Coherent Dependence Clusters

Syed Islam† Jens Krinke† David Binkley* Mark Harman†
†Centre for Research on Evolution, Search and Testing (CREST)
King's College London,
London, UK
*Loyola University in Maryland
Baltimore Maryland, USA
{syed.s.islam,jens.krinke,mark.harman}@kcl.ac.uk, binkley@cs.loyola.edu

ABSTRACT

Large clusters of mutual dependence can cause problems for comprehension, testing and maintenance. This paper introduces the concept of *coherent dependence clusters*, techniques for their efficient identification, visualizations to better understand them, and empirical results concerning their practical significance. As the paper will show, coherent dependence clusters facilitate a fine grained analysis of the subtle relationships between clusters of dependence.

1. INTRODUCTION

Program dependence analysis, a key component of source code analysis [4], explores the dependence relationships between program statements. Real-world code can contain large clusters of mutually dependent code [7, 15], which can be hard to comprehend [10], modify [14], test [3], and analyze [12]. As such, dependence clusters can be regarded as anti-patterns [5] or bad code smells [13].

Despite their potentially large negative impact, dependence clusters are not well understood. One could be forgiven for thinking that there is little to understand; at first sight, there appears to be little more to say than 'everything depends on everything else'. However, interprocedural dependence is *non-transitive* with the result that the relationships between dependence clusters can be highly subtle, even surprising. This motivates the introduction and study of coherent dependence clusters; dependence clusters within which all nodes share identical extra-cluster dependence.

The paper presents the results of an empirical study of coherent dependence clusters in eight open-source programs. The results show that the probability of finding large coherent dependence clusters in real-world programs is high, thereby motivating their further study. The results also reveal that, in most cases, large coherent dependence clusters are formed by partitioning even larger dependence clusters.

The paper also introduces two new visualizations: the Monotone Cluster-Size Graph and the Slice/Cluster-Size Graph. These two visualizations facilitate exploration and better understanding of the size and prevalence of coherent dependence clusters and the relationships between them. The paper applies these visualizations to the eight programs from the empirical study, illustrating their use with several case studies.

The remainder of this paper is organized as follows: Section 2 provides background on dependence clusters and previous visualization technique, while Section 3 introduces coherent dependence clusters. Section 4 describes the newly proposed visualization techniques and Section 5 presents the results of the empirical study. Section 6 describes related work, while Section 7 shows results of ongoing work as well as a glimpse into future work, and finally, Section 8 summarizes the work presented.

2. BACKGROUND

This section first provides a general definition of mutually dependent sets and dependence clusters. It then discusses existing techniques for detecting and visualizing dependence clusters. Finally, it illustrates the intransitivity of the dependence relation used in forming dependence clusters and discusses some implications.

Harman et al. [15] defined a dependence cluster as a maximal set of program statements that mutually depend upon one another. This notion is formalized in the following two definitions:

Definition 1 (Mutually-Dependent Set [15])
A *Mutually-Dependent Set (MDS)* is a set of statements, S, such that

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

Definition 2 (Mutual-Dependence Cluster [15])
A mutual-dependence cluster is simply a maximal set of mutually dependent statements. That is, a *Mutual-Dependence Cluster* is an MDS not properly contained within any other MDS.

Previous work used the more general term *Dependence Cluster* in the above definition. The more concise term *Mutual-Dependence Cluster* is used here to emphasize that such clusters only consider internal (mutual) dependence and thus to distinguish them from those that also consider external dependence relations.

The above definitions are parameterized by an underlying *depends-on* relation. One approach to capturing this relation, which leads to several useful approximations, is based on Weiser's *Program Slice* [21]: a slice is a set of program statements that affect the values computed at a statement of interest (referred to as the slicing criterion). While its computation is undecidable, a minimal (or precise) slice includes exactly those program elements that affect the criterion and thus can be used to provide an equivalent definition for an MDS in which t depends on s iff s is in the minimal slice taken with respect to slicing criterion t.

The slice-based definition is useful because algorithms to compute (approximations to minimal) slices can be used to define and

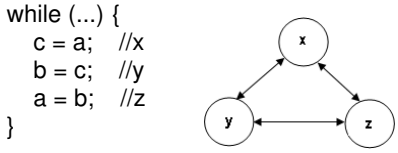


Figure 1: A Slice-based Cluster

compute approximations to mutual-dependence clusters. One such algorithm computes slices using a program's System Dependence Graph (SDG) [16]. An SDG is comprised of vertices, which essentially represent the statements of the program, and edges, which represent the immediate control and data dependence between vertices. Hereafter, the term *statement* is taken to be synonymous with the statement's SDG vertex. A control dependence arises when one statement controls the possible execution of another statement and a data dependence arises when a value flows from a defining statement to a use reached by the definition. In the SDG a slicing criterion is a vertex from the SDG.

Two kinds of SDG slices are used in this paper: backward slices and forward slices. The backward slice taken with respect to vertex t, denoted $\mathsf{BSlice}(t)$, is the set of vertices reaching t via a path of control and data dependence edges [19]. The second kind of slice, a forward slice, is also taken with respect to vertex t. Denoted $\mathsf{FSlice}(t)$, it includes the set of vertices reachable from t via a path of control and data dependence edges [16]. In both cases, when slicing programs that contain certain language features, the path of dependence edges considered must be restricted. For example to respect procedure calling convention of the language [16].

