the dependence clusters found are not, in fact, ‘true’ dependence clusters.

5.1 Introduction

A dependence cluster is a maximal set of program statements all of which are mutually dependent (see section 2.4, page 45). The presence of large dependence clusters in a program could hinder software maintenance as changing any statement in a cluster could potentially impact all other statements in that cluster; Binkley and Harman [16] define a large dependence cluster as one that covers 10% or more of a program.

Although some dependence clusters may naturally occur in a program, unwanted dependence clusters may be the result of bad programming practice which could be refactored away. Binkley and Harman [16] use the term 'dependence pollution' for such unwanted and avoidable dependence clusters, and large dependence clusters are an example of a 'dependence anti-pattern' [21]. Global variables have been shown to be a cause of 'dependence pollution' [22].

Harman et al. [87] define a slice-based dependence cluster as a slice-based Mutually Dependent Set (MDS) not properly contained within any other slice-based MDS; where a slice-based MDS is a set of System Dependence Graph (SDG) nodes $\{n_1, \ldots, n_m\}$ $(m > 1)$, such that for all i, j, $1 \leq i, j \leq m$, $n_i \in \mathit{Slice}(n_j)$.

In the major empirical studies of dependence clusters [16, 87], slice-based dependence clusters are calculated by saying that statements are in a dependence cluster if and only if they have the same slice (which we refer to as a Type 2 dependence cluster); and approximate to this by saying that statements are in a dependence cluster if and only if they have a slice of the same size (which we refer to as a Type 1 dependence cluster).

Type 2 dependence clusters, however, are not 'true' dependence clusters – a set of SDG nodes whose slices are the same is an MDS but it is not always a maximal

MDS.

In this chapter we think of a 'true' dependence cluster as a stricter form of dependence community in a Backward Slice Graph (see section 3.2, page 71). A 'true' dependence cluster is a maximal community in a Backward Slice Graph (BSG) where every node is connected to every other node – in other words, it is a *maximal clique* in a BSG.

Definition 5.1 (True Dependence Cluster) *A 'true' dependence cluster is a maximal clique in a Backward Slice Graph.*

Not all Type 2 dependence clusters are 'true' dependence clusters and not all 'true' dependence clusters in a BSG are Type 2 dependence clusters. Consider the example source code in listing 5.1 and its BSG in fig. 5.1. There is one Type 1 / 2 dependence cluster (larger than size 1) in this program, $\{1, 5\}$ – lines 1 and 5 both have the slice $\{1, 2, 3, 4, 5\}$; both lines have the same size slice (Type 1) and the same slice (Type 2). The nodes $\{1,\ 5\}$ form a maximal clique – a 'true' dependence cluster.

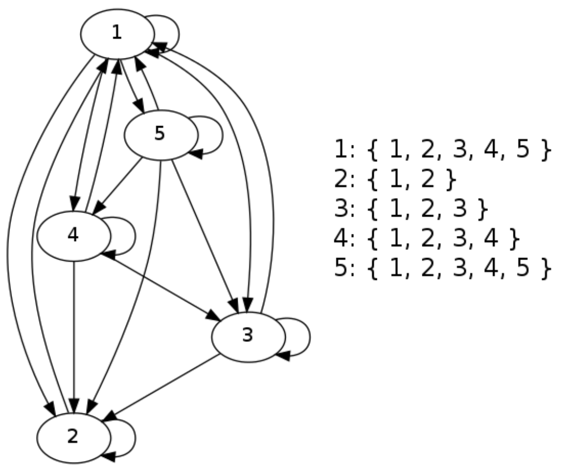
There are three further 'true' dependence clusters (i.e. maximal cliques): $\{1, 4\}$, $\{1, 3\}$ and $\{1, 2\}$; however, these are neither Type 1 nor Type 2 dependence clusters, as neither set contains nodes which give the same slices or same-size slices.

Now, consider the source code in listing 5.2 and its BSG in fig. 5.2. There are two Type 1 / 2 dependence clusters in this program – $\{1, 4\}$ and $\{2, 3, 5\}$; both dependence clusters have nodes whose slices are the same. The first, $\{1, 4\}$, is a maximal clique, however, $\{2, 3, 5\}$ is not a maximal clique.

The clique $\{2, 3, 5\}$ is not maximal because it can be extended by adding node 1, resulting in the maximal clique $\{1, 2, 3, 5\}$.

Listing 5.1: Dependence Cluster Example

```
1 f() { x=y+x; }
2 main() { x = f() + x;
3          x = k + x;
4          x = f() + x;
5          x = f() + x; }
```

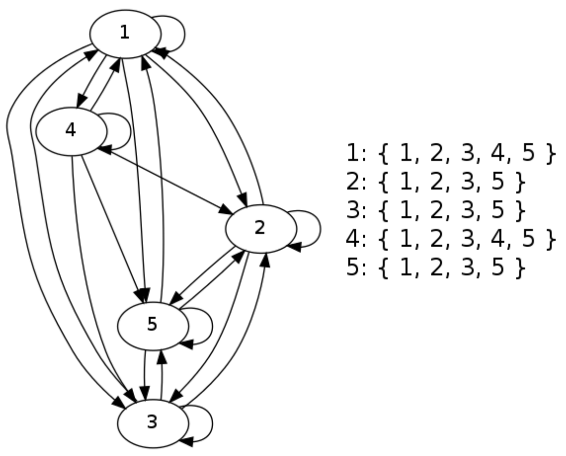


```
1: { 1, 2, 3, 4, 5 }
2: { 1, 2 }
3: { 1, 2, 3 }
4: { 1, 2, 3, 4 }
5: { 1, 2, 3, 4, 5 }
```

Figure 5.1: Dependence cluster example: not all 'true' dependence clusters are Type 1 or Type 2 dependence clusters

Listing 5.2: Dependence Cluster Example 2

```
1 f(x) { return x; }
2 main() { while(i < 10) {
3          i = f(i);
4          f(1);
5          i = i + 1; }
```



```
1: { 1, 2, 3, 4, 5 }
2: { 1, 2, 3, 5 }
3: { 1, 2, 3, 5 }
4: { 1, 2, 3, 4, 5 }
5: { 1, 2, 3, 5 }
```

Figure 5.2: Dependence cluster example: not all Type 1 or Type 2 dependence clusters are 'true' dependence clusters

5.2 Empirical Study

In this section we calculate Type 1 and Type 2 dependence clusters for our set of 44 programs to find out if Type 1 dependence clusters are a good approximation to Type 2 dependence clusters and discover how many of the Type 2 dependence clusters are maximal cliques in the BSG, i.e. how many are 'true' dependence clusters.

5.2.1 Research Questions

How prevelent are large dependence clusters in the set of 44 programs?

Harman et al. [87] have shown that large dependence clusters are prevelent in real-world software. Finding large dependence clusters in our set of 44 programs will provide further evidence of the dependence pollution in real-world software; if there are few, this may be evidence that there could have been a bias in the set of programs used in previous studies.

We show that, like previous studies, many programs have large dependence clusters; this result provides further evidence, supporting the findings of Harman et al. [87], of the *dependence pollution* created by dependence clusters in programs.

Are Type 1 dependence clusters a good approximation to Type 2 dependence clusters?

Harman et al. [87] introduced the Type 1 dependence cluster approximate to Type 2 dependence clusters and analysed 36 out of the 45 programs used in their empirical study to provide evidence that Type 1 dependence clusters are a good approximation to Type 2 dependence clusters. Due to the increase in computing power and the use of memory efficient data structures we can provide further evidence for or against this claim by computing both Type 1 and Type 2 dependence clusters for our whole set of programs.

We provide further evidence to support the findings of Harman et al. [87] by showing that Type 1 dependence clusters are indeed a good approximation to Type 2 dependence clusters. We show that there is a very strong correlation between both the number and the size of Type 1 and Type 2 dependence clusters and a high similarity of the Type 1 and Type 2 partitions of BSGs.

How many Type 2 dependence clusters are 'true' dependence clusters? Neither Type 1 dependence clusters nor Type 2 dependence are 'true' dependence clusters originally defined by Binkley and Harman [16]. We believe that 'true' dependence clusters are too strict a definition to be useful for finding groupings of statements related by control and data dependence and dependence clusters computed by previous studies have not been 'true' dependence clusters. How many of the Type 2 dependence clusters found are actually 'true' dependence clusters?

We show that not all Type 2 dependence clusters are 'true' dependence clusters – we show that over 75% of Type 2 dependence clusters in the set of 44 programs are not 'true' dependence clusters, i.e. they are not maximal cliques in the BSGs.

5.2.2 Method

In order to compute Type 1 and Type 2 dependence clusters we used CodeSurfer to compute backward slices for SDG nodes, as in chapter 3 (Dependence Communities) and chapter 4 (Maximal-Slice-Based Cohesion and Coupling Metrics). We then store each slice as a compressed bitset [37] to allow us to efficiently store them in memory and compare slices against one another (to test for slice equality).

5.2.3 Results

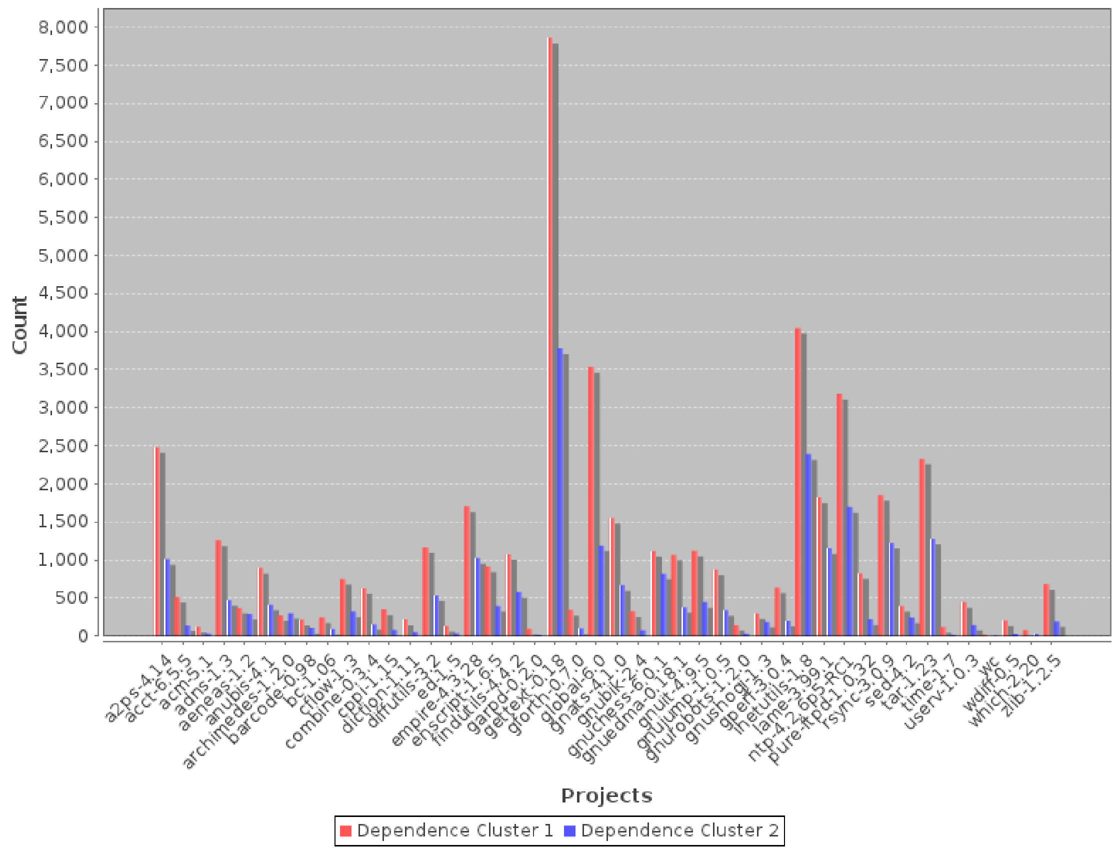
In the study we found that the program with the largest number of dependence clusters was *gettext-0.18* with 7,859 Type 1 and 3,780 Type 2 dependence clusters. The program with the largest dependence cluster was *ed-1.5* with a dependence cluster that covers 70.6% of the program; this observation was previously made by Binkley and Harman [16] who suggested that the program be refactored to group operations that affect the main data structure to break the large dependence cluster into smaller ones.

Figure 5.3a, page 121 shows the number of dependence clusters per program and fig. 5.3b shows the number of programs with a dependence cluster size greater than 10% of the program. Most programs have less than 2,000 dependence clusters and most have only 1 dependence cluster that is greater than 10% of the program size. *gforth* is the only program that has fewer Type 2 than Type 1 dependence clusters greater than 10% of the program code; two of the large Type 1 dependence clusters are not Type 2 clusters.

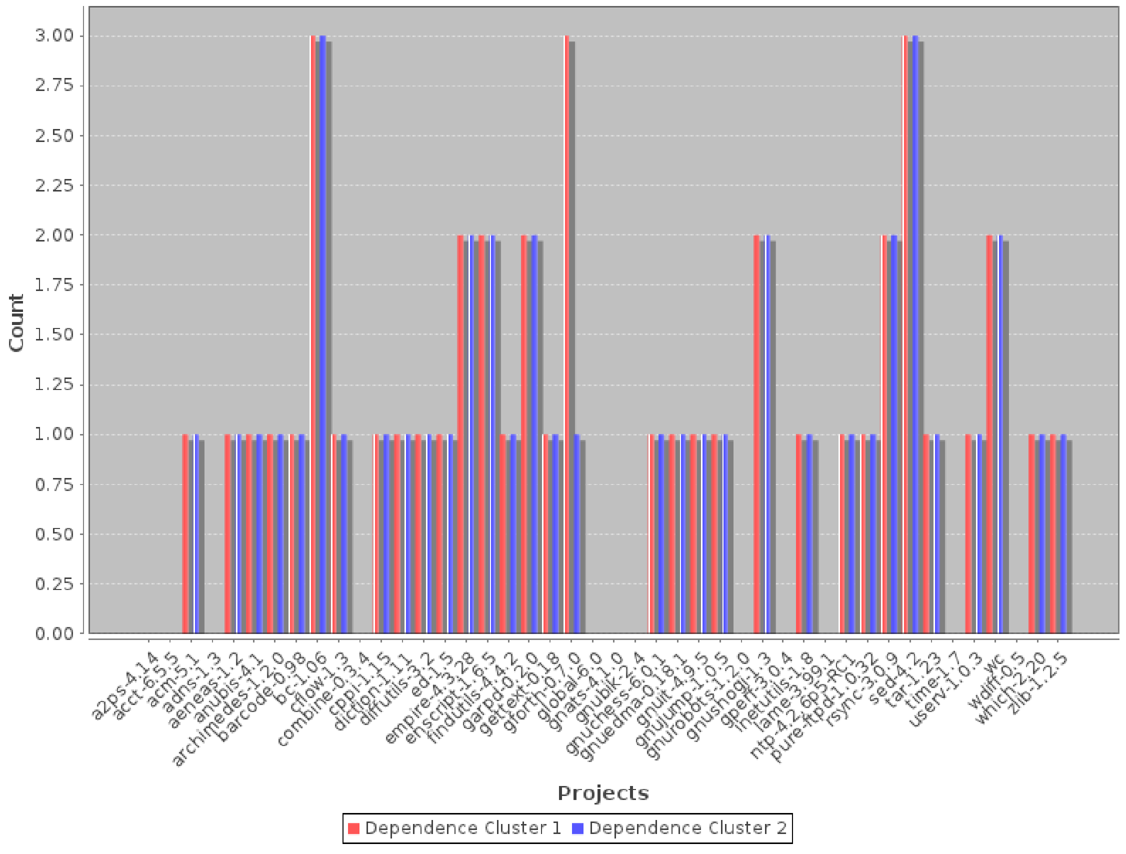
Figure 5.4, page 124 shows the Type 1 (centre) and Type 2 (outer) dependence clusters, scaled as a percentage of program size*. The complete set of partition charts for the 44 programs can be found in appendix A, page 160. Harman et al. [87] previously found that the 6 programs in fig. 5.4 contain large Type 1 dependence clusters.

The results of our study and Harman et al. study correspond well in some cases but differ in others. We used the latest versions of the programs available at the time of

*dependence clusters of size 1 are not included in the diagrams, leaving 'blank' areas in the rings; these are SDG nodes which give unique slices when used as slicing criteria.



(a) Number of Dependence Clusters



(b) Number of dependence clusters $> 10\%$

Figure 5.3: The number of Type 1 and Type 2 dependence clusters (Type 1 - Red, Type 2 - Blue)

the study but it is not clear which versions, of some of the programs, were used by Harman et al.[†].

GNU Barcode contains 1 large dependence cluster (both Type 1 and Type 2) and many very small Type 1 clusters. There are also 3 small Type 1 dependence clusters. This result is similar to that found by Harman et al.

GNU bc contains 3 large clusters; this version was used in previous studies and the result is similar to the discovery of 3 large *coherent dependence clusters* by Islam et al. [94] but not Harman et al. who found a large Type 1 dependence cluster that covered $\approx 90\%$ of the program.

