

# Dependence Cluster Visualization

Syed Islam, Jens Krinke  
CREST  
King's College London  
London, UK  
syed.s.islam@kcl.ac.uk,  
krinke@acm.org

David Binkley  
Loyola University Maryland  
Baltimore Maryland, USA  
binkley@cs.loyola.edu

## ABSTRACT

Large clusters of mutual dependence have long been regarded as a problem impeding comprehension, testing, maintenance, and reverse engineering. An effective visualization can aid an engineer in addressing the presence of large clusters. Such a visualization is presented. It allows a program's dependence clusters to be considered from an abstract high level down thru a concrete source-level. At the highest level of abstraction, the visualization uses a heat-map (a color scheme) to efficiently overview the clusters found in an entire system. Other levels include three source code views that allow a user to "zoom" in on the clusters starting from the high-level system view, down through a file view, and then onto the actual source code where each cluster can be studied in detail.

Also presented are two case studies, the first is the open-source calculator `bc` and the second is the industrial program `copla`, which performs signal processing. The studies consider qualitative evaluations of the visualization. From the results, it is seen that the visualization reveals high-level structure of programs and interactions between its components. The results also show that the visualization highlights potential candidates (functions/files) for re-factoring in `bc` and finds dependence pollution in `copla`.

## Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, Enhancement, Restructuring, reverse engineering, and reengineering

## General Terms

Design, Experimentation, Measurement

## Keywords

Dependence, program comprehension, program slicing, clustering, visualization, reverse engineering, re-engineering.

## 1. INTRODUCTION

Program dependence analysis, a key component of source code analysis [?], explores the dependence relationships between program statements. Real-world code has been shown to contain large *clusters* of mutually dependent statements [?]. Such clusters can

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*SOFTVIS'10*, October 17–21, 2010, Salt Lake City, Utah, USA.  
Copyright 2010 ACM 978-1-4503-0028-5/10/10 ...\$10.00.

impede comprehension [?], maintenance and evolution [?], testing [?], and analysis [?]. For these reasons, dependence clusters can be regarded as anti pattern [?], pollution [?] or a bad code smell [?].

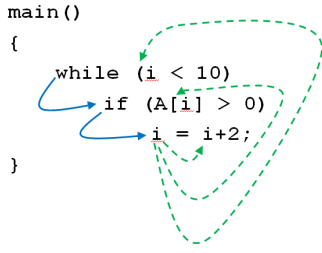
Prior work [?] has shown that dependence clusters are prevalent in real-world programs and their sizes can indicate how difficult a program will be to work with [?]. Most of the clusters found in programs are rather small and uninteresting. Typically only the top three or four clusters of medium-sized programs (around 10,000 SLoC) are of interest. For example, they are large enough to have an impact on program comprehension. Over all the programs considered including the two presented as case studies herein, no more than the top seven largest clusters were ever identified with high-level concepts of interest in a program. Dependence cluster visualizations thus far have been size-graphs aimed solely at showing a statistical summary of dependence clusters. In contrast, we provide a visualization of (coherent) dependence clusters that often map to particular concepts in a program. Such high-level abstraction helps an engineer form a mental model and consequently aids in comprehension, maintenance, and reverse engineering.

The primary contribution of this paper is the multi-level visualization of dependence clusters using a new tool named *decluvi*. The visualization aids an engineer in understanding the structure of a program by providing a quick summary of the dependence clusters found in the entire system and then mapping these clusters down onto the source code; thus providing a concrete view of the clusters. The paper also presents a qualitative evaluation of the dependence-cluster visualization for the open-source program `bc` and the industrial program `copla`. The evaluation illustrates how visualization of dependence clusters can facilitate extraction of high-level program structure and how it can suggest improvements to this structure. For example, the visualization helps to identify artifacts of `bc` that need restructuring to improve logic separation, cohesion, and abstraction. In the case of `copla`, dependence pollution is identified, which causes problems during software maintenance.

The remainder of this paper is organized as follows: Section ?? provides background on dependence clusters and previous dependence cluster visualization techniques, while Section ?? introduces the new visualization. Section ?? presents two case studies and evaluation. Section ?? describes related work, while Section ?? highlights future work, and finally, Section ?? summarizes the work presented.

## 2. BACKGROUND

This section provides background information on dependence clusters and existing visualizations. It first formalizes mutually dependent sets and dependence clusters together with a specialized



Blue (solid) lines represent control dependence.  
Green (broken) lines represent data dependence.

**Figure 1: Dependence cluster example**

form called *coherent* dependence clusters, followed by a review of the existing dependence cluster visualization techniques.

## 2.1 Dependence Clusters

Harman et al. [?] defined a *dependence cluster* as a maximal set of program statements that mutually depend upon one another.

### Definition 1 (Mutually-Dependent Cluster [?])

A *Mutually-Dependent Cluster* is a maximal set of statements,  $S$ , such that  $\forall x, y \in S : x$  depends on  $y$ .