The following definitions are given using BSlice. Each has a dual that uses FSlice. When the distinction is important, *backward* and *forward* will be used for clarification. Harman et al. [15] use backward slicing to define *slice-based* clusters:

Definition 3 (Slice-based MDS/Cluster [15])
A *Slice-based MDS* is a set of statements, S, such that
$\forall x, y \in S : x \in \mathsf{BSlice}(y)$.
A *Slice-based Cluster* is a slice-based MDS contained within no other slice-based MDS.

An example of a slice-based cluster is shown in Figure 1, which includes a fragment of source code on the left and a graphical depiction of its slice-based dependence (slice containment relations) on the right. In this graph, Statements x, y, and z are represented by nodes and the directional edges denote slice-based dependence relationships: $a \rightarrow b$ depicts that b depends on a or equivalently that $a \in \mathsf{BSlice}(b)$. In the example, $\mathsf{BSlice}(\mathsf{y})$ includes x and thus y depends on x as captured by the edge $\mathsf{x} \rightarrow \mathsf{y}$. Because all three statements are in each other's slices, they are mutually dependent. Therefore, the set $\{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$ forms a slice-based MDS. Assuming that the set is included in no larger cluster, the set also satisfies the definition of a slice-based cluster.

Calculating the complete set of slice-based clusters is expensive because it requires computing and comparing the slices for every pair of statements in a program. This process requires quadratic space and time and thus, even for mid-sized programs, can grow prohibitively expensive. To reduce this cost, Binkley and Harman [7] approximate the mutual-slice inclusion of Definition 3 using the *same-slice* relation. This technique replaces checking if two vertices are in each other's slice with checking if two vertices have the *same* slice. This notion is formalized as follows

Definition 4 (Same-Slice MDS/Cluster [7])
A *Same-Slice MDS* is a set of statements, S, such that
$\forall x, y \in S : \mathsf{BSlice}(x) = \mathsf{BSlice}(y)$.
A *Same-Slice Cluster* is a Same-Slice MDS contained within no other Same-Slice MDS.

Because $\mathsf{BSlice}(x)$ always includes x, two vertices that have the same slice will be in each other's slice. If slice-inclusion were transitive, then the Slice-based MDS (Definition 3) would be identical to the Same-Slice MDS (Definition 4). However, slice-inclusion is not transitive (as illustrated later in this section); thus, the relation is one of containment where every Same-Slice MDS is a Slice-based MDS but not necessarily a maximal one.

Although the introduction of same-slice clusters was motivated by the need for efficiency, the definition inadvertently introduced an *external* requirement on the cluster. This addition becomes important as it was later exploited in the development of coherent dependence clusters. Comparing the definitions for a Slice-based Cluster (Definition 3) and a Same-Slice Cluster (Definition 4), a Slice-based cluster includes only an *internal* requirement on the elements of a cluster. In contrast, a Same-Slice Cluster includes this same internal requirement, but adds the external requirement that all statements in the cluster are affected by the same statements external to the cluster. In the next section, coherent dependence clusters will extend the external requirements to include the statements affected by the elements of the cluster.

Even calculating same-slice clusters is expensive. In practice it requires tens of gigabytes of memory for even modest sized programs [15]. Thus, a second approximation was used. This approximation replaces 'same-slice' with 'same-slice-size': rather than checking if two vertices yield identical slices, the approach simply checks if the two vertices yield slices of the same size. The resulting *same-slice-size* approach is formalized as follows:

Definition 5 (Same-Slice-Size MDS/Cluster [15])
A *Same-Slice-Size MDS* is a set of statements, S, such that
$\forall x, y \in S : |\mathsf{BSlice}(x)| = |\mathsf{BSlice}(y)|$.
A *Same-Slice-Size Cluster* is a Same-Slice-Size MDS contained within no other Same-Slice-Size MDS.

The observation motivating this approximation is that two slices of the same (large) size are likely to be the same slice. In practice, this approximation is very accurate if a small tolerance for difference is allowed. This tolerance is needed when two slices of the same size have *almost* the same vertices. For example, one situation in which this commonly occurs is when a call-site is in a cluster. The slice taken with respect to each of the call's actual parameters often includes the parameter and then the same vertices as the slice taken with respect to the call-site itself. Thus, the slice size for each slice taken with respect to a parameter is the same, but these slices differ by the single vertex representing the parameter. With a tolerance of 1% the approximation is 99.9943% accurate. However, in the strict case of *zero* tolerance the accuracy falls to 78.3%.

The same-slice-size approximation also leads to a useful visualization: *Monotone Slice-Size Graph* (MSG) [7]. An MSG plots a landscape of monotonically increasing slice sizes where the x-axis includes each slice, in increasing order, and the y-axis shows the size of each slice, as a percentage of the entire program. In an MSG a dependence cluster appears as a sheer-drop cliff face followed by a plateau. The visualization assists with the inherently subjective task of deciding whether a cluster is large (how long is the plateau at the top of the cliff face relative to the surrounding landscape?) and whether it denotes a discontinuity in the dependence profile (how steep is the cliff face relative to the surrounding landscape?). MSGs drawn using backward slice sizes are referred to as backward-slice MSG (B-MSG), which those using forward slice sizes are referred to forward-slice MSG (F-MSG).