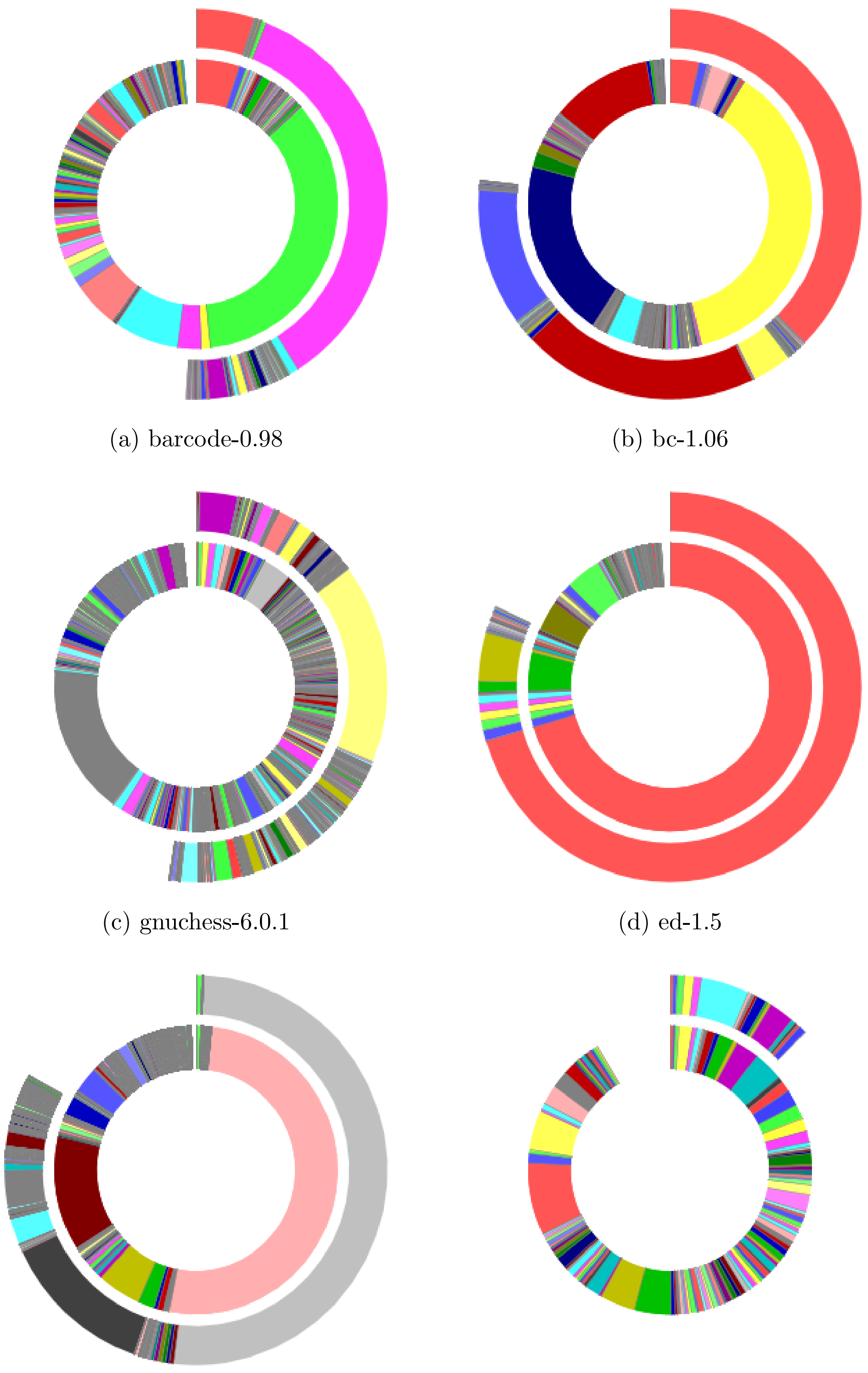
GNU Chess has a very higher number of small Type 1 dependence clusters but also contains 1 large dependence cluster (both Type 1 and Type 2) that covers $\approx 20\%$ of the program. Harman et al. found a dependence cluster that covered $\approx 80\%$ of the program. The smaller dependence cluster found could indicate that the code has undergone a refactoring, since the study by Harman et al., to reduce the *dependence pollution* in the code.

ed contains a very large dependence cluster (both Type 1 and Type 2) that covers approximately 70% of the program. Harman et al. found a similar result caused by code that operates on a common data structure [16]. Similarly, we found that *Empire* contains a large dependence cluster which corresponds to the result found by Harman et al.

time contains a small Type 2 dependence cluster and several small Type 1 dependence clusters (as well as many very small Type 1 dependence clusters). The largest of the small Type 1 dependence clusters is approximation 8% of the program size and bigger than the largest Type 2 dependence cluster. Harman et al. found a Type

[†]Barcode 0.98, bc 1.06 and time 1.7 were used in the previous study and this study, the version numbers for empire and GNU chess were not published, the version of ed used in the previous study was 1.2 whereas in this study it is 1.5

1 dependence cluster that covered approximately 20% of the program – it is not clear what the cause of the difference is; perhaps the use of different slicing criteria or CodeSurfer settings. Unlike the other 5 programs *time* contains very few Type 2 dependence clusters.



(a) barcode-0.98

(b) bc-1.06

(c) gnuchess-6.0.1

(d) ed-1.5

(e) empire-4.3.28

(f) time-1.7

Figure 5.4: Dependence clusters, as a percentage of program size (inner ring – Type 1, outer – Type 2)

5.2.4 Are Type 1 Dependence Clusters a Good Approximation to Type 2 Dependence Clusters?

Type 1 dependence clusters are an approximation to Type 2 dependence clusters introduced by Harman et al. [87] to reduce computational time; are they a good approximation to Type 2 dependence clusters?

Harman et al. [87] calculated Type 2 dependence clusters for only a subset of their test programs; here we calculate Type 1 and Type 2 dependence clusters for all 44 of our programs, by efficiently storing slices as compressed bitsets.

Our empirical study has found a very strong correlation between both the number and the size of Type 1 and Type 2 dependence clusters.

There is a very strong correlation (Pearson correlation coefficient 0.98, $p = 2.37 \times 10^{-29}$) between the number of Type 1 and the number of Type 2 dependence clusters in a program i.e. programs with more Type 1 clusters have more Type 2 clusters. There is also a very strong correlation (Pearson correlation coefficient 0.94 $p = 2.75 \times 10^{-21}$) between the number of procedures in a program and the number of dependence clusters which seems to suggest that the bigger the program (as bigger programs get bigger the number of procedures is likely to increase) the more dependence clusters there are. There is a very strong correlation between the size of Type 1 and Type 2 dependence clusters – programs with large Type 1 dependence clusters also have large Type 2 dependence clusters. The Pearson correlation coefficient between average size of Type 1 and Type 2 dependence clusters is 0.96 ($p = 2.14 \times 10^{-25}$).

Every program studied has a greater number of Type 1 than Type 2 dependence clusters. This is due to the conservative nature of the Type 1 approximation – the approximation may produce false positives. Large dependence clusters in software

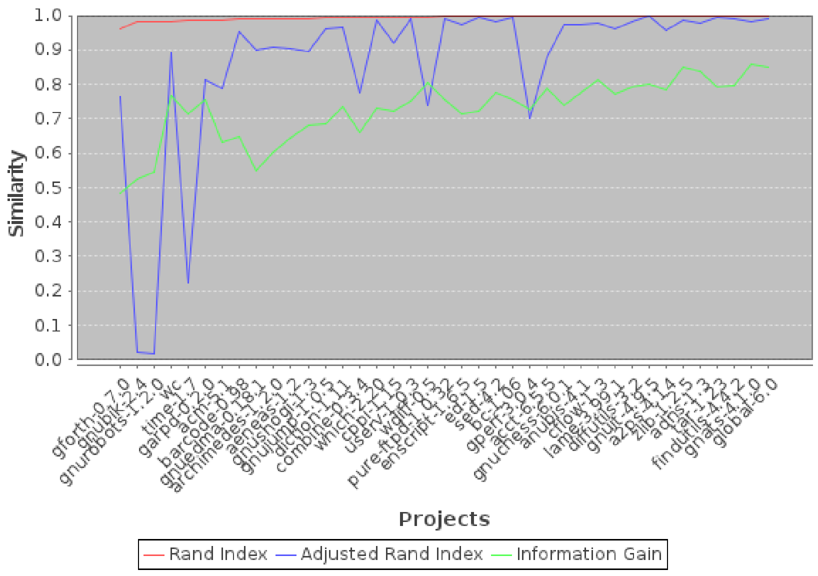


Figure 5.5: Type 1 and Type 2 dependence cluster BSG partition similarity

can be a cause of *dependence pollution* so it seems reasonable to be interested in the largest dependence cluster in a program.

We measured the Jaccard similarity between the largest Type 1 and Type 2 dependence clusters in each program and found that most had a similarity of 1, 3 had a similarity of 0.99 and 4 had a similarity of 0; therefore, in most programs the largest Type 1 dependence cluster is the same as the largest Type 2 dependence cluster.

In the 3 examples where the largest Type 1 dependence cluster does not exactly match the largest Type 2 dependence clusters the Type 1 dependence cluster has several nodes which are in only one of the other slices in the cluster; these are almost Type 2 dependence clusters.

A set of a program's Type 1 or Type 2 dependence clusters is a partition of the program's BSG[‡]; the similarity between these Type 1 and Type 2 dependence cluster partitions is shown in fig. 5.5. This chart displays the values of the Rand Index, Adjusted Rand Index and Information Gain Index – these are measures of partition similarity.

[‡]'true' dependence clusters do not partition the graph, as nodes may be in more than 1 maximal clique.

In general Type 1 dependence clusters are a good approximation to Type 2 dependence clusters. In several cases there is a low ARI value indicating a large discrepancy between Type 1 and Type 2 dependence clusters. In most cases, however, the value of the ARI is fairly high indicating a good similarity between the partition of the graph as given by Type 1 and Type 2 dependence clusters.

5.2.5 How Many Type 2 Dependence Clusters Are 'True' Dependence Clusters?

We have shown that the Type 1 dependence cluster approximation to Type 2 dependence clusters is valid, providing further evidence to support the findings of Harman et al. [87]. However, Type 2 dependence clusters are not 'true' dependence clusters – they are cliques but not, in general, maximal cliques. How many Type 2 dependence clusters are 'true' dependence clusters?

In order to answer this question we investigated whether or not each Type 2 dependence cluster, in our set of 44 programs, was a maximal clique (i.e. is the cluster contained within another cluster or not?). The results are shown in table 5.1.

In total, there were 23,018 Type 2 dependence clusters found in the set of 44 programs but only 5,245 of these are maximal cliques; in other words, over 75% of the Type 2 dependence clusters found are not 'true' dependence clusters.

In the worst case, the program *rsync-3.0.9* contained 1,225 dependence clusters but only 62 were found to be 'true' dependence clusters. Only the small *wc* program contained 100% Type 2 dependence clusters that were found to be 'true' dependence clusters.

On average programs contained only 36% of Type 2 dependence clusters that were also 'true' dependence clusters.

Program	**Dependence Cluster 2**		
	Number	**Cliques**	**M-Cliques**
a2ps-4.14	1013	1013	216
acct-6.5.5	143	143	29
acm-5.1	39	39	36
adns-1.3	475	475	37
aeneas-1.2	292	292	233
anubis-4.1	414	414	106
archimedes-1.2.0	303	303	195
barcode-0.98	109	109	59
bc-1.06	95	95	41
cflow-1.3	325	325	56
combine-0.3.4	159	159	35
cppi-1.15	84	84	35
diction-1.11	58	58	44
diffutils-3.2	535	535	214
ed-1.5	41	41	21
empire-4.3.28	1027	1027	138
enscript-1.6.5	398	398	113
findutils-4.4.2	581	581	134
garpd-0.2.0	19	19	15
gettext-0.18	3780	3780	667
gforth-0.7.0	101	101	34
global-6.0	1193	1193	155
gnats-4.1.0	672	672	103
gnubik-2.4	79	79	53
gnuchess-6.0.1	823	823	347
gnuedma-0.18.1	382	382	72
gnuit-4.9.5	449	449	85
gnujump-1.0.5	343	343	196
gnurobots-1.2.0	37	37	23
gnushogi-1.3	188	188	99
gperf-3.0.4	202	202	86
inetutils-1.8	2392	2392	416
lame-3.99.1	1156	1156	236
ntp-4.2.6p5-RC1	1698	1698	495
pure-ftpd-1.0.32	223	223	47
rsync-3.0.9	1225	1225	62
sed-4.2	246	246	77
tar-1.23	1281	1281	154
time-1.7	17	17	8
userv-1.0.3	150	150	40
wc	8	8	8
wdiff-0.5	35	35	9
which-2.20	34	34	3
zlib-1.2.5	194	194	13

Table 5.1: The number of clusters, cliques and maximal cliques.

5.3 Conclusion

In this chapter we have conducted an investigation into the approximations to dependence clusters published in previous work. We have found, like previous studies, that many programs have large dependence clusters; this result provides further evidence of the *dependence pollution* created by dependence clusters in programs.

We have confirmed results of previous studies which used the Type 1 dependence cluster approximation to Type 2 dependence clusters and provided further evidence that Type 1 dependence clusters are a good approximation to Type 2 dependence clusters.

We have shown that not all Type 2 dependence clusters are ‘true’ dependence clusters; a ‘true’ dependence cluster is a maximal clique in a BSG and we think of dependence clusters as a stricter form of dependence community in a BSG. We found that over 75% of Type 2 dependence clusters in the set of 44 programs were not ‘true’ dependence clusters, i.e. they were not maximal cliques. This result is important for further understanding dependence clusters and how they are formed by dependence between statements in programs.

We consider dependence clusters overly strict and a weaker interpretation of the clustering of statements in source-code due to dependence, such as dependence communities, would be more appropriate. The study of ‘clusters’ of dependence in software has a huge number of potential applications including program comprehension, maintenance, debugging, refactoring, testing and software protection.

CHAPTER 6

A Study Of Software Quality

What makes good software?

We have introduced the concept of dependence communities, introduced a new form of slice-based cohesion and coupling metrics, and investigated dependence clusters; these techniques can all be used to quantify the quality of software.

In this chapter, we discuss the inter-connected nature of the three main topics of this thesis. Are they all different measures of the same quality? Do programs with high cohesion have low coupling? Do programs with large dependence clusters also have large dependence communities?

We show that programs programs with higher coupling have larger dependence communities, programs with large dependence clusters also have large dependence communities and programs with high modularity have low coupling.

6.1 Introduction

Software metrics can be used to quantify the quality of software; for example, programmers should strive to obtain high cohesion, which measures the amount dependence between statements within a procedure, and low coupling between procedures, which measures the amount of dependence between procedures.

Software with high cohesion and low coupling would be built of procedures that perform singular tasks in near isolation from other parts of the program; such modular programs are easier to understand, maintain and allow for code re-use [192].

Another measure of the quality of a program is the presence or lack of large dependence clusters [21]; large dependence clusters - sets of program statements that are mutually dependent - can give rise to dependence pollution [16]. The presence of a large dependence cluster can make a program harder to maintain as changing any statement in a dependence cluster may affect *all* of the other statements in that cluster.

A possible measure of software quality is the community structure of a program [78]; although further work is required in this area, the results of the empirical study in chapter 3 suggest that dependence communities reflect the semantic concerns of a program. A measure of the strength of the community structure of a program is the modularity of its Backward Slice Graph (BSG). The results of the empirical study showed that all 44 programs studied had a positive modularity, indicating community structure, although some programs had a higher modularity than others – a promising result as a measure of program quality, but clearly more work is needed in this area; for example, the *Louvain method* may not be the best algorithm to use for detecting communities in BSGs and a comparison of other community detection algorithms is required.

So good programs have high cohesion, low coupling, small dependence clusters and communities that reflect the semantic concerns of a program. Is this the case? Are these different methods of measuring the same thing?

Intuitively, from the definition of dependence clusters and dependence communities we know that there is a connection to cohesion and coupling. Areas of high cohesion in a BSG give rise to both dependence clusters and dependence communities, though the former requires higher cohesion than the latter as a single edge is enough to bring a node into a dependence community but an edge between a single node and every other node in the dependence cluster is required.

In a program with very low coupling we would not expect large dependence clusters or dependence communities. In fact, a program with 0 coupling could not contain a cluster or community larger than the largest procedure (as there would be no edges in the BSG connecting nodes in different procedures).

A program with low coupling may be more likely to have larger dependence communities than dependence clusters as dependence communities rely on a weaker relation between nodes in a BSG; for example, a single edge between procedures could bring statements from both procedures into the same community.

A program with very low cohesion would also be unlikely to have large dependence clusters or dependence communities as, again, this could indicate few edges between statements.

Are large dependence communities as bad as large dependence clusters? Dependence clusters are a stricter form of dependence community and therefore we would expect a program with large dependence clusters to have large dependence communities.

At first thought, the community structure of a program should correspond to the procedures of that program – a good program should have procedures with high cohesion and low coupling between those procedures; in a sense, this would be the perfect community structure for a program to have.

In most programs, areas of functionality are not confined to single procedures; rather, a functional aspect of the program is broken down into smaller procedures. If dependence communities are to reflect the semantic concerns of a program, the perfect community structure for a program is therefore not the same as the procedures. We would expect to see large communities covering a number of procedures that reflect the semantic areas of the program that have been split across procedures.

However, there is a limit to the usefulness of the size of a dependence community; several of the programs in the empirical study in chapter 3 contained a dependence community that covered most of the program. A dependence community this large would not be of any practical use in uncovering the different semantic concerns of the program and would likely be the result of very high cohesion and/or coupling; such a community would likely not reflect the semantic concerns of the program. A result such as this may be useful for determining whether a program is well written; a good program should have a community structure which reflects it's semantic concerns.

6.2 Research Questions

Is there a relationship between dependence communities and cohesion & coupling?

Dependence between statements in software gives rise to dependence communities (see chapter 3). Cohesion & coupling are measures of the 'strength' of dependence within and between groups of statements; intuitively, this seems closely related to the definition of dependence communities (groups of highly dependent statements that are weakly connected to other groups). Our hypothesis is that there may be a relationship between cohesion & coupling and the number / size of dependence communities. Do procedures that are members of multiple communities have a lower cohesion? Do programs with higher coupling have larger communities?

We show that procedures which are part of multiple communities have a lower cohesion than those with one community and that programs with higher coupling have larger dependence communities.

Is there a relationship between dependence communities and dependence clusters?

Binkley and Harman [16] found that large dependence are prevalent in software and are a major cause of 'dependence pollution'; we provided further evidence to support this claim in chapter 5. We also provided evidence of the community structure in BSGs in chapter 3. Is there a relationship between the two? Dependence clusters are a stricter form of community in a BSG. We hypothesis that programs with large dependence clusters will likely have large dependence communities due to the large amount of dependence in the program.

We show that there is a correlation between dependence clusters and dependence communities; in fact, 99.94% of Type 2 dependence clusters are fully contained within a single dependence community. We also show that dependence communities give a stronger modularity than dependence clusters i.e. the *Louvain method* partitions the BSG into areas of 'tighter' dependence.

6.3 Dependence Communities and Cohesion

Given that a dependence community is a collection of dependent program statements it seems, intuitively, as though a procedure with multiple communities may contain distinct groups of semantically related statements. Do procedures that are members of multiple communities have a low cohesion?

The average number of communities in a procedure is 1.15; a procedure that is completely in one community only would suggest a high cohesion. If different parts of a procedure are in different communities it would suggest that there are subsections of code within the procedure which are strongly dependent internally but weakly dependent on each other; this would suggest a low cohesion.

The average number of procedures in a community is 17.07. A large proportion of communities have only 1 procedure in them but a smaller number have a very large number of communities in them. In each program some of the procedures are therefore in their own community but large communities cut across several procedures.

Figure 6.1 shows the weighted average cohesion values for procedures with only 1 and multiple communities.

The results suggest that counting the number of communities that appear in a procedure will give an indication of the cohesion of a procedure. Procedures that are part of multiple communities tend to have a lower cohesion value than those that are only in 1 community. This result matches the intuitive idea of a dependence community – an area of code that is more dependent on itself that the rest of the program. Having multiple communities in a procedure suggests that there are multiple areas of code that are not highly dependent on each other; this is therefore a measurement of cohesion.

For example, in the *main* procedure of the program *time-1.7* the tightness is low at

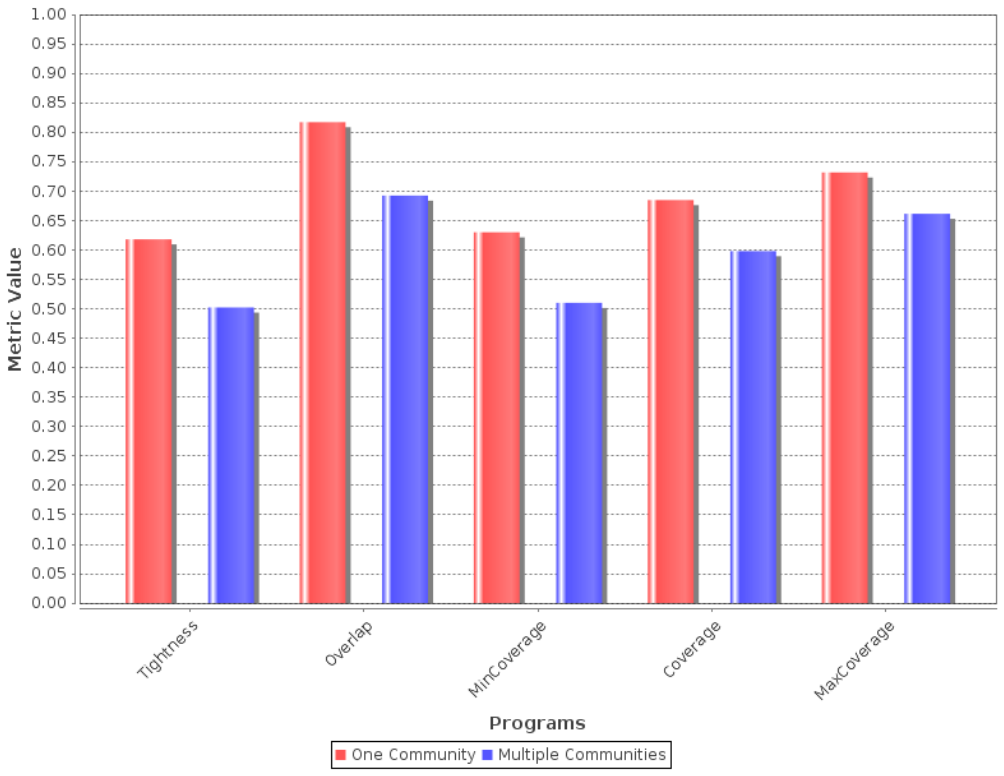


Figure 6.1: Weighted average cohesion values for procedures with one and multiple communities

0.36 and the number of procedures is greater than 1; the average community size is 25% of the procedure.

Although the results are encouraging, it is not always the case that having a single community means high tightness, for example in the procedure *usage* from the program *time-1.7*. The tightness for this procedure is 0.46 due to the multiple `actual-ins` being crucial points. There is, however, only 1 community which encompasses this whole procedure.

6.4 Dependence Communities and Coupling

We have shown that procedures with only 1 community have higher cohesion than those with multiple communities. How does coupling relate to dependence communities? Recall that a dependence community is a set of statements that have a high dependence on each other and low dependence on statements outside the community. It seems that a program with high coupling would, intuitively, result a program with larger dependence communities. High coupling between 2 or more procedures will mean that it is more likely that those procedures are in the same community.

A program containing two procedures with a coupling of 0 would mean that the procedures could not be in the same dependence community. However, unlike dependence clusters, a single edge between two procedures could, in theory, cause the two procedures to be in the same dependence community.

Do programs with high coupling have larger dependence communities than those will low coupling?

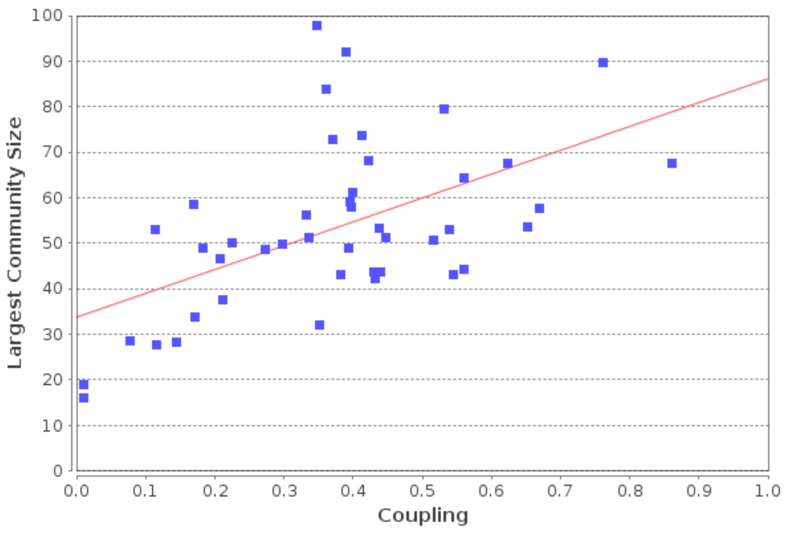


Figure 6.2: Coupling vs largest community size ($R = 0.54$, $p < 0.0001$)

Figure 6.2 plots weighted average coupling of a program against the size of the largest dependence cluster as a percentage of the program. There is a moderate correlation between the weighted average of coupling of a program and the size of the largest dependence community; the Pearson correlation coefficient is 0.54 ($p < 0.0001$).

This result means that as coupling increases the size of the largest dependence community increases i.e. programs with higher coupling have larger dependence communities. The result seems intuitive in the sense that procedures with greater dependence on each other will be more likely to be in the same dependence community. If the average coupling of a program is high it means that many procedures have a high dependence on other procedures; in turn, this means that dependence communities will be larger.

Programs with very low coupling tend to have small dependence communities, while those with high coupling will tend to have larger dependence communities. It is interesting to note that this is not always the case and there are a few outliers in the fig. 6.2 which have moderate coupling and very large dependence communities.

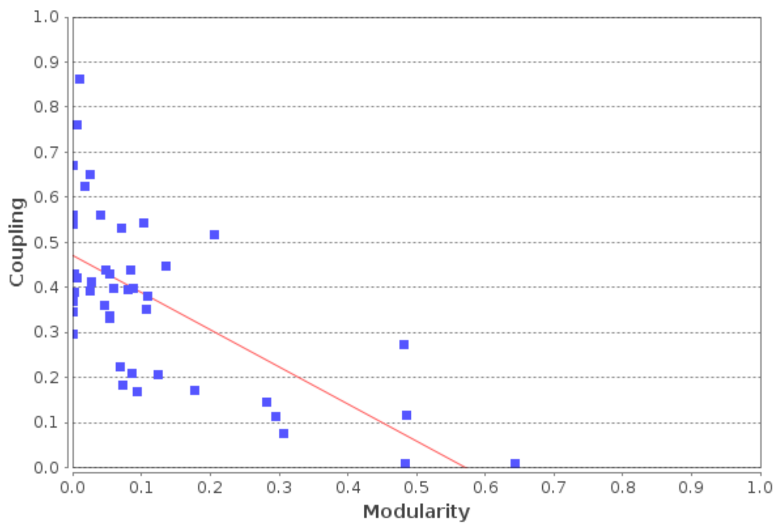


Figure 6.3: Modularity vs coupling ($R = -0.66$, $p < 0.000001$)

Modularity is a measure of the 'strength' of community structure in a graph. A high modularity indicates a better community structure. A high coupling in a program would indicate that many procedures are highly dependent on each other which intuitively would reduce the number of well-defined dependence communities. Do programs with high modularity have a low coupling?

Figure 6.3 plots the BSG modularity, when partitioned using the *Louvain method*, against weighted average coupling for each of the 44 programs. There is a strong negative correlation between the modularity of a program's BSG and the average coupling; the Pearson correlation coefficient is -0.66 ($p < 0.000001$).

This result confirms the hypothesis that programs with a high modularity have a low coupling, suggesting that modularity could be used as a metric in the field of software maintenance. Programs with a high modularity will not have a high coupling, therefore a well-written program should have high modularity.

6.5 Dependence Communities and Dependence Clusters

A ‘true’ dependence cluster is a maximal clique within a Backward Slice Graph in which every node is connected to every other node; thus a dependence cluster can be thought of as a stricter form of dependence community in a BSG.

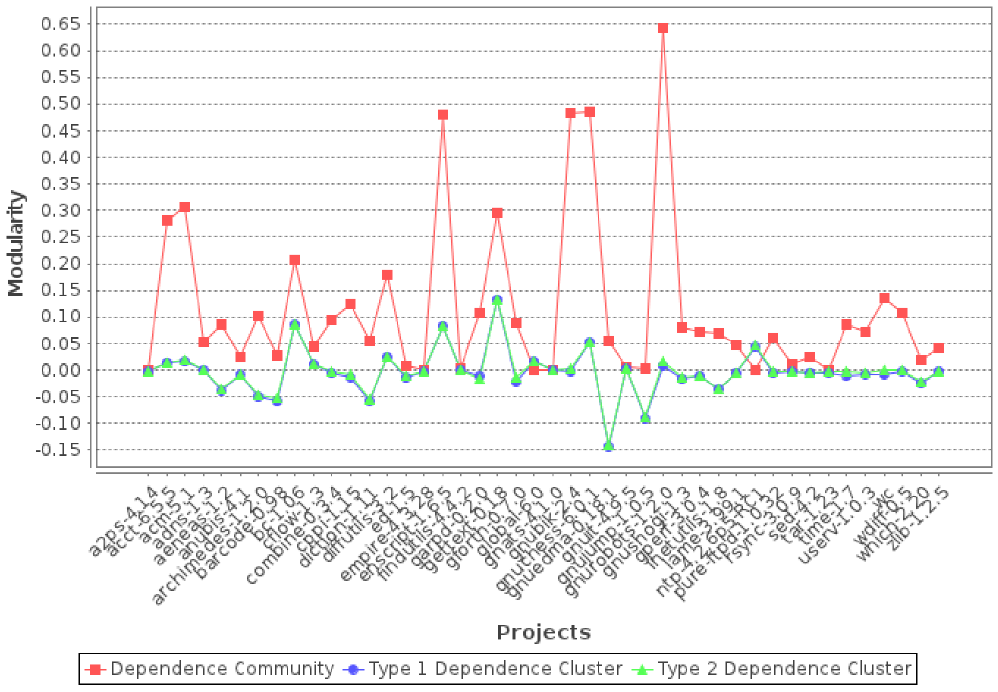


Figure 6.4: Modularity of the 44 BSGs using 3 partitions

Figure 6.4 shows the modularity for the 44 BSGs when partitioned using dependence communities and dependence clusters. It turns out that if we apply the *Louvain method* to the same graphs we get a partition with higher modularity. In other words it produces ‘clusters’ with a stronger ‘internal inter-dependence’ than those produced by Type 1 and Type 2 dependence clusters. There is very little difference between the modularity when partitioning the graph with Type 1 and Type 2 dependence clusters (perhaps further evidence that Type 1 clusters are a good approximation to Type 2 clusters).

It could be argued, therefore, that dependence communities may be a better approximation to dependence clusters or at least a better approximation to the properties of programs that the authors are trying to capture using dependence clusters.

The rings in figs. 6.5 to 6.7 depict the Type 1 and Type 2 dependence clusters, and the dependence communities for 3 programs discussed in the chapter 3. Each section corresponds to a dependence cluster or dependence community with its size as a percentage of the total program size (the inner ring depicts Type 1 dependence clusters, middle ring Type 2 and outer ring dependence communities).

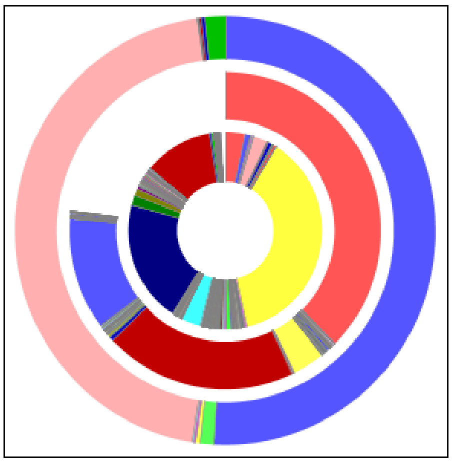


Figure 6.5: GNU bc dependence clusters and communities as percentages of program size (inner ring - Type 1, middle ring - Type 2, outer ring - communities)

GNU bc has similar dependence clusters and dependence communities, except that the dependence communities are bigger – there are two large Type 1 and Type 2 dependence clusters and two large dependence communities.

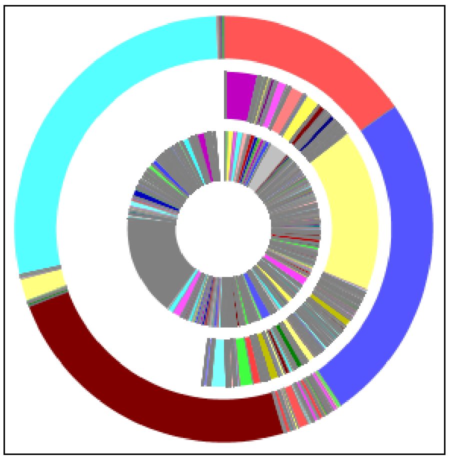


Figure 6.6: GNU Chess dependence cluster and community BSG as percentages of program size (inner ring - Type 1, middle ring - Type 2, outer ring - communities)

However, with *GNU Chess* there is only one 'large' dependence cluster and many small dependence clusters; this is clearly a different result from dependence communities.

In the *GNU Robots* program there are no large dependence clusters, most likely due to the low coupling between procedures; there are however, moderately sized dependence communities.

One reason for this difference results from the fact that dependence clusters must be cliques whereas dependence communities are not as strict – this means that there are more chances for the semantically related statements in a program to be grouped together.

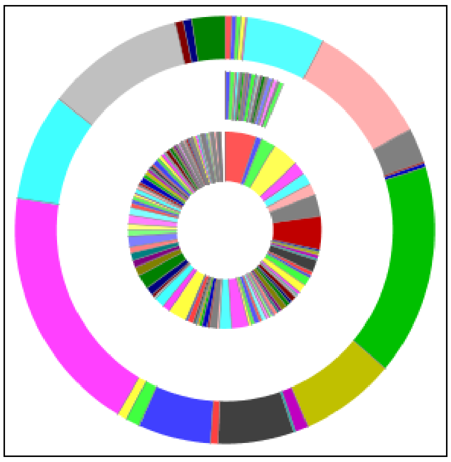


Figure 6.7: GNU Robots dependence clusters and communities as percentages of program size (inner ring - Type 1, middle ring - Type 2, outer ring - communities)

In *GNU Chess*, and to a much greater extent *GNU Robots*, a large proportion of each program is 'ignored' when using Type 2 dependence clusters; i.e. there are many System Dependence Graph (SDG) nodes which produce unique slices when used as slicing criterion and the focus is on the analysis of large dependence clusters in a program.

When the BSG is partitioned into dependence communities, however, these nodes are included in a community. These nodes do contribute to the program and therefore should be 'fairly represented' in any form of dependence-based 'clustering'.

Figure 6.8 plots the size of the largest dependence community (as a percentage of program size) against the size of the largest dependence cluster (as a percentage of program size). There is a correlation between the size of the largest dependence community in a program and the size of the largest dependence cluster; the Pearson correlation between these is 0.51 with a p-value < 0.0001.

It turns out that dependence communities tend to be larger than dependence clusters. A dependence cluster is a stricter form of dependence community and a program with large dependence clusters will have large dependence communities. In

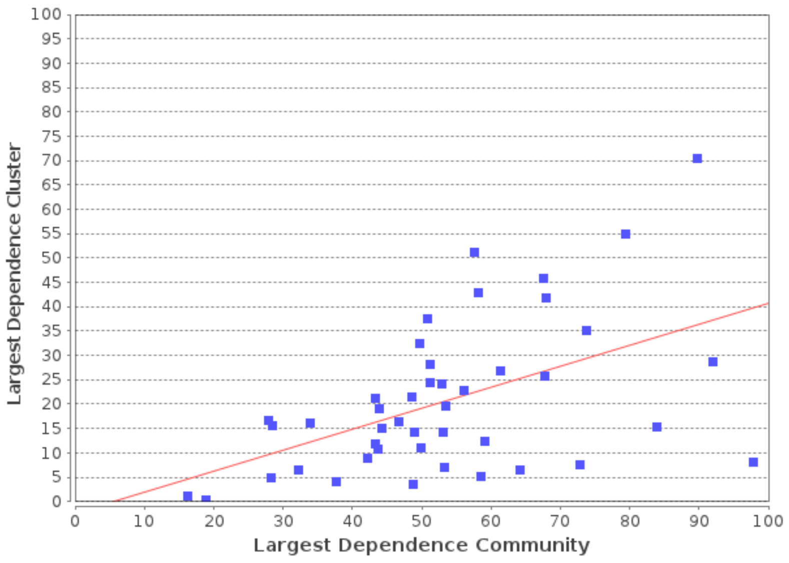


Figure 6.8: Largest dependence community vs largest Type 2 dependence cluster

fact, 99.94% of Type 2 dependence clusters are fully contained within a single dependence community and there are, on average, 16.25 dependence clusters in a dependence community. Perhaps this is not surprising when considering that a dependence cluster is a fully connected sub-graph – the 'strongest' form of community possible. It is therefore unlikely that splitting a dependence cluster will increase modularity.

The presence of extremely large dependence clusters may hinder the usefulness of community detection algorithms on BSGs; it may be a necessary to find the cause of a large dependence cluster and eliminate any possible dependence pollution before dependence communities are able to accurately reflect the semantic concerns of a program. Of course, the presence of an extremely large dependence community is therefore also an indication of dependence pollution.