Example. The open source calculator bc contains 16,763 lines of code represented by 7,538 SDG vertices. The MSG for bc, shown in Figure 2, contains a large plateau spanning almost 70% of the MSG. This indicates a same-slice-size cluster formed from slices that appear to have the same size. However, "zooming" in reveals that the cluster is actually composed of multiple smaller clusters made from slices of very similar size. The tolerance implicit in the visual resolution used to plot the MSG obscures this detail. Two alternative visualizations that both overcome this implicit tolerance are presented in Section 4.

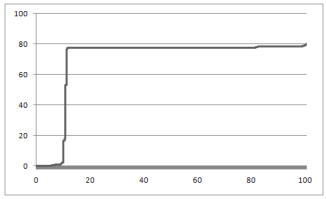


Figure 2: Monotone Slice-Size Graph (MSG) for bc

The remainder of this section considers the impact of slice inclusion not being a transitive relation. For programs with features such as multiple threads or multiple procedures, $x \in \mathsf{BSlice}(y)$ and $y \in \mathsf{BSlice}(z)$, does not imply $x \in \mathsf{BSlice}(z)$. This is illustrated by Program P shown in the first column of Table 1. Consider the Statements labeled a, b, and c. Columns 3, 4, and 5 show the slices of P taken with respect to a, b and c, respectively. From these slices it can be seen that c depends on b (b ∈ BSlice(c)) and b depends on a (a ∈ BSlice(b)); however, c does not depend on a (a ∉ BSlice(c)). This example illustrates that slice inclusion is not a transitive relation.

Program P	Lbl	P sliced on a	P sliced on b	P sliced on c

void f1() {
  x = 1;
  y = f3(x);
}
	a	void f1() {
		  x = 1;
		
		}
			void f1() {
			  x = 1;
			  y = f3(x);
			}	

void f2() {
  x = 2;
  y = f3(x);
  return y;
}
	c		void f2() {
			  x = 2;
			  y = f3(x);
			
			}
				void f2() {
				  x = 2;
				  y = f3(x);
				  return y;
				}

int f3(int z) {
  return z+1;
}
	b		int f3(int z) {
			  return z+1;
			}
				int f3(int z) {
				  return z+1;
				}

Table 1: Intransitive Dependence

One implication of the lack of transitivity is that a statement can be in multiple clusters. For example, in Figure 3 Statements i, j, and k are mutually dependent upon each other as are Statements i, j, and l. However, Statements k and l do not have a dependence relationship (i.e., are not part of each other's slices). The slice-based clusters (Definition 3) formed from the nodes of this graph are $\{\mathsf{i},\mathsf{j},\mathsf{l}\}$ and $\{\mathsf{i},\mathsf{j},\mathsf{k}\}$. Notice that Statements i and j are in both clusters. In contrast, the same-slice clusters (and the same-slice-size clusters) are $\{\mathsf{k}\}, \{\mathsf{i},\mathsf{j}\}, \{\mathsf{l}\}$

Rather than calculating all the slice-based clusters for a program allowing overlaps (creating larger clusters), it is more interesting to find maximal partitions that do not include overlaps (creating smaller clusters) because such partitions model the various components of a program. This motivates the introduction of *coherent dependence clusters*.

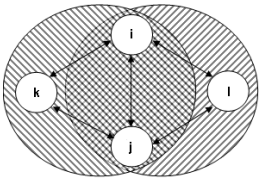


Figure 3: Overlapping Dependence Clusters

3. COHERENT DEPENDENCE CLUSTER

This section first formalizes *coherent dependence clusters*, and then presents a slice-based instantiation of the definition for coherent (dependence) clusters. 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. Coherent dependence clusters extend dependence clusters to include not only internal dependence (each statements of a cluster must depend on all the other statements of the cluster) but also external dependence. External dependence includes both that each statement of a cluster depends on the same external statements and that the same set of external statements depends on each statement of the cluster. The first external dependence (that each statement of a cluster depends on the same external statements) was inadvertently introduced by Harman et al. [15] in the same-slice approximation. The second external dependence is new. Incorporating internal and both kinds of external dependence, Coherent Clusters are defined in terms of the coherent MDS:

Definition 6 (Coherent MDS/Cluster)
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.

A slice-based instantiation of coherent clusters is now considered. Unlike the definitions presented in Section 2 these new definitions employ both backward *and* forward slices. The combination has the advantage that the entire cluster is both affected by the same set of statements (as in the case of same-backward-slice clusters) and also affects the same set of statements (as in the case of same-forward-slice clusters). The slice-based instantiation produces the *Coherent-Slice Cluster*:

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

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

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

At first glance the use of both backward and forward slices might seem redundant because $x \in \mathsf{BSlice}(y) \Leftrightarrow y \in \mathsf{FSlice}(x)$. This is true up to a point; as for the internal requirement of a coherent-slice cluster, the use of either BSlice or FSlice would be sufficient. However, the two are not redundant for the external requirements of a coherent-slice cluster. With a mutual-dependence cluster (as defined by Definition 2) it is possible for two statements within the cluster to affect or be affected by different statements *external* to the cluster. Neither is allowed with a coherent-slice cluster. To ensure both external effects are captured, both forward and backward slices are required.

The example shown in Figure 4 illustrates the differences between same-backward-slice clusters, same-forward-slice clusters, and coherent-slice clusters. The illustration includes six nodes: a, b, c, d, x, and y. As can be seen in the graph, d depends only on c, only b depends on a, and b, c, x and y are mutually dependent upon (completely interconnected with) each other and, hence form a Mutual-Dependence Cluster (Definition 2). However, this (non-coherent) cluster includes statements that have different influences and are influenced by different statements. This is easily seen by looking at the slices shown in Table 2. Here, c, x, and y form a same-backward-slice cluster and are thus affected by the same external statements. Furthermore, b, x, and y form a same-forward-slice cluster as they affect the same external statements. However, only the coherent-slice cluster of x and y incorporates both influences.

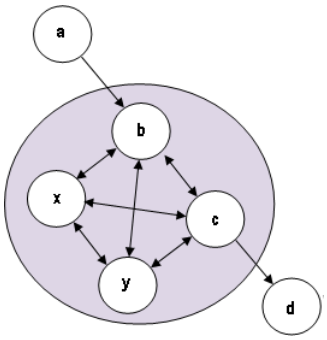


Figure 4: Slice Inclusion Example

Slice Criterion	Backward Slice	Forward Slice
a	{a}	{a,b}
b	{a, x, y, b, c}	{x, y, b, c}
c	{x, y, b, c}	{x, y, b, c, d}
d	{c, d}	{d}
x	{x, y, b, c}	{x, y, b, c}
y	{x, y, b, c}	{x, y, b, c}

Table 2: Slices for Figure 4

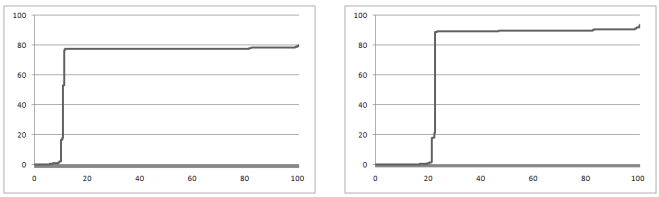
The computation of the coherent-slice clusters (Definition 7) requires considerable computational effort. Therefore, an approximation similar to the same-slice-size approach is employed. This approximation replaces slice comparisons with the comparison of a hash value, yielding the following approximation to Coherent-Slice Clusters in which H denotes the hash function. A *Hash-Based Coherent-Slice MDS* is a set of statements, S, such that

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

A *Hash-Based Coherent-Slice Cluster* is a Hash-Based Coherent-Slice MDS properly contained within no other Hash-Based Coherent-Slice MDS.

4. VISUALIZATION

This section introduces two new visualizations: the *Monotone Cluster-Size Graph* (MCG) and the *Slice/Cluster-Size Graph* (SCG). Both provide greater visual precision than the MSG. While the visual blurring in the MSG is at times an advantage, it can also preclude the precise identification of dependence clusters. These two visualizations are empirically studied in the Section 5.



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

Figure 5: MSGs for the program bc.

4.1 Monotone Cluster-Size Graph

The *Monotone Cluster-Size Graph* (MCG) visualizes clusters based on cluster size rather than slice size as done in the MSG:

Definition 8 (Monotone Cluster-Size Graph)
A *Monotone Cluster-Size Graph (MCG)* is a graph of cluster sizes, plotted for monotonically increasing size.

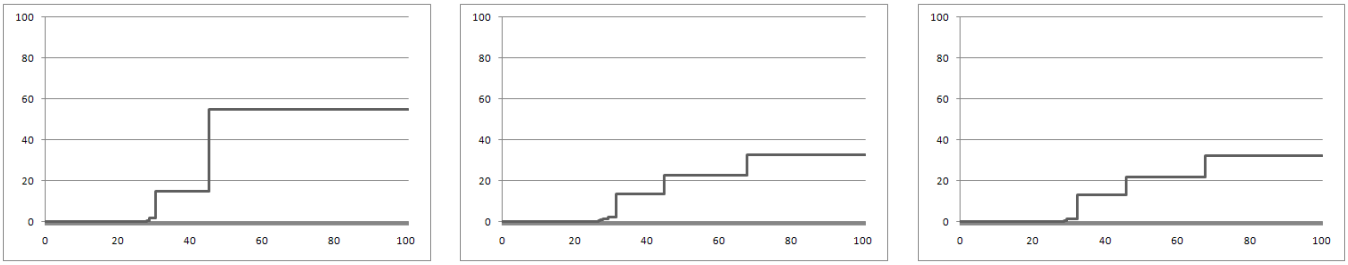
Thus in an MCG, cluster sizes (measured in vertices) are plotted on the horizontal axis in monotonically increasing order with their sizes plotted on the vertical axis. As a result, a program's (same-slice) clusters are clearly identified as steps in the MCG.

An MCG can be drawn using the sizes of same-backward-slice clusters (B-MCG), same-forward-slice clusters (F-MCG), or coherent-slice clusters (C-MCG). Figure 6 shows these three MCGs for the program bc. Comparing the B-MSG from Figure 5a and the B-MCG from Figure 6a the precision difference becomes apparent. In the MSG the long plateau indicates that approximately 70% of the program is involved in a cluster. In contrast, the B-MCG clearly shows that this long plateau is composed of two separate clusters that span 15% and 55% of the program. This observation is supported by the raw slice data where the two same-backward-slice clusters cover 14.74% and 54.86% of the program. The F-MCG for bc (Figure 6b) shows three distinct steps depicting three same-forward-slice clusters. Finally, similar to the F-MCG, in the C-MCG, the presence of three coherent-slice clusters spanning about 15%, 20% and 30% of the program's statements can be seen. As coherent-slice clusters are formed using both backward and forward slices, they will tend to closely resemble the smaller of the same-backward/forward-slice clusters.