6.6 Conclusion

In this thesis we have discussed several methods of measuring software quality and bought them together in this chapter:

high cohesion

procedures should be highly cohesive, allowing for code re-use and ease of maintenance.

low coupling

procedures should have low coupling to other procedures, indicating a highly modular design.

small dependence clusters

programs should not have large dependence clusters which result in dependence pollution.

semantic dependence communities

programs should have communities which reflect the semantic concerns of the program.

We have shown that procedures which are part of multiple communities have a lower cohesion than those with one community; this result is intuitive because if there are multiple communities within a procedure that procedure may be performing more than one task. We would also expect a procedure performing more than one task to have a low cohesion, as there would be statements which do not depend on one another.

We have shown that programs with higher coupling have larger dependence communities. The link between coupling and larger dependence communities is intuitive in the sense that the more connections there are between procedures, the more chances there are of nodes in those procedures being in the same community.

We have shown that dependence communities give a stronger modularity than dependence clusters i.e. the *Louvain method* partitions the BSG into areas of 'tighter' dependence. We have shown that there is a correlation between dependence clusters and dependence communities; in fact, 99.94% of Type 2 dependence clusters are fully contained within a single dependence community.

We have also shown that programs with high modularity - a measure of the quality of a partition of a network - have low coupling. This result suggests that modularity is a promising measure of software quality; having a higher modularity indicates a better community structure and a lower coupling.

We have discussed the connections between the different areas focused on in this thesis. Clearly, there are links between communities, clusters, and cohesion and coupling as they all come about due to dependence between statements in software (and therefore also nodes in a BSG).

Future work requires a measure of software quality outside of the techniques investigated in this thesis. Such comparisons would likely involve comparing the quantitative measures of software quality, used in this thesis, against information such as bug reports, or software revision history.

CHAPTER 7

Conclusion

In this thesis we have introduced the concept of *dependence communities*, investigated the closely related *dependence clusters*, and developed an efficient technique for calculating slice-based cohesion and coupling.

We have used the notion of a Backward Slice Graph (BSG) as the basis for the empirical studies in this thesis, which is a graph of a program's backward slices such that a is connected to b if $b \in Slice(a)$. A graph-based representation of software allows us to apply existing graph-theoretic techniques commonly used in other fields, such as community detection.

The first major contribution of this thesis is the concept of a *dependence community*; this work is the first to investigate community structure at the statement-level in software.

We applied a well known community detection algorithm to the BSGs of 44 open-source programs, analysing a total of 464,621 lines of code. We have shown that BSGs have community structure and in some examples we have observed that the communities approximate to the semantic concerns of the program.

The *Louvain* algorithm has previously been successfully used on large graphs of up to 118 million nodes and 1 billion edges. We have successfully applied the algorithm to graphs with up to 305,000 nodes and 16 billion edges.

The modularity for the set of 44 programs in the empirical study varies, with some programs showing a stronger community structure than others. It is not yet clear what 'stronger community structure' means in terms of software dependence but the positive findings merit further work. We suspect that there is a connection between high modularity and how well a program is separated into its different semantic concerns. It is likely that programs with extremely large dependence clusters will not have a good community structure and the presence of an extremely large dependence cluster will be mirrored by the presence of an equally large dependence community. The presence of an extremely large dependence community is further evidence of dependence pollution in a program.

We manually inspected a number of programs to answer the question "what do dependence communities in software mean?" The analysis revealed that dependence communities seem to reflect the functional behaviour or semantics of the program; for example in the GNU *wc* program, we found two dependence communities: the *counting* community, which consists of the parts of the program which count the number of lines, characters and words in a file, and the *I/O* community which contains the code which opens the file, prints error messages and prints results.

Being able to break down software into meaningful sub-components is one of the major challenges in software engineering. Previous research in the area has focused on the clustering of high-level components in a software system to recover a modular structure or re-modularise software. The Bunch tool [129, 133], for example, works on a Module Dependency Graph (MDG) which includes high-level system components such as Java classes or C files that are connected due to dependence.

There is little point in breaking a piece of software into smaller components if these components do not in some way reflect different *functionalities*. We, therefore, believe that good semantic separation is the key to the usefulness of partitioning techniques in software engineering.

Of course no automated approach can perform perfect semantic separation. Our hypothesis is that dependence communities computed using community detection algorithms applied to program graphs can give *sufficiently good* semantic separation to be highly applicable in a number of areas of software engineering. This new approach has applications in all areas of software engineering where system decomposition is important, including software comprehension, maintenance, reverse engineering, restructuring, software evolution and information recovery.

Our results provide compelling evidence that there is great potential for using dependence communities as an alternative software clustering methodology. Our results have shown that there is merit in further investigations of dependence communities and their application to various software engineering areas.

The second major contribution of this thesis is a new, efficient form of slice-based metrics based on maximal slices. Previously, slice-based software metrics have been defined using the ambiguous concept of output variables; it is difficult to define output variables, different studies have used different definitions of output variable and slice-based metrics are undefined for procedures that do not contain output variables.

Our new forms of slice-based metrics using maximal slices are defined for *all* procedures in a system and are strongly-correlated with the previous output variable based metrics.

We undertook an empirical study where we calculated the cohesion and coupling values for 20,588 procedures in 44 programs using both the old output variable based definitions and our new maximal-slice-based definitions.

In the empirical study we found that about 5% of the 20,588 procedures considered did not have output variables at all, and so the slice-based metrics for these procedures are not defined.

Our results showed that there is a strong correlation between output variable based metrics and the new maximal-slice-based metrics, for procedures which contain output variables. This is strong evidence that the new maximal-slice-based metrics are a valid generalisation to all programs of the previous more restrictive sliced-based metrics.

The use of maximal slices is appropriate because a maximal slice captures an 'interesting' part of a program; any code that occurs in a maximal slice cannot affect code outside the maximal slice. Intersecting maximal slices represent inter-related program statements, while disjoint maximal slices suggest multiple tasks are being performed.

A limitation to the use of maximal-slice-based metrics is the greater number of slices required to be computed for the computation of the metrics. Unlike output variable based metrics, slices must be computed for every slicing criterion, in order to be able to calculate the maximal slices. The total number of slices computed for the set of 44 programs was 1,725,800 and the number of output variables in the set of programs was between 670,953 and 743,758 (depending on which output variable definition is used). Therefore, approximately 1 million more slices had to be computed for the calculation of maximal-slice-based metrics, compared with output variable based metrics.

We took advantage of the near transitivity of data and control dependence [47] to develop a much more efficient approach to computing extremely accurate approximations to maximal-slice-based metrics. These were computed very quickly and led to metrics which were almost identical to their maximal-slice-based counterparts. Additionally, these *pseudo*-maximal-slice-based metrics required fewer slices to be computed than previous output-variable-based metrics.

The third major contribution of this thesis is an investigation into *dependence clusters*; these are closely related to our new concept of *dependence communities*.

A dependence cluster is a maximal set of program statements all of which are mutually dependent. The presence of large dependence clusters in a program could hinder software maintenance as changing any statement in a cluster could potentially impact *all* other statements in that cluster.

We conducted an empirical study which confirmed the results of previous studies. Our results showed that programs contain large dependence clusters which provides further evidence of the 'dependence pollution' created by dependence clusters in programs.

However, we showed that over 75% of dependence clusters, as calculated by previous studies, are not 'true' dependence clusters. Continuing our graph-based approach in this thesis, we redefined 'true' dependence clusters as maximal cliques within a BSG, and think of dependence clusters as stricter forms of dependence communities.

We consider dependence clusters overly strict and a weaker interpretation of the clustering of nodes due to dependence would be more appropriate. Previous authors have also suggested that an approximation to dependence clusters is, in fact, more useful than 'true' dependence clusters due to the weaker definition [87].

The major contributions of this thesis are all based on the analysis of dependence between statements in software; they also all provide a quantitative measure of software quality:

high cohesion

procedures should be highly cohesive, allowing for code re-use and ease of maintenance.

low coupling

procedures should have low coupling to other procedures, indicating a highly modular design.

small dependence clusters

programs should not have large dependence clusters which result in dependence pollution.

semantic dependence communities

programs should have communities which reflect the semantic concerns of the program.

We have shown that procedures which are part of multiple communities have a lower cohesion than those with one community; this result is intuitive because if there are multiple communities within a procedure that procedure may be performing more than one task. We would also expect a procedure performing more than one task to have a low cohesion, as there would be statements which do not depend on one another.

We have shown that programs with higher coupling have larger dependence communities. The link between coupling and larger communities is intuitive in the sense that the more connections there are between procedures, the more chances there are of nodes in those procedures being in the same community or cluster.

We have shown that dependence communities give a stronger modularity than dependence clusters i.e. the *Louvain* algorithm partitions the BSG into areas of 'tighter' dependence. We have shown that there is a correlation between dependence clusters and dependence communities; in fact, 99.94% of Type 2 dependence clusters are fully contained within a single dependence community.

We have also shown that programs with high modularity - a measure of the strength of a partition of a network - have low coupling. This result suggests that modularity is a promising measure of software quality; having a higher modularity indicates a better community structure and a lower coupling.

Measures of software quality do not always agree and further work requires a measure of software quality outside of the techniques investigated in this thesis. Such comparisons would likely involve comparing the quantitative measures of software quality, used in this thesis, against information such as bug reports, or software revision history.

CHAPTER 8

Future Work

8.1 Dependence Communities

The results of the study using the *Louvain* algorithm show great potential in its use for the detection of dependence communities but future work into the multitude of other community detection algorithms [119] is merited, to investigate both efficiency and accuracy for the purpose of finding dependence communities in software.

Further work will investigate the impact of community detection algorithms in software engineering by measuring the resulting semantic separation and comparing it with other approaches. Importantly, we need to understand why some approaches to semantic separation are better than others so that the techniques can be improved.

Measuring semantic separation will be done by employing techniques similar to watermark injection. 'Intertwining' a number of pieces of code with separate functionality and measuring how well the pieces are separated again by the community detection algorithm. This will give us the ability to measure semantic separation (and hence usefulness) of other clustering and community detection algorithms.

Further work can be done to optimise the implementation of algorithms for the detection of dependence communities. For example, GraphChi [1, 113] is a recently released framework for performing computations on very large graphs by using a novel parallel sliding window technique to efficiently cache parts of the graph to disk; implementing community detection algorithms using this framework would allow the detection of dependence communities in large software using modest hardware.

The run-time costs of the slicing required to produce the Backward Slice Graphs (BSGs) is another issue to be considered in future implementations. Binkley et al. [19] studied techniques for such “massive slicing” of System Dependence Graphs (SDGs) where slices are required for all possible slicing criteria, as in the case of BSGs. Further work will need to consider optimisations to the slicing implementation, based on techniques used for “massive slicing” in order to more effectively analyse large code-bases.

The combination of “massive slicing” optimisations and graph algorithm optimisations represent two major improvements, which require further study, to an implementation of a large-scale dependence community analysis tool.

An initial investigation into the dependence communities found in watermarked programs provides promising evidence that dependence communities do partially, at least, lead to the separation of the functionally independent watermark code; further investigation is merited, as the ability to detect software watermarks will reveal which watermarking techniques are and which are not suited for the protection of intellectual property in software.

There is a large amount of research into community detection in various fields such as physics and biology which could also be applicable to software. Any novel algorithms developed for detecting dependence communities in software may also be applied in these areas; providing opportunities for inter-disciplinary work with various groups.

8.2 Dependence and the Topology of Slice Graphs

Preliminary results suggest that BSGs, unlike other software networks, are not scale-free due to the presence of dependence clusters. Figure 8.1 shows a visual comparison between the scale-free SDG and the BSG of the *wc* program. In a scale-free network there is no typical out degree and a common property of scale-free networks is the presence of *hubs* [10] – these are important, influential nodes in the network.

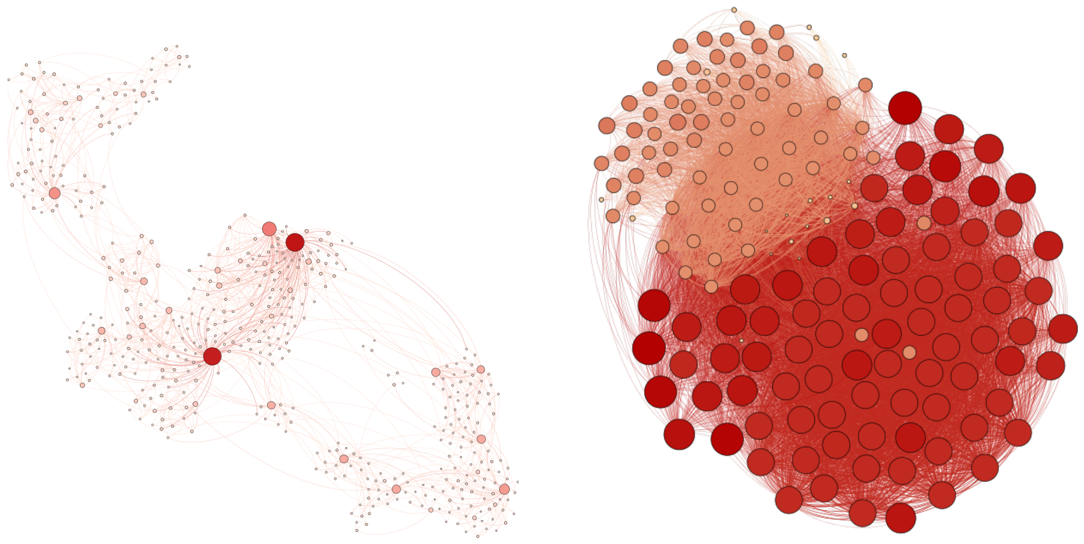
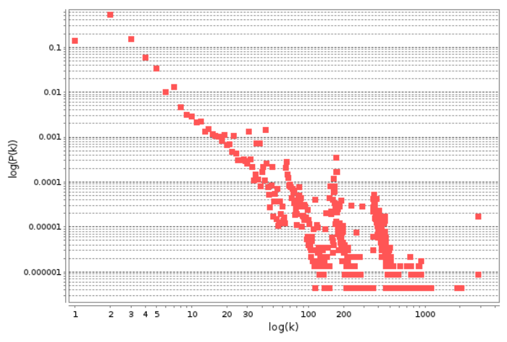

(a) SDG (b) BSG

Figure 8.1: The *wc* program as two complex networks

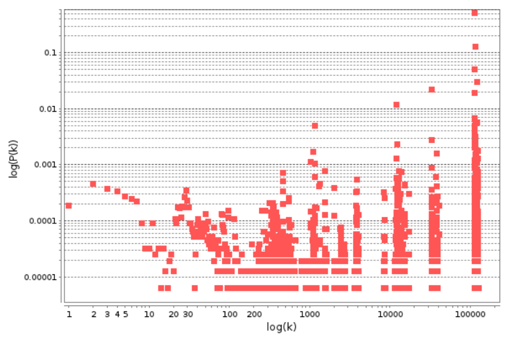
The SDG clearly shows the presence of *hubs* – there are a few nodes that have a very high out degree (larger/dark in the figure). In comparison the BSG contains large groups of nodes with the same out degree and there are no clear hubs in the graph.

Recall that a Type 2 dependence cluster is a set of SDG nodes that have the same slice. In a BSG a set of nodes that have the same slice translates to a set of nodes that have the same out degree. In other words, programs with large dependence clusters cannot be scale-free.

Figure 8.2 plots the degree distribution of the *empire-4.3.28* on a log-log plot. In a scale-free network we would expect to see values around a 45 degree line as in the SDG plot (fig. 8.2a). The BSGs do start like this but there are a large number of slices of similar sizes which disrupt the scale-free geometry of the graph (fig. 8.2b).



(a) SDG



(b) BSG

Figure 8.2: Degree distributions in *empire-4.3.28* (log-log scale)

Further work will involve verifying this result and conducting an empirical study to investigate the effects that dependence clusters, and dependence in general, have on the topology of BSGs.

8.2.1 The Topology of Watermarked Graphs

An application of the further work into topology of slice graphs is the detection of watermarks embedded in software. It is important to know if an embedded watermark is vulnerable to detection – if it can be detected by the embedder then we must assume that the attacker can also detect the watermark. An attack vector such as this is important to study for the development of secure and protected software. Figure 8.3 shows the *slice subset ordering graph*[*] for a simple Tic Tac Toe program, without and with watermark. It is clear that the embedded watermark has had a large effect on the topology of the graph. Further work will investigate various watermarking algorithms and their effect on the topology of a variety of program

[*] a is connected to b iff $slice(a) \subseteq slice(b)$ such that there is no r where $slice(r) \subseteq slice(b)$

graphs. The results could be used to introduce novel watermarking algorithms which remain hidden from attackers.

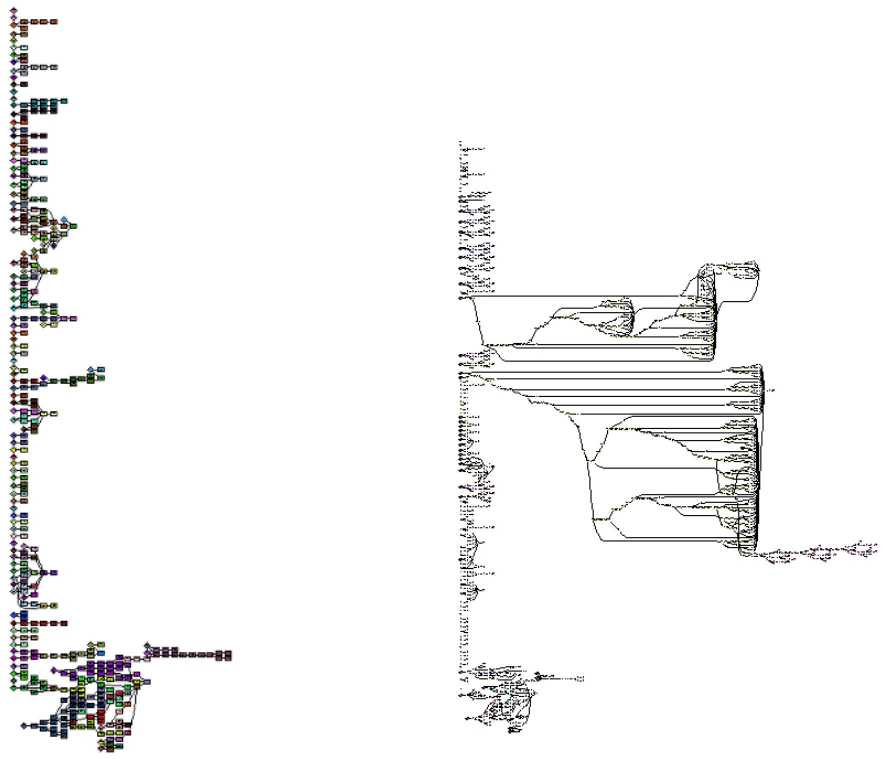

(a) before watermarking (b) after watermarking

Figure 8.3: Tic-Tac-Toe Program

8.2.2 Developing Models of Software

Barabási and Albert [9] developed a preferential attachment graph model to model the expansion of scale-free networks. Further work will consider whether there exists a graph model which closely approximates the evolution of a software system over time. For example, how does the topology of a slice graph change over time? Do new nodes attach preferentially to already well connected sites? Wang et al. [181] previously showed that the call graph of the Linux kernel evolves following a preferential attachment model.