4.2 Slice/Cluster-Size Graphs

As illustrated by Figures 5 and 6, the MCG provides an accurate visualization of the clusters. However, information regarding the size of the slices that form the cluster is no longer available. The *Slice/Cluster-Size Graphs (SCGs)* provides a link between slice and cluster sizes. Two variants of the SCG are used: the backward-slice SCG (B-SCG) is built from the sizes of backward slices, same-backward-slice clusters, and coherent-slice clusters, while the forward-slice SCG (F-SCG) is build from the sizes of forward slices, same-forward-slice clusters, and coherent-slice clusters. Note that both backward and forward SCGs use the same coherent-slice cluster sizes.

An SCG plots three landscapes, one for increasing slice sizes, one for the corresponding same-slice cluster sizes, and the third for the corresponding coherent-slice cluster sizes. As a result, this visualization not only shows the slices with similar sizes but also distinctly identifies which clusters are formed. In the SCG, vertices are ordered along the x-axis first according to their slice size, second according to their same-slice cluster size, and third according to the coherent-slice cluster size. Three values are plotted on the y-axis: slice sizes form the first landscape, while cluster sizes form



(a) B-MCG (b) F-MCG (c) C-MCG

Figure 6: MCGs for the program bc.

the second and third. Thus, SCGs not only show the sizes of the slices and the clusters, they also show the relation between them.

The B-SCG and F-SCG for the program bc are shown in the second row of Figure 8. In both figures the slice size landscape is plotted using a solid black line, the same-slice cluster size landscape using a gray line, and the coherent-slice cluster size landscape using a broken line. Like the B-MCG, the B-SCG shows that bc contains a large same-backward-slice cluster consisting of almost 55% of the program and a second consisting of almost 15% of the program. In contrast to the MCG, the size of the slices that formed the clusters can be ascertained. In fact, Figure 6a gives the impression that the larger cluster is of greater interest because it includes more program statements. However, in the left SCG on second row of Figure 8 the larger cluster is composed of smaller slices than the smaller cluster. The smaller cluster thus has a bigger impact (slice size) than the larger cluster.

Finally, three interesting observations can be made from considering bc's two SCGs. First, the program bc contains two large same-backward-slice cluster as opposed to three large same-forward-slice clusters visible in the light gray landscapes. Secondly, looking at the B-SCG it can be seen that the space corresponding to the largest same-backward-slice cluster is occupied by two coherent-slice clusters (shown in dotted landscape). This indicates that the same-backward-slice cluster splits into the two coherent-slice clusters; a phenomenon found to be common and studied in Section 5. Finally, coherent-slice clusters are almost identical to the same-forward-slice clusters. Since the same-forward-slice clusters for the program bc were smaller in size than the same-backward-slice clusters, the coherent-slice clusters were essentially restricted to resemble the size of the same-forward-slice clusters.

5. EMPIRICAL VALIDATION

This section presents an empirical evaluation of the coherent-slice clusters and the two visualizations. It first considers the experimental setup and the subject programs used in the study. The core of the study includes results from the search for coherent-slice clusters and a case study that consider split clusters. Finally, threats to validity are considered.

5.1 Experimental Setup

The data for the empirical study was computed from the forward and backward slices taken with respect to each source-code-representing SDG vertex. This excludes pseudo vertices introduced into the SDG, to represent, for example, global variables, which are modeled as additional pseudo parameters by CodeSurfer [1], the tool used to build the SDGs. The hash values for each of the slices were stored and compared to calculate the clusters.

5.2 Experiment Subjects

Initially 18 open-source C programs were analyzed. Data from eight representative examples is presented in this section. Table 3 provides a brief description of the selected programs alongside two measures of each program's size: LoC – lines of code (as counted by the Unix utility wc) and SLoC – the non-comment non-blank lines of code (as counted by the utility sloc [22]). It also shows the number of slices that were calculated for each program.

Program	LoC	SLoC	Slices	Description
acct	10,182	6,764	1,417	Process monitoring
barcode	5,926	3,975	4,801	Barcode generator
bc	16,763	11,173	7,538	Calculator
diffutils	19,811	12,705	8,061	File differencing
ed	13,579	9,046	5,688	Line text editor
indent	6,724	4,834	6,222	Text formatter
userv	8,009	6,132	2,784	Access control
which	5,407	3,618	1,902	Unix utility

Table 3: Subject Programs Studied

5.3 Do Large Coherent-Slice Clusters occur in practice?

The evaluation focuses on the existence question "do large coherent-slice clusters exist?" This is done first visually and then quantitatively. The visual study considers the B-SCG and F-SCG for each subject program. These are shown in Figures 7 and 8. The graphs in the figures have been laid out in descending order of the largest coherent-slice cluster present in the program; that is, barcode has the largest coherent-slice clusters while the acct has the smallest.

The graphs shown in Figures 7 and 8 clearly contain large mutual and coherent dependence clusters whose sizes often resemble each other's closely. In particular for the program barcode all the clusters have an almost identical size. The programs bc and ed show instances where the sizes of the coherent-slice clusters closely resemble (i.e., are constrained by) the sizes of the same-forward-slice clusters but not that of the same-backward-slice clusters. On the contrary, the programs indent and diffutils show instances where the sizes of the coherent-slice cluster are constrained by the backward-same-slice clusters. The SCGs clearly identify coherent-slice clusters that occur in programs and their relation to the same-slice clusters and the actual slice sizes.

Turning to the quantitative evaluation, to quantitatively assess if a program includes a large coherent-slice cluster, there is a value judgment to be made concerning what constitutes large. For the purpose of the empirical study, a threshold of 10% is used. In other words, a program is said to contain a large coherent-slice cluster if 10% of the program (10% of its SDG's vertices) produce the

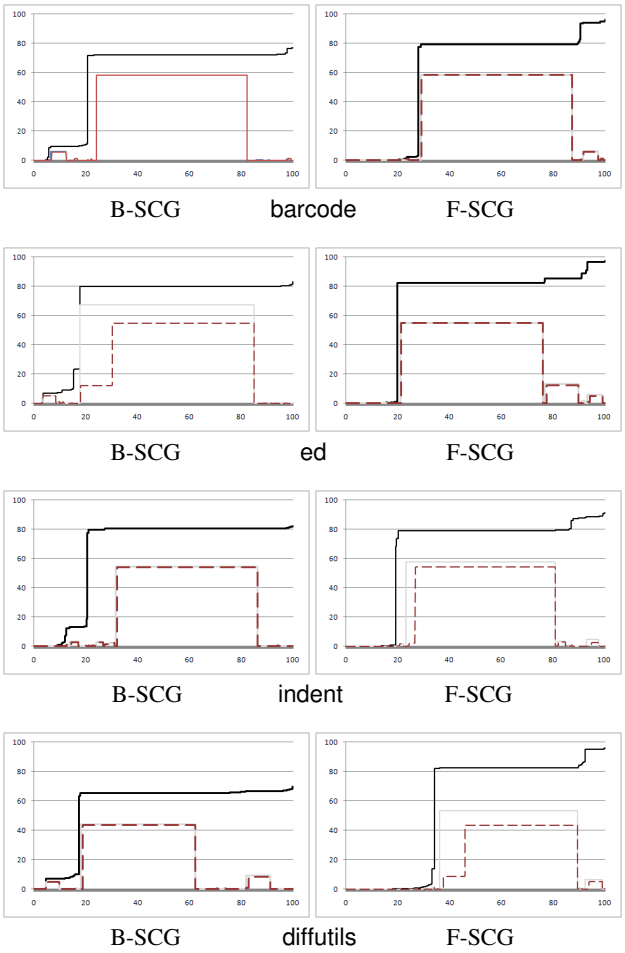


Figure 7: Slice/Cluster-Size Graphs (SCGs) 1-4

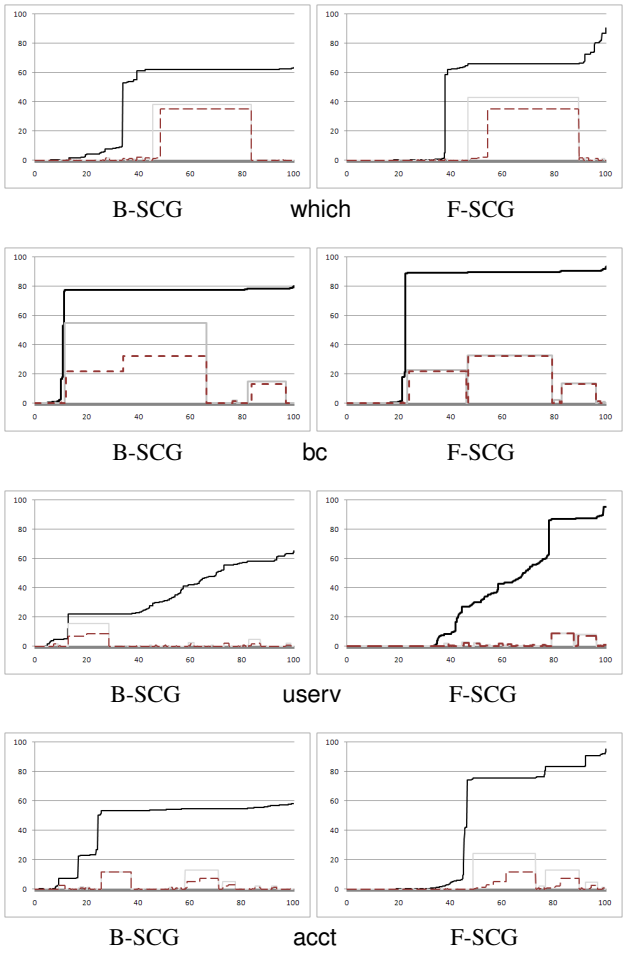


Figure 8: Slice/Cluster-Size Graphs (SCGs) 5-8

same backward slice and the same forward slice. This threshold matches that used in previous work [7, 15]. The choice of a relatively large threshold also provides a conservative answer to the existence question. However, smaller coherent-slice clusters may also be interesting and worthy of further investigation. Therefore, the results presented in this section can be thought of as a lower bound on the quantity of large coherent-slice clusters found in the programs studied.

Figure 9 shows the size of the largest coherent-slice cluster for each of the eight subject programs. These sizes are reported as a percentage of the program to facilitate comparison. Only *one* of the eight programs (userv) does not contain a large coherent-slice cluster using the 10% threshold. Of the remaining programs, three (barcode, ed and indent) have very large coherent-slice clusters that include over 50% of the program. In fact, the B-SCG for ed shown in Figure 7, includes two separate large coherent-slice clusters that cover over 10% of the program. (One is larger than 50% and the second covers just over 10% of the program). The programs diffutils, which and bc are seen to contain a coherent-slice cluster that covers over 25% of the program's SDG vertices.