8.2.3 Visualising Software Dependence With Slice Graphs

A fruitful avenue of research that we are now investigating is representing a program module as a graph of slices. The visualisation of these graphs may give a better intuitive insight into the cohesive properties of programs rather than a simple value.

These visualisations might also be useful for locating software watermarks [38, 75–77] or malware [36, 166] within programs - allowing us to define better software watermarks, or easily remove malware.

8.3 Software Metrics

We intend to investigate the relationship between standard graph metrics such as density, centrality, clustering coefficient, network diameter and the dependence in software graphs. Modelling dependence as a graph affords us the opportunity to apply such metrics which may have significance in the context of software dependence. Graph metrics have already been successfully applied to object-oriented programs [43, 79, 96, 97, 126, 190].

8.4 A Large-scale Study of Open-Source and Commercial Software

We have conducted empirical studies using 44 open-source programs and although these represent a variety of software applications a further study should consider a larger sample. A large-scale study of both software metrics and dependence communites using both open-source and commercial software will be undertaken.

Future work will, like previous studies, use maximal-slice-based metrics to study the evolution of metric values over different versions of software [131]; previous studies have applied this technique, for example, to fault analysis [23].

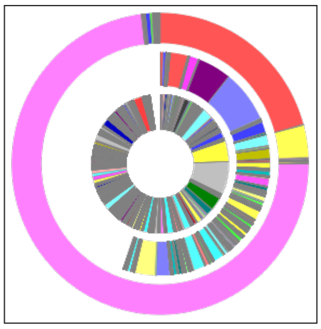
APPENDIX A

Partition Charts

Chapter A is the full set of 44 partition charts described in chapters 3 and 5. The inner ring shows Type 1 dependence clusters, the middle ring shows Type 2 dependence clusters and the outer ring shows dependence communities; each section of the ring depicts the size of the community/cluster as a percentage of program size.

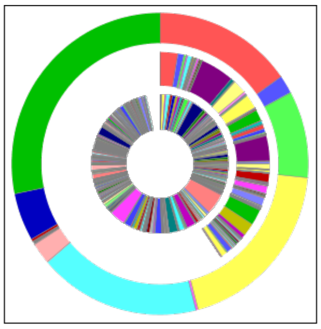
Cohesion and coupling metrics, as calculated using maximal slices, are shown next to the charts (see chapter 4, page 86); these are the weighted average and average values for the program.

The modularity and the size of the largest community (see chapter 3, page 67) and Type 1 / 2 dependence cluster (see chapter 5, page 114) is also shown.



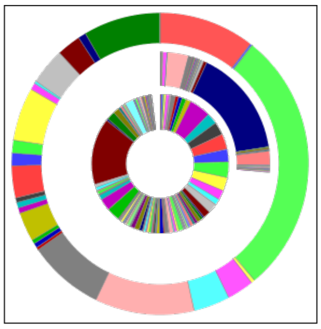
Tightness: 0.52, 0.55
Overlap: 0.78, 0.74
MinCoverage: 0.54, 0.6
Coverage: 0.62, 0.68
MaxCoverage: 0.68, 0.76
Coupling: 0.37, 0.32
Modularity: 0.00101
Largest DC1: 7.53%
Largest DC2: 7.53%
Largest Community: 72.89%

(1) a2ps-4.14



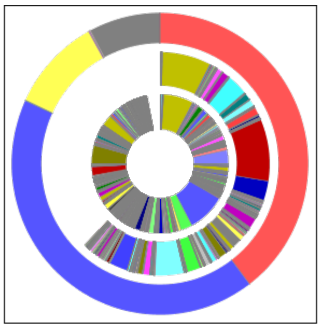
Tightness: 0.34, 0.47
Overlap: 0.57, 0.67
MinCoverage: 0.36, 0.51
Coverage: 0.53, 0.65
MaxCoverage: 0.67, 0.78
Coupling: 0.14, 0.15
Modularity: 0.282
Largest DC1: 4.92%
Largest DC2: 4.89%
Largest Community: 28.19%

(2) acct-6.5.5



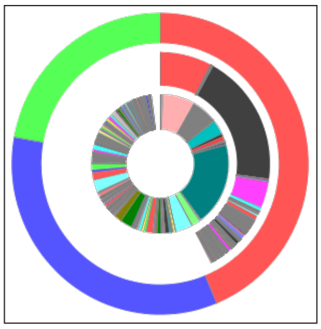
Tightness: 0.16, 0.22
Overlap: 0.41, 0.48
MinCoverage: 0.19, 0.28
Coverage: 0.41, 0.46
MaxCoverage: 0.57, 0.63
Coupling: 0.08, 0.06
Modularity: 0.306
Largest DC1: 15.57%
Largest DC2: 15.57%
Largest Community: 28.49%

(3) acm-5.1



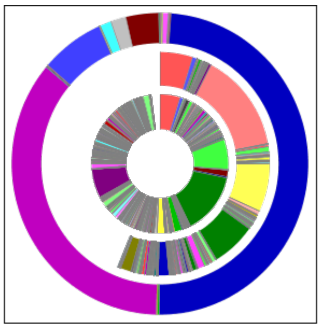
Tightness: 0.58, 0.63
Overlap: 0.79, 0.83
MinCoverage: 0.6, 0.66
Coverage: 0.67, 0.71
MaxCoverage: 0.72, 0.76
Coupling: 0.43, 0.43
Modularity: 0.0535
Largest DC1: 8.94%
Largest DC2: 8.94%
Largest Community: 42.19%

(4) adns-1.3



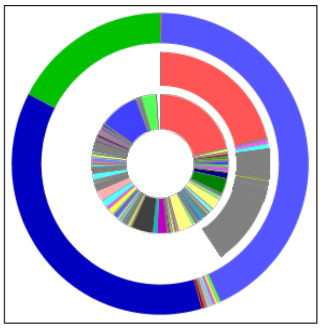
Tightness: 0.29, 0.4
Overlap: 0.5, 0.66
MinCoverage: 0.29, 0.43
Coverage: 0.41, 0.55
MaxCoverage: 0.65, 0.79
Coupling: 0.44, 0.41
Modularity: 0.0844
Largest DC1: 19.17%
Largest DC2: 19.17%
Largest Community: 43.79%

(5) aeneas-1.2



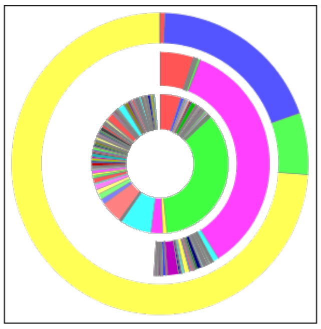
Tightness: 0.52, 0.6
Overlap: 0.74, 0.78
MinCoverage: 0.54, 0.64
Coverage: 0.63, 0.71
MaxCoverage: 0.71, 0.79
Coupling: 0.39, 0.39
Modularity: 0.0257
Largest DC1: 14.33%
Largest DC2: 14.33%
Largest Community: 48.91%

(6) anubis-4.1



Tightness: 0.17, 0.24
Overlap: 0.32, 0.45
MinCoverage: 0.18, 0.27
Coverage: 0.39, 0.52
MaxCoverage: 0.71, 0.84
Coupling: 0.54, 0.47
Modularity: 0.103
Largest DC1: 21.29%
Largest DC2: 21.29%
Largest Community: 43.21%

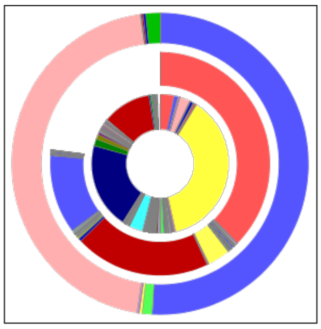
(7) archimedes-1.2.0



Tightness: 0.44, 0.53
Overlap: 0.75, 0.76
MinCoverage: 0.45, 0.55
Coverage: 0.53, 0.63
MaxCoverage: 0.64, 0.73
Coupling: 0.41, 0.4
Modularity: 0.0277
Largest DC1: 35.12%
Largest DC2: 35.09%
Largest Community: 73.79%

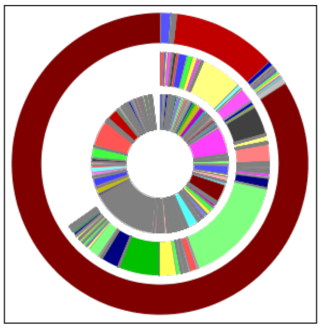
(8) barcode-0.98

Figure A.1: Dependence cluster and dependence community BSG partitions (Type 1 - center, Type 2 - middle, communities - outside)



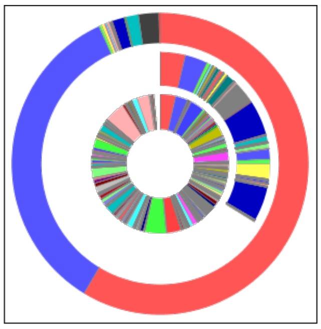
Tightness: 0.71, 0.72
Overlap: 0.81, 0.84
MinCoverage: 0.72, 0.75
Coverage: 0.82, 0.8
MaxCoverage: 0.85, 0.85
Coupling: 0.52, 0.47
Modularity: 0.206
Largest DC1: 37.58%
Largest DC2: 37.58%
Largest Community: 50.77%

(9) bc-1.06



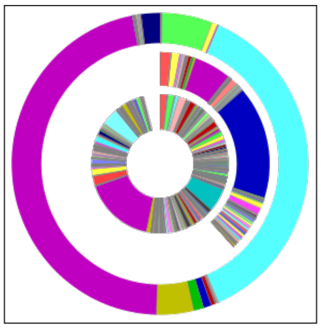
Tightness: 0.57, 0.64
Overlap: 0.77, 0.8
MinCoverage: 0.59, 0.67
Coverage: 0.7, 0.74
MaxCoverage: 0.77, 0.82
Coupling: 0.36, 0.38
Modularity: 0.0454
Largest DC1: 15.48%
Largest DC2: 15.45%
Largest Community: 83.95%

(10) cflow-1.3



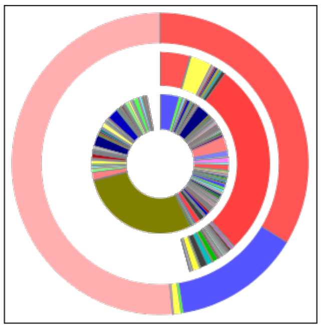
Tightness: 0.28, 0.44
Overlap: 0.62, 0.7
MinCoverage: 0.29, 0.47
Coverage: 0.38, 0.57
MaxCoverage: 0.43, 0.66
Coupling: 0.17, 0.16
Modularity: 0.0935
Largest DC1: 5.08%
Largest DC2: 5.02%
Largest Community: 58.57%

(11) combine-0.3.4



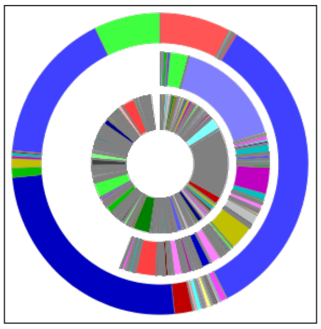
Tightness: 0.35, 0.44
Overlap: 0.68, 0.66
MinCoverage: 0.37, 0.49
Coverage: 0.48, 0.61
MaxCoverage: 0.63, 0.75
Coupling: 0.21, 0.17
Modularity: 0.123
Largest DC1: 16.43%
Largest DC2: 16.43%
Largest Community: 46.61%

(12) cppi-1.15



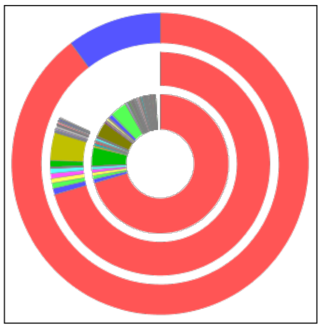
Tightness: 0.37, 0.48
Overlap: 0.53, 0.66
MinCoverage: 0.38, 0.52
Coverage: 0.51, 0.65
MaxCoverage: 0.59, 0.8
Coupling: 0.34, 0.33
Modularity: 0.0539
Largest DC1: 28.33%
Largest DC2: 28.14%
Largest Community: 51.26%

(13) diction-1.11



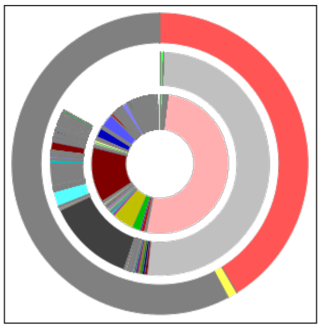
Tightness: 0.52, 0.54
Overlap: 0.7, 0.71
MinCoverage: 0.54, 0.58
Coverage: 0.65, 0.69
MaxCoverage: 0.76, 0.81
Coupling: 0.17, 0.14
Modularity: 0.178
Largest DC1: 16.1%
Largest DC2: 16.1%
Largest Community: 33.86%

(14) diffutils-3.2



Tightness: 0.77, 0.76
Overlap: 0.88, 0.86
MinCoverage: 0.79, 0.78
Coverage: 0.83, 0.82
MaxCoverage: 0.87, 0.87
Coupling: 0.76, 0.71
Modularity: 0.00692
Largest DC1: 70.57%
Largest DC2: 70.57%
Largest Community: 89.73%

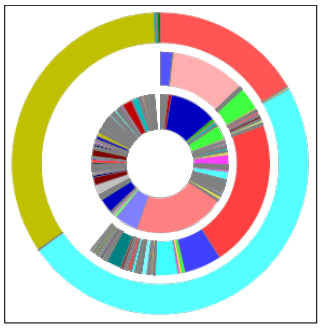
(15) ed-1.5



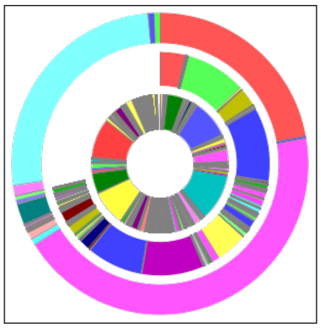
Tightness: 0.78, 0.79
Overlap: 0.95, 0.92
MinCoverage: 0.79, 0.81
Coverage: 0.81, 0.83
MaxCoverage: 0.83, 0.87
Coupling: 0.67, 0.63
Modularity: 0.000247
Largest DC1: 51.27%
Largest DC2: 51.27%
Largest Community: 57.56%

(16) empire-4.3.28

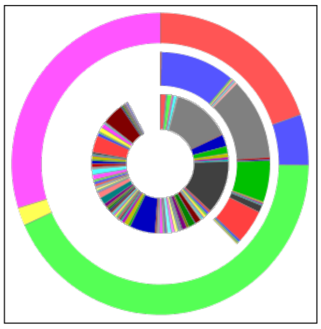
Figure A.1: Dependence cluster and dependence community BSG partitions (Type 1 - center, Type 2 - middle, communities - outside)


Tightness: 0.53, 0.6
Overlap: 0.8, 0.8
MinCoverage: 0.54, 0.63
Coverage: 0.62, 0.7
MaxCoverage: 0.68, 0.77
Coupling: 0.27, 0.26
Modularity: 0.481
Largest DC1: 21.46%
Largest DC2: 21.43%
Largest Community: 48.51%

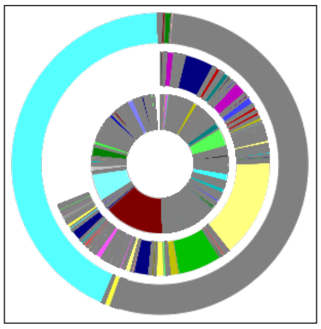
(17) enscript-1.6.5


Tightness: 0.69, 0.7
Overlap: 0.85, 0.85
MinCoverage: 0.7, 0.73
Coverage: 0.76, 0.79
MaxCoverage: 0.8, 0.84
Coupling: 0.43, 0.41
Modularity: 0.00207
Largest DC1: 10.73%
Largest DC2: 10.73%
Largest Community: 43.65%

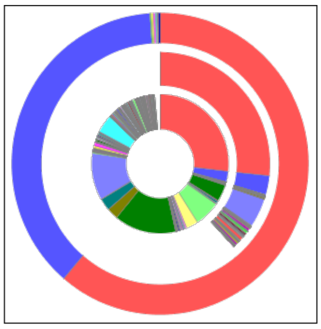
(18) findutils-4.4.2


Tightness: 0.36, 0.26
Overlap: 0.62, 0.51
MinCoverage: 0.38, 0.31
Coverage: 0.52, 0.48
MaxCoverage: 0.67, 0.68
Coupling: 0.38, 0.31
Modularity: 0.108
Largest DC1: 13.73%
Largest DC2: 11.8%
Largest Community: 43.23%

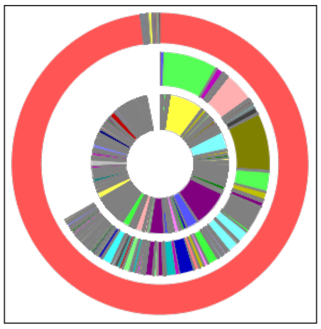
(19) garpd-0.2.0


Tightness: 0.64, 0.64
Overlap: 0.82, 0.8
MinCoverage: 0.65, 0.67
Coverage: 0.7, 0.73
MaxCoverage: 0.74, 0.79
Coupling: 0.11, 0.11
Modularity: 0.294
Largest DC1: 14.24%
Largest DC2: 14.24%
Largest Community: 53.12%