The results of the empirical study show that large coherent-slice clusters are found frequently in real-world programs. These clusters are significant as any change made to such cluster or a statement that impacts a member of the cluster impacts the entire cluster and also all statements influenced by any member of the cluster. This can pose severe problems during program maintenance.

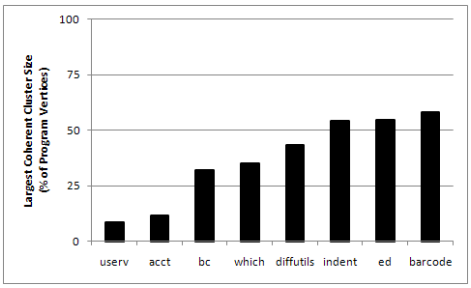


Figure 9: Coherent-Slice Clusters (Largest)

5.4 Case Study: Split Clusters

Because coherent-slice clusters incorporate both backward and forward slices, they potentially *split* same-backward-slice clusters and same-forward-slice clusters. This splitting can be seen visually in the left SCG on the second row of Figure 8, which includes a large same-backward-slice cluster (the gray landscape) that runs from 10% to 65% on the horizontal axis. The statements that make up this same-backward-slice cluster break in two coherent-slice clusters (the dashed landscape): the first runs from 10% to 35% and the second from 35% to 65%. Since these two coherent-slice clusters comprise the same statements (the same segment of the x-axis) they represent a splitting of the single same-backward-slice cluster.

As shown in Table 4, over all eight programs, splitting was found to be quite common. The table shows the number of same-backward-slice and same-forward-slice clusters that were split into at least two coherent-slice clusters. All but one of the programs had same-backward-slice or same-forward-slice cluster split. The exception was barcode where no splitting was observed. Same-backward-slice cluster splits were found to be more common, which can be partially attributed to the fact that same-backward-slice clusters are larger than same-forward-slice clusters just as backward slices are larger than forward slices [6].

	Cluster Splits	
Program	Backward-Slice	Forward-Slice
acct	1	1
barcode	0	0
bc	2	0
diffutils	1	1
ed	1	1
indent	0	1
userv	1	0
which	1	1
Total	7	5

Table 4: Split Cluster Distribution

5.5 Threats to validity

This section considers threats to the validity of the results presented in the paper. In the study, the primary external threat arises from the possibility that the selected programs are not representative of programs in general (i.e., the findings of the experiments do not apply to 'typical' programs). This is a reasonable concern that applies to any study of program properties. To address this issue a set of 18 open source programs were used as test subjects. The programs were not selected based on any criteria or property and thus represent a random selection. However, these were from the set of programs that were studied in previous work on dependence clusters (to facilitate comparison with previous results). In addition, all of the programs studied were C programs, so there is greater uncertainty that the results will hold for other programming paradigms such as object-oriented or aspect-oriented.

Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variables on the dependent variable. In this experiment, one possible threat arises from the potential for faults in the slicer. A mature and widely used slicing tool (CodeSurfer) was used to mitigate this concern. Another possible concern regards the precision of the pointer analysis used. An overly conservative, and therefore imprecise, analysis would tend to increase the levels of dependence and potentially also increase the size of clusters. There is no automatic way to tell whether a cluster arises because of imprecision in the computation of dependence or whether it is 'real'. CodeSurfer's most precise points-to analysis options were used for the study in order to address this potential concern.

The last threat is the use of a hash function to compare slices during the calculation of hash-based coherent-slice clusters. This hash function has been carefully crafted to minimize the hash collisions.

6. RELATED WORK

There has been much work concerned with clustering of one form or another; however, Binkley and Harman [7] were the first to introduce the notion of dependence clusters that looked into the fine grained structure of clustering based on vertices of an SDG rather than clustering of larger entities such as modules. They also showed how slicing could be used to detect such clusters and introduced the MSG visualization. Their work closely followed the work by Balmas [2] who used dependence analysis as a part of a visualization to assist with comprehension and maintenance activities. Harman et al. [15] extended this work presenting additional evidence to support the claim that large clusters are prevalent and exploring the implications for impact analysis. Secondly, there is relevant work on dependence anti-patterns [5]. A Dependence 'Anti-Pattern' is a dependence structure that may indicate potential problems for on-going software maintenance and evolution. Dependence anti-patterns are not structures that must always be avoided. Rather, they denote warnings that should be investigated. Dependence clusters were deemed to be one of the patterns that can potentially cause an increase in effort required for comprehension and testing due to mutual dependence of statements in a cluster. Black [11] hypothesized that faults are more likely to occur in source code that is highly interdependent and thus suggest a direct relationship between the presence of large dependence clusters to the number of faults in the program.

Further work by Binkley et al. [9] considered one of the causes of dependence clusters, namely global variables. They considered the quantity of dependence in 20 programs in general and on the presence of dependence clusters in particular. Techniques were outlined in the study that could be used to locate global variables that were the cause of dependence clusters (i.e., that held the dependence cluster together). There was also some interesting cluster reduction patterns found by removing global variables using transformation techniques. It would be interesting to see how global variables contribute to the formation of coherent dependence clusters rather than same-slice clusters.

Binkley and Harman have presented recent work into the low-level causes of dependence clusters [8]. They show how to isolate and find small atomic units of source code that are responsible for formation of large clusters. They have termed such atomic units 'linchpins'. It would also be interesting to understand how these linchpins affect coherent dependence clusters.

There has also been work on locating dependence structures within source code using Search-Based Software Engineering techniques. Jiang et al. [18] have introduced a general framework for search based slicing, in order to detect dependence structures using search techniques. The application of greedy, hill climbing, and genetic algorithms to find structures showed that search-based techniques could also be used to detect dependence clusters.

Finally, in software maintenance, dependence analysis is used to protect the software maintenance engineer against the potentially unforeseen side effects of a maintenance change. This can be achieved by measuring the impact of the proposed change [10, 17] or by attempting to identify portions of code for which a change can be safely performed free from side effects [20]. The presence of large coherent dependence clusters will be associated with higher values for change impact metrics. This is not only because the statements of a coherent dependence cluster are mutually dependent, but also they have the same dependence relationship with statements external to the cluster.

7. ONGOING AND FUTURE WORK

Coherent-slice and same-slice clusters are built from equivalent slices. Using slice equivalence produces smaller clusters, and thus a conservative result. A more liberal approach would require only mutual slice inclusion. A preliminary experiment was designed to gain a better understanding of how conservative it is to use slice equivalence. The experiment compared the number of pairs of SDG

vertices that were in each other's slices to the number of pairs where both vertices have the same slice:

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

A large difference would give cause for further research into developing better detection algorithms for dependence clusters. The program bc was used in the investigation. It has 7,538 vertices which require 56,821,444 comparisons. Because this process runs in $O(n^3)$ time, the analysis took about 30 days. The result was that 60% of the pairs in each other slice also had the same slice. This result suggests the need for further research into new slice-inclusion based cluster identification techniques. Such clusters could potentially be much larger than those previously reported.

A second direction for future work is to consider mapping clusters onto larger units of the source. For example, identifying source files, classes, or functions associated with a dependence cluster. A visualization tool is currently under development to allow clusters to be mapped to actual source code rather than vertices of an SDG. It would be interesting to look into causes of coherent dependence clusters such as global variables and other linchpin vertices.

Future work on coherent dependence clusters will also consider a wider class of programs. One goal of this experiment is to categorize programs based on the size of coherent dependence clusters that they contain.

8. CONCLUSION

This paper introduced the concept of coherent dependence clusters and presented techniques to detect them. It also introduced Monotone Cluster-Size Graph and Slice/Cluster-Size Graph visualizations to better understand coherent dependence clusters and the relationships between them.

The empirical evaluation of eight subject programs showed that seven contain large coherent-slice clusters, which could potentially cause problems, for example, during maintenance. It also revealed that most coherent clusters are formed from partitions of larger incoherent clusters.

9. REFERENCES

[1] P. Anderson and T. Teitelbaum. Software inspection using CodeSurfer. In *First Workshop on Inspection in Software Engineering*, pages 1–9, 2001.

[2] F. Balmas. Using dependence graphs as a support to document programs. In *Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 145–154, Montreal, 2002.

[3] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23:498–516, 1997.

[4] D. Binkley. Source code analysis: A road map. *ICSE 2007 Special Track on the Future of Software Engineering*, May 2007.

[5] D. Binkley, N. Gold, M. Harman, Z. Li, K. Mahdavi, and J. Wegener. Dependence anti patterns. In *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 25–34, September 2008.

[6] D. Binkley and M. Harman. Forward slices are smaller than backward slices. *Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 15–24, 2005.

[7] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In *21st IEEE International Conference on Software Maintenance*, pages 177–186, 2005.

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

[9] D. Binkley, M. Harman, Y. Hassoun, S. Islam, and Z. Li. Assessing the impact of global variables on program dependence and dependence clusters. *Journal of Systems and Software*, April 2009.

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

[11] S. Black, S. Counsell, T. Hall, and P. Wernick. Using program slicing to identify faults in software. In *Beyond Program Slicing, number 05451 in Dagstuhl Seminar Proceedings*, Dagstuhl, Germany, 2006.

[12] G. Canfora. An integrated environment for reuse reengineering C code. *Journal of Systems and Software*, 42:153–164, August 1998.

[13] A. Elssamadisy and G. Schalliol. Recognizing and responding to "bad smells" in extreme programming. In *International Conference on Software Engineering*, pages 617–622, 2002.

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

[15] M. Harman, K. Gallagher, D. Binkley, N. Gold, and J. Krinke. Dependence clusters in source code. *ACM Transactions on Programming Languages and Systems*, 32(1):1–33, 2009.

[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12:26–60, January 1990.

[17] M. Hutchins and K. Gallagher. Improving visual impact analysis. In *International Conference on Software Maintenance*, pages 294–303, 1998.

[18] T. Jiang, N. Gold, M. Harman, and Z. Li. Locating dependence structures using search-based slicing. *Information and Software Technology*, 50(12):1189–1209, 2008.

[19] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in software development environments. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environmt, SIGPLAN Notices*, 19(5):177–184, 1984.

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

[21] M. Weiser and C. Park. Program slicing. In *International Conference on Software Engineering*, 1981.

[22] D. A. Wheeler. SLOC count user's guide. *http://www.dwheeler.com/sloccount/sloccount.html.*, 2004.