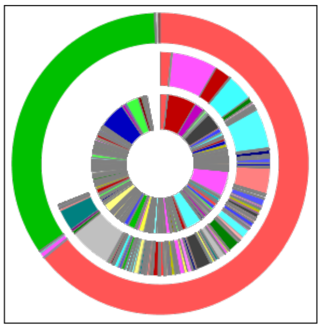
(20) gettext-0.18


Tightness: 0.11, 0.44
Overlap: 0.17, 0.61
MinCoverage: 0.12, 0.48
Coverage: 0.25, 0.64
MaxCoverage: 0.53, 0.8
Coupling: 0.4, 0.39
Modularity: 0.0872
Largest DC1: 26.73%
Largest DC2: 26.73%
Largest Community: 61.26%

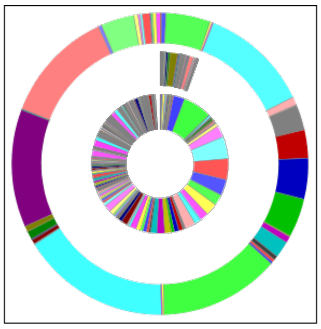
(21) gforth-0.7.0


Tightness: 0.6, 0.67
Overlap: 0.73, 0.82
MinCoverage: 0.61, 0.7
Coverage: 0.71, 0.76
MaxCoverage: 0.75, 0.81
Coupling: 0.35, 0.34
Modularity: 7.47e-05
Largest DC1: 8.22%
Largest DC2: 8.22%
Largest Community: 97.8%

(22) global-6.0

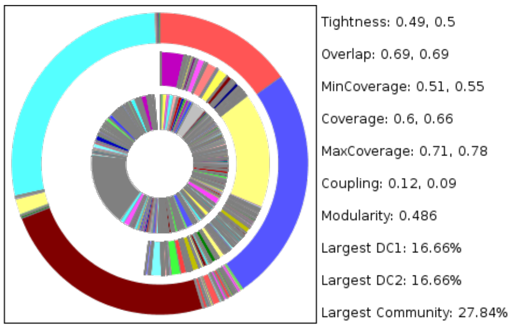

Tightness: 0.65, 0.7
Overlap: 0.8, 0.83
MinCoverage: 0.67, 0.73
Coverage: 0.74, 0.79
MaxCoverage: 0.79, 0.83
Coupling: 0.56, 0.55
Modularity: 0.000332
Largest DC1: 6.59%
Largest DC2: 6.59%
Largest Community: 64.24%

(23) gnats-4.1.0

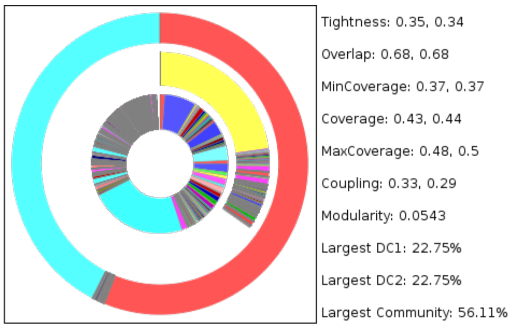

Tightness: 0.09, 0.18
Overlap: 0.35, 0.43
MinCoverage: 0.12, 0.23
Coverage: 0.23, 0.4
MaxCoverage: 0.41, 0.58
Coupling: 0.01, 0.01
Modularity: 0.483
Largest DC1: 6.76%
Largest DC2: 1.14%
Largest Community: 16.18%

(24) gnubik-2.4

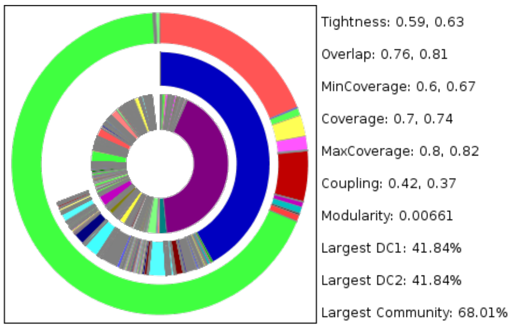
Figure A.1: Dependence cluster and dependence community BSG partitions (Type 1 - center, Type 2 - middle, communities - outside)


Tightness: 0.49, 0.5
Overlap: 0.69, 0.69
MinCoverage: 0.51, 0.55
Coverage: 0.6, 0.66
MaxCoverage: 0.71, 0.78
Coupling: 0.12, 0.09
Modularity: 0.486
Largest DC1: 16.66%
Largest DC2: 16.66%
Largest Community: 27.84%

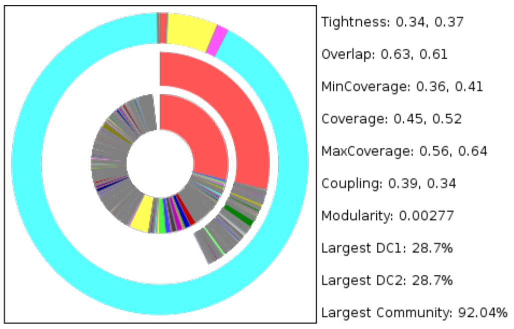
(25) gnuchess-6.0.1


Tightness: 0.35, 0.34
Overlap: 0.68, 0.68
MinCoverage: 0.37, 0.37
Coverage: 0.43, 0.44
MaxCoverage: 0.48, 0.5
Coupling: 0.33, 0.29
Modularity: 0.0543
Largest DC1: 22.75%
Largest DC2: 22.75%
Largest Community: 56.11%

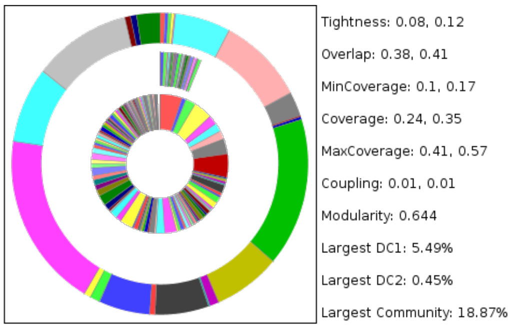
(26) gnuedma-0.18.1


Tightness: 0.59, 0.63
Overlap: 0.76, 0.81
MinCoverage: 0.6, 0.67
Coverage: 0.7, 0.74
MaxCoverage: 0.8, 0.82
Coupling: 0.42, 0.37
Modularity: 0.00661
Largest DC1: 41.84%
Largest DC2: 41.84%
Largest Community: 68.01%

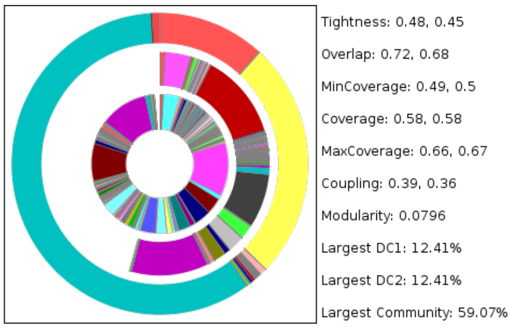
(27) gnuit-4.9.5


Tightness: 0.34, 0.37
Overlap: 0.63, 0.61
MinCoverage: 0.36, 0.41
Coverage: 0.45, 0.52
MaxCoverage: 0.56, 0.64
Coupling: 0.39, 0.34
Modularity: 0.00277
Largest DC1: 28.7%
Largest DC2: 28.7%
Largest Community: 92.04%

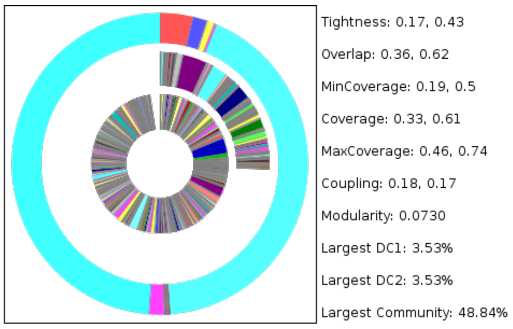
(28) gnujump-1.0.5


Tightness: 0.08, 0.12
Overlap: 0.38, 0.41
MinCoverage: 0.1, 0.17
Coverage: 0.24, 0.35
MaxCoverage: 0.41, 0.57
Coupling: 0.01, 0.01
Modularity: 0.644
Largest DC1: 5.49%
Largest DC2: 0.45%
Largest Community: 18.87%

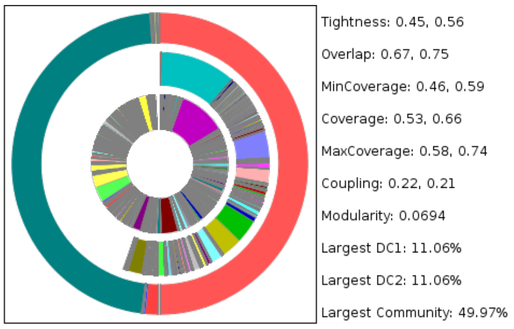
(29) gnurobots-1.2.0


Tightness: 0.48, 0.45
Overlap: 0.72, 0.68
MinCoverage: 0.49, 0.5
Coverage: 0.58, 0.58
MaxCoverage: 0.66, 0.67
Coupling: 0.39, 0.36
Modularity: 0.0796
Largest DC1: 12.41%
Largest DC2: 12.41%
Largest Community: 59.07%

(30) gnushogi-1.3

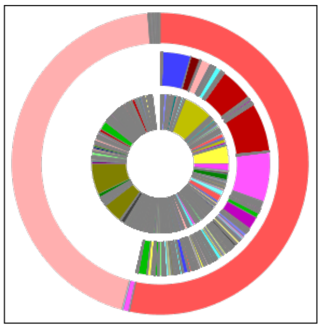

Tightness: 0.17, 0.43
Overlap: 0.36, 0.62
MinCoverage: 0.19, 0.5
Coverage: 0.33, 0.61
MaxCoverage: 0.46, 0.74
Coupling: 0.18, 0.17
Modularity: 0.0730
Largest DC1: 3.53%
Largest DC2: 3.53%
Largest Community: 48.84%

(31) gperf-3.0.4


Tightness: 0.45, 0.56
Overlap: 0.67, 0.75
MinCoverage: 0.46, 0.59
Coverage: 0.53, 0.66
MaxCoverage: 0.58, 0.74
Coupling: 0.22, 0.21
Modularity: 0.0694
Largest DC1: 11.06%
Largest DC2: 11.06%
Largest Community: 49.97%

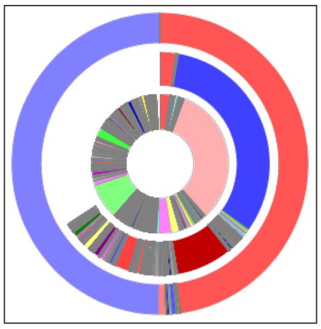
(32) inetutils-1.8

Figure A.1: Dependence cluster and dependence community BSG partitions (Type 1 - center, Type 2 - middle, communities - outside)



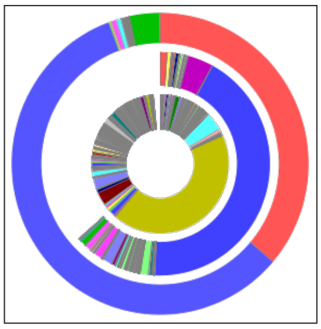
Tightness: 0.49, 0.53
Overlap: 0.71, 0.71
MinCoverage: 0.51, 0.58
Coverage: 0.61, 0.69
MaxCoverage: 0.74, 0.81
Coupling: 0.44, 0.34
Modularity: 0.0482
Largest DC1: 7.13%
Largest DC2: 7.12%
Largest Community: 53.3%

(33) lame-3.99.1



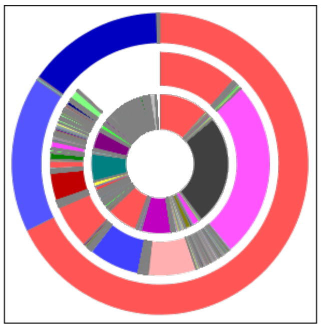
Tightness: 0.55, 0.55
Overlap: 0.84, 0.8
MinCoverage: 0.56, 0.58
Coverage: 0.61, 0.63
MaxCoverage: 0.65, 0.7
Coupling: 0.3, 0.28
Modularity: 8.38e-05
Largest DC1: 32.37%
Largest DC2: 32.37%
Largest Community: 49.74%

(34) ntp-4.2.6p5-RC1



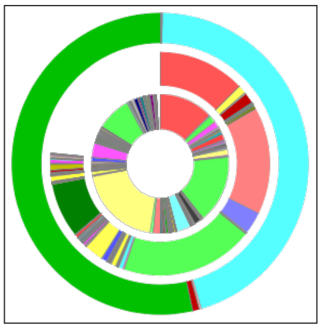
Tightness: 0.59, 0.66
Overlap: 0.73, 0.79
MinCoverage: 0.61, 0.69
Coverage: 0.7, 0.76
MaxCoverage: 0.79, 0.83
Coupling: 0.4, 0.39
Modularity: 0.0595
Largest DC1: 42.96%
Largest DC2: 42.84%
Largest Community: 58.1%

(35) pure-ftpd-1.0.32



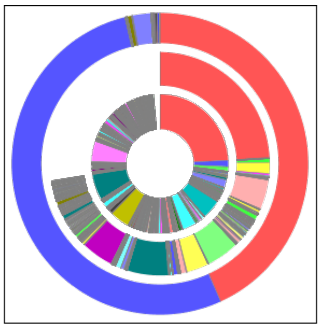
Tightness: 0.8, 0.8
Overlap: 0.91, 0.9
MinCoverage: 0.81, 0.81
Coverage: 0.83, 0.84
MaxCoverage: 0.84, 0.88
Coupling: 0.86, 0.8
Modularity: 0.00965
Largest DC1: 25.7%
Largest DC2: 25.7%
Largest Community: 67.69%

(36) rsync-3.0.9



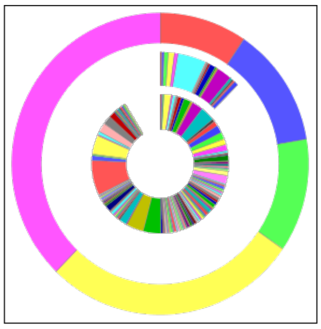
Tightness: 0.74, 0.69
Overlap: 0.9, 0.84
MinCoverage: 0.76, 0.72
Coverage: 0.79, 0.78
MaxCoverage: 0.82, 0.83
Coupling: 0.65, 0.6
Modularity: 0.0248
Largest DC1: 19.63%
Largest DC2: 19.53%
Largest Community: 53.53%

(37) sed-4.2



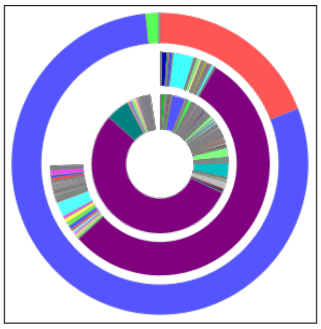
Tightness: 0.64, 0.7
Overlap: 0.84, 0.85
MinCoverage: 0.65, 0.73
Coverage: 0.7, 0.78
MaxCoverage: 0.74, 0.83
Coupling: 0.54, 0.52
Modularity: 0.00109
Largest DC1: 24.15%
Largest DC2: 24.15%
Largest Community: 52.9%

(38) tar-1.23



Tightness: 0.08, 0.17
Overlap: 0.27, 0.45
MinCoverage: 0.11, 0.22
Coverage: 0.26, 0.4
MaxCoverage: 0.43, 0.62
Coupling: 0.21, 0.24
Modularity: 0.0853
Largest DC1: 8.11%
Largest DC2: 4.12%
Largest Community: 37.65%

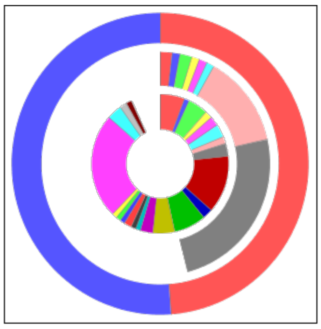
(39) time-1.7



Tightness: 0.67, 0.65
Overlap: 0.85, 0.83
MinCoverage: 0.69, 0.69
Coverage: 0.74, 0.74
MaxCoverage: 0.79, 0.8
Coupling: 0.53, 0.49
Modularity: 0.0715
Largest DC1: 54.86%
Largest DC2: 54.86%
Largest Community: 79.41%

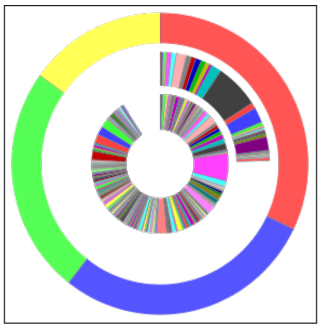
(40) userv-1.0.3

Figure A.1: Dependence cluster and dependence community BSG partitions (Type 1 - center, Type 2 - middle, communities - outside)



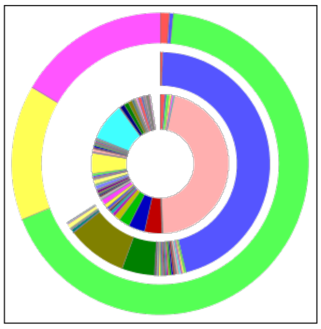
Tightness: 0.38, 0.37
Overlap: 0.67, 0.64
MinCoverage: 0.42, 0.42
Coverage: 0.53, 0.53
MaxCoverage: 0.59, 0.58
Coupling: 0.45, 0.4
Modularity: 0.136
Largest DC1: 24.42%
Largest DC2: 24.42%
Largest Community: 51.16%

(41) wc



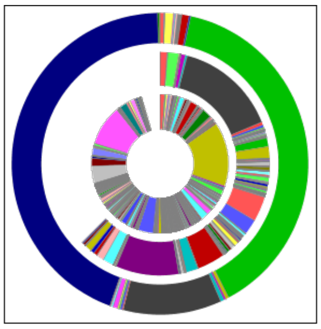
Tightness: 0.2, 0.29
Overlap: 0.4, 0.54
MinCoverage: 0.23, 0.35
Coverage: 0.4, 0.51
MaxCoverage: 0.59, 0.67
Coupling: 0.35, 0.3
Modularity: 0.107
Largest DC1: 6.53%
Largest DC2: 6.53%
Largest Community: 32.13%

(42) wdiff-0.5



Tightness: 0.55, 0.62
Overlap: 0.73, 0.8
MinCoverage: 0.57, 0.65
Coverage: 0.67, 0.72
MaxCoverage: 0.77, 0.8
Coupling: 0.62, 0.58
Modularity: 0.0181
Largest DC1: 45.82%
Largest DC2: 45.82%
Largest Community: 67.57%

(43) which-2.20



Tightness: 0.61, 0.58
Overlap: 0.73, 0.74
MinCoverage: 0.62, 0.61
Coverage: 0.74, 0.72
MaxCoverage: 0.81, 0.82
Coupling: 0.56, 0.49
Modularity: 0.0407
Largest DC1: 14.92%
Largest DC2: 14.92%
Largest Community: 44.27%

(44) zlib-1.2.5

Figure A.1: Dependence cluster and dependence community BSG partitions (Type 1 - center, Type 2 - middle, communities - outside)

Bibliography

[1] GraphChi, 2012. URL `http://graphlab.org/graphchi/`.

[2] Hiralal Agrawal. On slicing programs with jump statements. In *ACM Sigplan Notices*, volume 29, pages 302–312, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: 10.1145/178243.178456.

[3] Hiralal Agrawal and J.R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Notices*, volume 25, pages 246–256. ACM, 1990.

[4] Réka Albert. *Statistical mechanics of complex networks.* Phd, University of Notre Dame, January 2001.

[5] Réka Albert. Scale-free networks in cell biology. *Journal of cell science*, 118 (Pt 21):4947–57, November 2005. ISSN 0021-9533. doi: 10.1242/jcs.02714.

[6] Matthew Allen and Susan Horwitz. Slicing java programs that throw and catch exceptions. pages 44–54, New York, NY, USA, 2003. ACM. ISBN 1-58113-667-6. doi: 10.1145/777388.777394.

[7] Paul Anderson and Tim Teitelbaum. Software inspection using codesurfer. In *Proceedings of WISE01 International Workshop on Inspection in Software Engineering*, pages 1–9, 2001.

[8] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. *Automated and Algorithmic Debugging*, pages 206–222, 1993.

[9] A. Barabási and Reka Albert. Emergence of scaling in random networks. *Science*, 286(5439):11, October 1999. ISSN 00368075. doi: 10.1126/science.286.5439.509.

[10] Albert-László Barabási and Eric Bonabeau. Scale-free Networks. *Scientific American*, (May):50–59, May 2003.

[11] Albert-Laszlo Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means for Business, Science and Everyday Life*. Plume, 2nd edition, April 2009. ISBN 978-0-452-28439-5.

[12] S.S. Barpanda and D.P. Mohapatra. Dynamic slicing of distributed object-oriented programs. *IET Software*, 5(5):425, 2011. ISSN 17518806. doi: 10.1049/iet-sen.2010.0141.

[13] J.M. Bieman and L.M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, 1994. ISSN 00985589. doi: 10.1109/32.310673.

[14] D. Binkley and M. Harman. An empirical study of predicate dependence levels and trends. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 330–339. IEEE, 2003. ISBN 0-7695-1877-X. doi: 10.1109/ICSE.2003.1201212.

[15] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects. *IEEE Transactions on Software Engineering*, 32(9):698–717, September 2006. ISSN 0098-5589. doi: 10.1109/TSE.2006.95.

[16] David Binkley and Mark Harman. Locating Dependence Clusters and Dependence Pollution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 177–186. IEEE, September 2005. ISBN 0-7695-2368-4. doi: 10.1109/ICSM.2005.58.

[17] David Binkley and Mark Harman. Animated Visualisation of Static Analysis : Characterising , Explaining and Exploiting the Approximate Nature of Static

Analysis. In *6th International Workshop on Source Code Analysis and Manipulation (SCAM 06)*, pages 43–52, 2006. URL `http://www.cs.loyola.edu/~binkley/scam06.html`.

[18] David Binkley and Mark Harman. Identifying 'Linchpin Vertices' That Cause Large Dependence Clusters. In *Source Code Analysis and Manipulation, IEEE International Workshop on*, pages 89–98. Ieee, 2009. ISBN 978-0-7695-3793-1. doi: 10.1109/SCAM.2009.18.

[19] David Binkley, Mark Harman, and Jens Krinke. Empirical study of optimization techniques for massive slicing. *ACM Transactions on Programming Languages and Systems*, 30(1):3–es, November 2007. ISSN 01640925. doi: 10.1145/1290520.1290523.

[20] David Binkley, Nicolas Gold, Mark Harman, Zheng Li, and Kiarash Mahdavi. Evaluating Key Statements Analysis. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 121–130. IEEE, September 2008. ISBN 978-0-7695-3353-7. doi: 10.1109/SCAM.2008.40.

[21] David Binkley, Nicolas Gold, Mark Harman, Zheng Li, Kiarash Mahdavi, Joachim Wegener, and Y Xie. Dependence Anti Patterns. In *Automated Software Engineering - Workshops, 2008. ASE Workshops 2008*, pages 25 – 34, 2008.

[22] 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, January 2010. ISSN 01641212. doi: 10.1016/j.jss.2009.03.038.

[23] Sue E Black, Steve Counsell, Tracy Hall, and David Bowes. Fault analysis in OSS based on program slicing metrics. In *Software Engineering and Advanced*

Applications, 2009. SEAA '09. 35th Euromicro Conference on. IEEE, 2009. doi: 10.1109/SEAA.2009.94.

[24] 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, October 2008. ISSN 1742-5468. doi: 10.1088/1742-5468/2008/10/P10008.

[25] S Boccaletti, V Latora, Y Moreno, M Chavez, and D Hwang. Complex networks: Structure and dynamics. *Physics Reports*, 424(4-5):175–308, February 2006. ISSN 03701573. doi: 10.1016/j.physrep.2005.10.009.

[26] Korel Bogdan and Lasi Janusz. Dynamic program slicing. *Information Processing Letters*, 29:155–163, 1988.

[27] D. Bowes, S. Counsell, and Tracy Hall. Calibrating program slicing metrics for practical use. In *Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART 2009).* IEEE, 2009. ISBN 978-1-4244-497774.

[28] David Bowes, Tracy Hall, Andrew Kerr, and Kingston Lane. Program Slicing-Based Cohesion Measurement: The Challenges of Replicating Studies Using Metrics. In *Proceeding of the 2nd international workshop on Emerging trends in software metrics - WETSoM '11*, pages 75–80, New York, New York, USA, May 2011. ACM Press. ISBN 9781450305938. doi: 10.1145/1985374.1985392.

[29] D. Bu. Topological structure analysis of the protein-protein interaction network in budding yeast. *Nucleic Acids Research*, 31(9):2443–2450, May 2003. ISSN 13624962. doi: 10.1093/nar/gkg340.

[30] Alexander Chatzigeorgiou, Nikolaos Tsantalis, and George Stephanides. Application of graph theory to OO software engineering. In *Proceedings of the*

2006 international workshop on Workshop on interdisciplinary software engineering research - WISER '06, page 29, New York, New York, USA, May 2006. ACM Press. ISBN 159593409X. doi: 10.1145/1137661.1137669.

[31] Zhenqiang Chen and Baowen Xu. Slicing Object-Oriented Java Programs. *ACM SIGPLAN Notices*, 36(4):33–40, April 2001. ISSN 0362-1340.

[32] Zhenqiang Chen and Baowen Xu. Slicing concurrent java programs. *ACM SIGPLAN Notices*, 36(4):41–47, April 2001. ISSN 03621340. doi: 10.1145/375431.375420.

[33] Zhenqiang Chen, Baowen Xu, and Jianjun Zhao. An overview of methods for dependence analysis of concurrent programs. *ACM SIGPLAN Notices*, 37(8):45, August 2002. ISSN 03621340. doi: 10.1145/596992.597003.

[34] Jingde Cheng. Slicing Concurrent Programs - A Graph-theoretical Approach. In *1st Automatic Algorithmic Debugging Conference1*, pages 223–240, 1993.

[35] Jong-deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Trans. Program. Lang. Syst.*, 16(4):1097–1113, 1994. ISSN 0164-0925. doi: 10.1145/183432.183438.

[36] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. pages 12–12, Washington, DC, 2003. USENIX Association.

[37] Alessandro Colantonio and Roberto Di. Concise: Compressed 'n' Composable Integer Set. *CoRR*, 2010.

[38] Christian Collberg. Software watermarking: Models and dynamic embeddings. In *Proceedings of the 26th ACM SIGPLAN-SIGACT*, January 1999.

[39] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley, 2009. ISBN 0321549252.

[40] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, January 1998. ISBN 0897919793.

[41] Christian Collberg, Clark Thomborson, and Gregg M. Townsend. Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.*, 29 (6):35, 2007.

[42] Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. Power-Laws in a Large Object-Oriented Software System. *IEEE Transactions on Software Engineering*, 33(10):687–708, October 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.1019.

[43] Giulio Concas, Michele Marchesi, Alessandro Murgia, and Roberto Tonelli. An Empirical Study of Social Networks Metrics in Object-Oriented Software. *Advances in Software Engineering*, 2010:1–21, 2010. ISSN 1687-8655. doi: 10.1155/2010/729826.

[44] Thomas Corbat. Dependence Graphs for Slicing and Refactoring. 2008.

[45] S. Counsell, T. Hall, E. Nasseri, and D. Bowes. An Analysis of the 'Inconclusive' Change Report Category in OSS Assisted by a Program Slicing Metric. In *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 283–286. IEEE, September 2010. ISBN 978-1-4244-7901-6. doi: 10.1109/SEAA.2010.17.

[46] Steve Counsell, Tracy Hall, and David Bowes. Program Slice Metrics and Their Potential Role in DSL Design. In *Workshop on Knowledge Industry Survival Strategy Initiative. ASWEC 2009: 20th Australian Software Engineering Conference.*, Brisbane, 2009.

[47] Sebastian Danicic, Richard Barraclough, Mark Harman, John Howroyd, Akos Kiss, and Mike Laurence. A unifying theory of control dependence and its application to arbitrary program structures. *Theoretical Computer Science*, October 2011.

[48] D. Doval, S. Mancoridis, and B.S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Software Technology and Engineering Practice, 1999. STEP '99.*, pages 73–81. IEEE Comput. Soc, 1999. ISBN 0-7695-0328-4. doi: 10.1109/STEP.1999.798481.

[49] S. Drape and A. Majumdar. Design and evaluation of slicing obfuscations. Technical report, The University of Auckland, New Zealand, Centre for Discrete Mathematics & Theoretical Computer Science, June 2007.

[50] Stephen Drape and Anirban Majumdar. Slicing aided design of obfuscating transforms. *Computer and Information Science, ACIS International Conference on*, 0:1019–1024, 2007. doi: 10.1109/ICIS.2007.167.

[51] Antonio Castaldo D'Ursi, Luca Cavallaro, and Mattia Monga. On bytecode slicing and aspectJ interferences. pages 35–43, New York, NY, USA, 2007. ACM. ISBN 1-59593-661-5. doi: 10.1145/1233833.1233839.

[52] Arnini Einarsson and Janus Dam Nielsen. A Survivor's Guide to Java Program Analysis with Soot, 2008. URL `www.brics.dk/SootGuide/sootsurvivorsguide.ps`.

[53] P Erdős and A. Rényi. On the strength of connectedness of a random graph. *Acta Mathematica Hungarica*, 12(1-2):261–267, 1961. doi: 10.1007/BF02066689.

[54] T. Evans and R. Lambiotte. Line graphs, link partitions, and overlapping communities. *Physical Review E*, 80(1):9, July 2009. ISSN 1539-3755. doi: 10.1103/PhysRevE.80.016105.

[55] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, January 1998. ISBN 0534954251.

[56] 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, July 1987. ISSN 01640925. doi: 10.1145/24039.24041.

[57] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5): 75–174, February 2010. ISSN 03701573. doi: 10.1016/j.physrep.2009.11.002.

[58] Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences of the United States of America*, 104(1):36–41, January 2007. ISSN 0027-8424. doi: 10.1073/pnas.0605965104.

[59] Linton C. Freeman. *The Development of Social Network Analysis*. Empirical Press, Vancouver, BC, Canada, 2004. ISBN 1594577145.

[60] K. B. Gallagher and J. R. Lyle. Using Program Slicing in Software Maintenance. *IEEE Transactions of Software Engineering*, 17(8):751 – 761, 1991.

[61] K. B. Gallagher and Liam O'Brian. Analyzing programs via decomposition slicing: Initial data and observations. In *7th IEEE Workshop on Empirical Studies of Software*, 2001.

[62] Keith Gallagher and David Binkley. An Empirical Study of Computation Equivalence as Determined by Decomposition Slice Equivalence. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2027-8.

[63] Keith Gallagher and Liam O Brien. Reducing Visualization Complexity Using Decomposition Slices. In *Software Visualization Workshop*, 1997.

[64] Keith Gallagher and Lucas Layman. Are decomposition slices clones? In *11th IEEE International Workshop on Program Comprehension*. IEEE, 2003.

[65] Dennis Giffhorn and Christian Hammer. An Evaluation of Slicing Algorithms for Concurrent Programs. *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 17–26, September 2007. doi: 10.1109/SCAM.2007.9.

[66] E. N. Gilbert. Random Graphs. *The Annals of Mathematical Statistics*, 30 (4):1141–1144, December 1959. ISSN 0003-4851.

[67] Ginger Myles. Using Software Watermarking to Discourage Piracy. *Crossroads - The ACM Student Magazine*, 2004. URL `http://www.acm.org/crossroads/xrds10-3/watermarking.html`.

[68] M Girvan and M E J Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America*, 99(12):7821–6, June 2002. ISSN 0027-8424. doi: 10.1073/pnas.122653799.

[69] GNU. bc, 2011. URL `http://www.gnu.org/software/bc/`.

[70] GNU. GNU Chess 6.0.0, 2011. URL `http://www.gnu.org/software/chess/manual/gnuchess.html`.

[71] N. E. Gold, M. Harman, D. Binkley, and R. M. Hierons. Unifying program slicing and concept assignment for higher-level executable source code extraction. *Software: Practice and Experience*, 35(10):977–1006, August 2005. ISSN 0038-0644. doi: 10.1002/spe.664.

[72] GrammaTech Inc. CodeSurfer, 2011. URL `www.grammatech.com`.

[73] GrammaTech Inc. Dependence Graphs and Program Slicing. 2011. URL `http://www.grammatech.com/research/papers/slicing/slicingWhitepaper.html`.

[74] Pam Green, Peter C R Lane, Austen Rainer, and S Scholz. An Introduction to Slice-Based Cohesion and Coupling Metrics. Technical Report 488, University of Hertfordshire, 2009.

[75] James Hamilton and Sebastian Danicic. An Evaluation of Static Java Bytecode Watermarking. In *Proceedings of the International Conference on Computer Science and Applications (ICCSA'10), The World Congress on Engineering and Computer Science (WCECS'10)*, volume 1, pages 1 – 8, San Francisco, USA, October 2010. ISBN 978-988-17012-0-6.

[76] James Hamilton and Sebastian Danicic. An Evaluation of the Resilience of Static Java Bytecode Watermarks Against Distortive Attacks. *IAENG International Journal of Computer Science*, 38(1):1–15, 2011.

[77] James Hamilton and Sebastian Danicic. A Survey of Static Software Watermarking. In *Proceedings of the World Congress on Internet Security 2011*, pages 114–121, London, 2011. IEEE.

[78] James Hamilton and Sebastian Danicic. Dependence Communities in Source Code. In *IEEE 28th International Conference on Software Maintenance, Early Research Achievements Track*. IEEE, 2012.

[79] Zhang Haohua, Zhao Hai, Cai Wei, Zhao Ming, and Luo Guilan. A metrics suite for static structure of large-scale software based on complex networks. In *2008 International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, Proceedings - 2008 4th International Conference on Intelligent Information Hiding and Multimedia Signal Processing, IIH-MSP

2008, pages 512–515. Inst. of Elec. and Elec. Eng. Computer Society, 2008. ISBN 9780769532783. doi: 10.1109/IIH-MSP.2008.293.

[80] M. Harman, S. Danicic, Y. Sivagurunathan, and B. Jones. Cohesion Metrics. In *8th International Quality Week*, pages 1–14, San Francisco, USA, 1995.

[81] Mark Harman. Cleaving Together: Program Cohesion with Slices. *EXE*, 11 (8):35 – 42, 1997.

[82] Mark Harman. The Current State and Future of Search Based Software Engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.29.

[83] Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance*, 10(6):1–30, 1998. ISSN 1040-550X.

[84] Mark Harman and Robert M. Hierons. An Overview of Program Slicing. Technical report, Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex. URL `http://www.cs.ucl.ac.uk/staff/mharman/sf.html`.

[85] Mark Harman, Margaret Okulawon, Bala Sivagurunathan, and Sebastian Danicic. Slice-based measurement of function coupling. In *ICSE workshop on Process Modelling and Empirical Studies of Software Evolution (PMESSE'97)*, pages 28–32. IEEE/ACM, 1997.

[86] Mark Harman, Nicolas Gold, R. Hierons, and David Binkley. Code extraction algorithms which unify slicing and concept assignment. In *Working Conference on Reverse Engineering*, pages 11 – 21, Los Alamitos, CA, USA, October 2002. IEEE Computer Society Press.

[87] Mark Harman, David Binkley, Keith Gallagher, Nicolas Gold, and Jens Krinke. Dependence clusters in source code. *ACM Transactions on Programming Languages and Systems*, 32(1):1–33, October 2009. ISSN 01640925. doi: 10.1145/1596527.1596528.

[88] John Hatcliff, James Corbett, Matthew Dwyer, Stefan Sokolowski, Hongjun Zheng, Agostino Cortesi, and Gilberto Filé. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. *Lecture Notes in Computer Science*, 1694:848, October 1999. doi: 10.1007/3-540-48294-6.

[89] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. pages 238–250, Denver, Colorado, United States, 1972. ACM.

[90] S. Henry and D. Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, September 1981. ISSN 0098-5589. doi: 10.1109/TSE.1981.231113.

[91] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990. ISSN 01640925. doi: 10.1145/77606.77608.

[92] David Hyland-Wood, David Carrington, and Simon Kaplan. Scale-Free Nature of Java Software Package, Class and Method Collaboration Graphs. *Electrical Engineering*, 2006.

[93] Syed S. Islam. *Dependence Cluster Analysis.* Msc, King's College London, 2007.

[94] Syed S. Islam, Jens Krinke, and David Binkley. Dependence cluster visualization. In *Proceedings of the 5th international symposium on Software visualization - SOFTVIS '10*, page 93, New York, New York, USA, October 2010. ACM Press. ISBN 9781450300285. doi: 10.1145/1879211.1879227.

[95] Syed S. Islam, Jens Krinke, David Binkley, and Mark Harman. Coherent dependence clusters. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '10*, PASTE '10, page 53, New York, New York, USA, 2010. ACM Press. ISBN 9781450300827. doi: 10.1145/1806672.1806683.

[96] S. Jenkins and S.R. Kirk. Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution. *Information Sciences*, 177(12):2587–2601, June 2007. ISSN 00200255. doi: 10.1016/j.ins. 2007.01.021.

[97] Liu Jing, He Keqing, Ma Yutao, and Peng Rong. Scale Free in Software Metrics. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 229–235. IEEE, 2006. ISBN 0-7695-2655-1. doi: 10.1109/COMPSAC.2006.75.

[98] Mariam Kamkar, Nahid Shahmehri, Peter Fritzson, Maurice Bruynooghe, and Martin Wirsing. Interprocedural dynamic slicing. *Lecture Notes in Computer Science*, 631:370–384, 1992. doi: 10.1007/3-540-55844-6.

[99] Sakari Karstu. *An examination of the behavior of slice based cohesion measures.* Masters, Michigan Technological University, 1994.

[100] Akos Kiss. *Program Code Analysis and Manipulation.* PhD thesis, September 2008.

[101] B. Korel. Computation of dynamic program slices for unstructured programs. *IEEE Transactions on Software Engineering*, 23(1):17–34, 1997. ISSN 00985589. doi: 10.1109/32.581327.

[102] B Korel and J Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988. ISSN 00200190. doi: 10.1016/ 0020-0190(88)90054-3.

[103] B. Korel and J. Rilling. Dynamic program slicing in understanding of program execution. In *Proceedings Fifth International Workshop on Program Comprehension. IWPC'97*, pages 80–89. IEEE Comput. Soc. Press. ISBN 0-8186-7993-X. doi: 10.1109/WPC.1997.601269.

[104] Bogdan Korel and Jurgen Rilling. Dynamic program slicing methods. *Information and Software Technology*, 40(11-12):647–659, December 1998. ISSN 09505849. doi: 10.1016/S0950-5849(98)00089-5.

[105] Bogdan Korel and Satish Yalamanchili. Forward computation of dynamic program slices. In *Proceedings of the 1994 international symposium on Software testing and analysis - ISSTA '94*, pages 66–79, New York, New York, USA, August 1994. ACM Press. ISBN 0897916832. doi: 10.1145/186258.186514.

[106] J. Krinke. Advanced slicing of sequential and concurrent programs. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 464–468. IEEE, 2004. ISBN 0-7695-2213-0. doi: 10.1109/ICSM.2004.1357836.

[107] Jens Krinke. Static slicing of threaded programs. *ACM SIGPLAN Notices*, 33(7):35–42, July 1998. ISSN 03621340. doi: 10.1145/277633.277638.

[108] Jens Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE '03*, volume 28, page 178, New York, New York, USA, September 2003. ACM Press. ISBN 1581137435. doi: 10.1145/940071.940096.

[109] Jens Krinke. Statement-Level Cohesion Metrics and their Visualization. *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 37–48, September 2007. doi: 10.1109/SCAM.2007.28.

[110] Anand Krishnaswamy. Program slicing: An application of object-oriented program dependency graphs. Technical report, Department of Computer Science, Clemson University,, 1994.

[111] Bradley M Kuhn, Dennis J Smith, and Keith Brian Gallagher. The Decomposition Slice Display System. In *SEKE'95, The 7th International Conference on Software Engineering and Knowledge Engineering*, pages 328–333, Rockville, Maryland, USA, 1995.

[112] Sumit Kumar, Susan Horwitz, Ralf-Detlef Kutsche, and Herbert Weber. Fundamental Approaches to Software Engineering. *Lecture Notes in Computer Science*, 2306:371–388, March 2002. doi: 10.1007/3-540-45923-5.

[113] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi : Large-Scale Graph Computation on Just a PC Disk-based Graph Computation. In *In Proceedings of the 10th USENIX Symposium on Operating System Design and Implementation (OSDI '12)*, pages 31 – 46, Hollywood, California, USA, 2012. USENIX Association.

[114] Nathan LaBelle and Eugene Wallingford. Inter-Package Dependency Networks in Open-Source Software. *CoRR*, November 2004.

[115] Arun Lakhotia. Rule-based approach to computing module cohesion. *Software Engineering, 1993. Proceedings., 15th*, pages 35–44, 1993.

[116] Arun Lakhotia and J.C. Deprez. Precise slices of block-structured programs with goto statements. 1997.

[117] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer, 2006. ISBN 3540244298.

[118] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. pages 495–505, May 1996.

[119] Jure Leskovec, Kevin J. Lang, and Michael Mahoney. Empirical comparison of algorithms for network community detection. In *Proceedings of the 19th international conference on World wide web - WWW '10*, page 631, New York, New York, USA, April 2010. ACM Press. ISBN 9781605587998. doi: 10.1145/1772690.1772755.

[120] F Liljeros, C R Edling, L A Amaral, H E Stanley, and Y Aberg. The web of human sexual contacts. *Nature*, 411(6840):907–8, June 2001. ISSN 0028-0836. doi: 10.1038/35082140.

[121] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '06*, page 872, New York, New York, USA, August 2006. ACM Press. ISBN 1595933395. doi: 10.1145/1150402.1150522.

[122] H Longworth. *Slice based program metrics.* Masters, Michigan Technological University, Houghton, Michigan, 1985.

[123] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Transactions on Software Engineering and Methodology*, 18 (1):1–26, September 2008. ISSN 1049331X. doi: 10.1145/1391984.1391986.

[124] J.P. Loyall and S.A. Mathisen. Using dependence analysis to support the software maintenance process. In *1993 Conference on Software Maintenance*, pages 282–291. IEEE Comput. Soc. Press, 1993. ISBN 0-8186-4600-4. doi: 10.1109/ICSM.1993.366934.

[125] R D Luce and A D Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, 1949.

[126] Yutao Ma, Keqing He, Dehui Du, Jing Liu, and Yulan Yan. A hybrid set of complexity metrics for large-scale object-oriented software systems. *Journal of Computer Science and Technology*, 25(6):1184–1201, 2010. doi: 10.1007/s11390-010-1094-3.

[127] Anirban Majumdar, SJ Drape, and C.D. Thomborson. Slicing obfuscations: design, correctness, and evaluation. In *Proceedings of the 2007 ACM workshop on Digital Rights Management*, pages 70–81, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-884-8.

[128] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organizations of source code. *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98*, pages 45–52, 1998. doi: 10.1109/WPC.1998.693283.

[129] S Mancoridis, B S Mitchell, Y Chen, and E R Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 50–59, 1999. doi: 10.1109/ICSM.1999.792498.

[130] Timothy M. Meyers and David Binkley. A longitudinal and comparative study of slice-based metrics. In *10th International Software Metrics Symposium*, Chicago, IL, USA, 2004.

[131] Timothy M. Meyers and David Binkley. An empirical study of slice-based cohesion and coupling metrics. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–27, December 2007. ISSN 1049331X. doi: 10.1145/1314493.1314495.

[132] T.M. Meyers and David Binkley. Slice-based cohesion metrics and software intervention. In *11th Working Conference on Reverse Engineering*, pages 256–

265. IEEE Computer Society, November 2004. ISBN 0-7695-2243-2. doi: 10.1109/WCRE.2004.34.

[133] BS Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):1–16, 2006.

[134] D.P. Mohapatra, Rajib Mall, and Rajeev Kumar. An overview of slicing techniques for object-oriented programs. *Informatica (Slovenia)*, 30(2):253, 2006.

[135] Durga Prasad Mohapatra, Rajib Mall, and Rajeev Kumar. Computing dynamic slices of concurrent object-oriented programs. *Information and Software Technology*, 47(12):805–817, September 2005. ISSN 09505849. doi: 10.1016/j.infsof.2005.02.002.

[136] G.B. Mund and Rajib Mall. An efficient interprocedural dynamic slicing method. *Journal of Systems and Software*, 79(6):791–806, June 2006. ISSN 01641212. doi: 10.1016/j.jss.2005.07.024.

[137] G.B Mund, R Mall, and S Sarkar. An efficient dynamic program slicing technique. *Information and Software Technology*, 44(2):123–132, February 2002. ISSN 09505849. doi: 10.1016/S0950-5849(01)00224-5.

[138] G.B. Mund, R. Mall, and S. Sarkar. Computation of intraprocedural dynamic program slices. *Information and Software Technology*, 45(8):499–512, June 2003. ISSN 09505849. doi: 10.1016/S0950-5849(03)00029-6.

[139] Christopher Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68 (4), October 2003. ISSN 1063-651X. doi: 10.1103/PhysRevE.68.046116.

[140] Mangala Gowri Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of the International Symposium on Software Testing and Analysis - ISSTA '00*, volume 25, pages 180–190, New York, New York, USA, August 2000. ACM Press. ISBN 1581132662. doi: 10.1145/347324.349121.

[141] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, March 2003.

[142] M E J Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences of the United States of America*, 103(23): 8577–82, June 2006. ISSN 0027-8424. doi: 10.1073/pnas.0601602103.

[143] M E J Newman and M Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):26113, 2004. ISSN 01678655. doi: 10.1016/j.patrec.2009.11.001.

[144] Mark Newman. *Networks: An Introduction*. OUP Oxford, Oxford, England, 1st edition, 2010. ISBN 0199206651.

[145] M.E.J. Newman. Analysis of weighted networks. *Physical Review E*, 70(5), 2004.

[146] Robert Weeks O'Callahan. *Generalized aliasing as a basis for program analysis tools*. PhD thesis, 2001.

[147] Fumiaki Ohata, Kouya Hirose, and Masato Fujii. A slicing method for object-oriented programs using lightweight dynamic information. pages 273–280, 2001.

[148] Linda M Ott and James M Bieman. Program slices as an abstraction for cohesion measurement. *Information and Software Technology Special Issue on Program Slicing*, 40(11-12):691–699, 1998. ISSN 09505849.

[149] L.M. Ott and J.J. Thuss. The Relationship Between Slices And Module Cohesion. In *11th International Conference on Software Engineering*, pages 198–204. IEEE, 1989. ISBN 0-8186-8941-2. doi: 10.1109/ICSE.1989.714420.

[150] L.M. Ott and J.J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings First International Software Metrics Symposium*, pages 71–81. IEEE Comput. Soc. Press, 1993. ISBN 0-8186-3740-4. doi: 10.1109/METRIC.1993.263799.

[151] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *ACM SIGSOFT Software Engineering Notes*, 9(3):177–184, May 1984. ISSN 01635948. doi: 10.1145/390010.808263.

[152] Kai Pan, Sunghun Kim, and James Whitehead. Bug Classification Using Program Slicing Metrics. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 31–42. IEEE, 2006. ISBN 0769523536. doi: 10.1109/SCAM.2006.6.

[153] Prashant Paymal, Rajvardhan Patil, Sanjukta Bhowmick, and Harvey Siy. Empirical Study of Software Evolution Using Community Detection. Technical report, University of Nebraska, Omaha, 2011.

[154] A. Podgurski and L.A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990. ISSN 00985589. doi: 10.1109/32.58784.

[155] Mason A. Porter, Jukka-Pekka Onnela, and Peter J. Mucha. Communities in Networks. *Notices of the American Mathematical Society*, 56(9), February 2009.

[156] Alex Potanin, James Noble, and Marcus Frean. Scale-free geometry in Object Oriented programs. *Communications of the ACM*, 48(5):1–8, 2002.

[157] Paul Pritchard. An Old Sub-Quadratic Algorithm for Finding Extremal Sets. Technical report, 1994.

[158] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences of the United States of America*, 101 (9):2658–63, March 2004. ISSN 0027-8424. doi: 10.1073/pnas.0400054101.

[159] Venkatesh Ranganath. *Object-Flow Analysis For Optimizing Finite-State Models of Java Software.* Masters, Kansas State University, 2002.

[160] Venkatesh Ranganath, John Hatcliff, and Evelyn Duesterwald. Pruning Interference and Ready Dependence for Slicing Concurrent Java Programs. *Lecture Notes in Computer Science*, 2985:2725, 2004. doi: 10.1007/b95956.

[161] Venkatesh Prasad 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, July 2007. ISSN 1433-2779. doi: 10.1007/s10009-007-0043-0.

[162] V.P. Ranganath. *Scalable and accurate approaches for program dependence analysis, slicing, and verification of concurrent object oriented programs.* Phd, KANSAS STATE UNIVERSITY, 2007.

[163] V.P. Ranganath. Indus - Java Program Slicer, 2009. URL `http://indus.projects.cis.ksu.edu`.

[164] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes*, 19(5):11–20, December 1994. ISSN 01635948. doi: 10.1145/195274.195287.

[165] Sebastian Schnettler. A structured overview of 50 years of small-world re-

search. *Social Networks*, 31(3):165–178, July 2009. ISSN 03788733. doi: 10.1016/j.socnet.2008.12.004.

[166] M. A Siddiqui. *Data Mining Methods for Malware Detection.* PhD thesis, Florida, 2008.

[167] Chaoming Song, Shlomo Havlin, and Hernán A Makse. Self-similarity of complex networks. *Nature*, 433(7024):392–5, January 2005. ISSN 1476-4687. doi: 10.1038/nature03248.

[168] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and G.L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43–60, January 2002. ISSN 1350-1917. doi: 10.1046/j.1365-2575.2002.00117.x.

[169] Christoph Steindl. Intermodular Slicing of Object-Oriented Programs. In Kai Koskimies, editor, *7th International Conference, CC'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28 – April 4, 1998 Proceedings*, volume 1383 of *Lecture Notes in Computer Science*, pages 264–278. Springer Berlin Heidelberg, 1998. ISBN 3540654607. doi: 10.1007/BFb0026437.

[170] Ricky E Sward, AT Chamillard, and D Cook. Using Software Metrics and Program Slicing for Refactoring. *The Journal of Defense Software Engineering*, (July), 2004.

[171] Attila Szegedi and Tibor Gyimothy. Dynamic slicing of Java bytecode programs. In *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on*, volume 0, pages 35–44, Los Alamitos, CA, USA, 2005. IEEE. ISBN 0769522920. doi: 10.1109/SCAM.2005.8.

[172] P. Tonella. Using a concept lattice of decomposition slices for program under-

standing and impact analysis. *IEEE Transactions on Software Engineering*, 29(6):495–509, June 2003. ISSN 0098-5589. doi: 10.1109/TSE.2003.1205178.

[173] Fumiaki Umemori, Kenji Konda, Reishi Yokomori, and Katsuro Inoue. Design and Implementation of Bytecode-based Java Slicing System. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:108, 2003.

[174] K. Ushijima. Static slicing of concurrent object-oriented programs. In *Proceedings of 20th International Computer Software and Applications Conference: COMPSAC '96*, pages 312–320. IEEE Comput. Soc. Press. ISBN 0-8186-7579-9. doi: 10.1109/CMPSAC.1996.544182.

[175] R. Vallee-Rai and L.J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University, Montreal, Quebec, Canada, 1998.

[176] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - A Java Bytecode Optimization Framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[177] Raja Vallee-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? pages 18–34, 2000.

[178] S Valverde and R. V Solé. Logarithmic growth dynamics in software networks. *Europhysics Letters (EPL)*, 72(5):858–864, December 2005. ISSN 0295-5075. doi: 10.1209/epl/i2005-10314-9.

[179] R. van der Hofstad. Random graphs and complex networks. Technical report, Eindhoven University of Technology, 2008.

[180] Lovro Šubelj and Marko Bajec. Community structure of complex software systems: Analysis and applications. *Physica A: Statistical Mechanics and its Applications*, 390(16):1–12, August 2011. ISSN 03784371. doi: 10.1016/j.physa.2011.03.036.

[181] Lei Wang, Zheng Wang, Chen Yang, Li Zhang, and Qiang Ye. Linux kernels as complex networks: A novel method to study evolution. In *2009 IEEE International Conference on Software Maintenance*, pages 41–50. IEEE, September 2009. ISBN 978-1-4244-4897-5. doi: 10.1109/ICSM.2009.5306348.

[182] Tao Wang and Abhik Roychoudhury. Using Compressed Bytecode Traces for Slicing Java Programs. *Software Engineering, International Conference on*, 0:512–521, 2004. ISSN 0270-5257.

[183] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications (Structural Analysis in the Social Sciences).* Cambridge University Press, 1994. ISBN 0521387078.

[184] D J Watts and S H Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–2, June 1998. ISSN 0028-0836. doi: 10.1038/30918.

[185] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0897911466.

[186] Mark D Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* Phd, University of Michigan, Ann Arbor, 1979.

[187] Lian Wen, D Kirk, and R G Dromey. Software Systems as Complex Networks. In *Cognitive Informatics 6th IEEE International Conference on*, pages 106–115. Ieee, 2007. doi: 10.1109/COGINF.2007.4341879.

[188] Richard Wheeldon and Steve Counsell. Power Law Distributions in Class Relationships. In *In Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '03)*, pages 45–54. IEEE, May 2003.

[189] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005. ISSN 0163-5948. doi: 10.1145/1050849.1050865.

[190] Guoai Xu, Yang Gao, Fanfan Liu, Aiguo Chen, and Miao Zhang. Statistical Analysis of Software Coupling Measurement Based on Complex Networks. *2008 International Seminar on Future Information Technology and Management Engineering*, pages 577–581, 2008. doi: 10.1109/FITME.2008.62.

[191] Vyacheslav Yakovlev. *Cluster Analysis of Object-Oriented Programs.* Masters, Massey University, Palmerston North, New Zealand, 2009.

[192] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, February 1979. ISBN 0138544719.

[193] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. *ACM SIGPLAN Notices*, 39(6):94, June 2004. ISSN 03621340. doi: 10.1145/996893.996855.

[194] Jianjun Zhao. Using Dependence Analysis to Support Software Architecture Understanding. *New Technologies on Computer Software*, pages 135–142, May 1997.

[195] Jianjun Zhao. Slicing concurrent Java programs. In *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*, pages 126–133. IEEE, 1999.

[196] Jianjun Zhao. Dynamic Slicing of Object-Oriented Programs. *Wuhan University Journal of Natural Sciences*, 6(1-2):391–397, 2001. doi: 10.1007/BF03160274.

Index