The above definition is parameterized by an underlying *dependence* relation. Ideally, such a relation would precisely represent the impact, influence, or dependence of one statement upon another. Unfortunately, such a relation is not computable. One well known approximation is based on Weiser’s *Program Slice* [?]: a slice is a set of program statements that affect the values computed at a statement of interest. One common slicing algorithm is based on a program’s System Dependence Graph (SDG) [?]. 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.

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

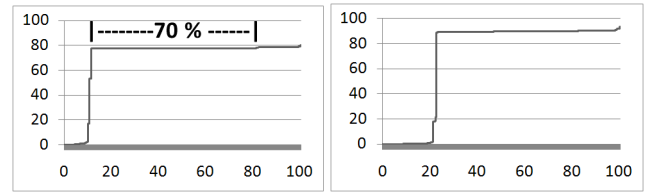
The following definitions are given using  $BSlice$ . Each has a dual that uses  $FSlice$ . When the distinction is important, *backward* and *forward* will be added to the definition name for clarification.

### Definition 2 (Slice-based Cluster [?])

A *Slice-based Cluster* is a maximal set of vertices,  $V$ , such that  $\forall x, y \in V : x \in BSlice(y)$ .

As a simple illustrative example of a slice-based dependence cluster, consider the example in Figure ???. In this example, the predicate  $i < 10$  data depends on the assignment to  $i$ , this assignment control depends on the predicate of the if statement, and the if control depends on the predicate  $i < 10$ . As a result, all three statements are mutually inter-dependent (or in each other’s  $BSlice$ ); they form a cluster.

Definition ?? for slice-based clusters permits clusters to overlap. An alternative identifies maximal partitions. Such partitions correspond to statements which closely model the components that work together within a program. The partitioning can be achieved



(a) B-MSG

(b) F-MSG

$y$ -axis: Slice Size (percent of program)  
 $x$ -axis: Slice Rank (percent of program)

**Figure 2: MSGs for the program bc.**

by replacing the *slice inclusion* relationship of Definition ?? with *same-slice*:

### Definition 3 (Same-Slice Cluster [?])

A *Same-Slice Cluster* is a maximal set of vertices,  $V$ , such that  $\forall x, y \in V : BSlice(x) = BSlice(y)$ .

Along with the internal requirement found in the slice inclusion definition of a Slice-based Cluster, a Same-Slice Cluster has the added external requirement that, all vertices in the cluster are affected by the same vertices external to the cluster. Coherent (dependence) clusters extend this external requirement to further include the external vertices affected by the elements of a cluster. The extension has the advantage that the entire cluster is both affected by the same set of vertices (as is the case with same-backward-slice clusters) and also affects the same set of vertices (as is the case with same-forward-slice clusters). Incorporating internal dependence and both kinds of external dependence, Coherent Clusters are defined as follows:

### Definition 4 (Coherent Cluster [?])

A *Coherent Cluster* is a maximal set of vertices  $V$ , such that  $\forall x, y \in V : x$  depends on  $a$  implies  $y$  depends on  $a$  and  $a$  depends on  $x$  implies  $a$  depends on  $y$ .

A slice-based instantiation for the above definition of coherent cluster is *Coherent-Slice Cluster*.

### Definition 5 (Coherent-Slice Cluster [?])

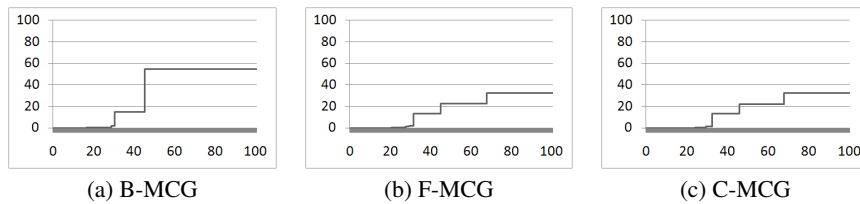
A *Coherent-Slice Cluster* is a maximal set of vertices,  $V$ , such that

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

## 2.2 Cluster Visualizations with Size-Graphs

Three size-based graphs have been previously considered to visualize dependence clusters: the *Monotone Slice-Size Graph*, the *Monotone Cluster-Size Graph* and the *Slice/Cluster-Size Graph*. First, as illustrated in Figure ??, the Monotone Slice-Size Graph (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. MSGs drawn using backward slice sizes are referred to as backward-slice MSG (B-MSG), those drawn using forward slice sizes are referred to as forward-slice MSG (F-MSG). In an MSG a dependence cluster appears as a sheer-drop cliff face followed by a plateau. For example, the B-MSG in Figure ??a shows a large plateau depicting a same-backward-slice cluster spanning almost 70% of the program bc.

The second view, illustrated in Figure ??, is the Monotone Cluster-Size Graph (MCG) [?], which visualizes clusters based on their cluster size rather than their slice size. In an MCG, cluster sizes of vertices are plotted on the  $x$ -axis in monotonically increasing



*y*-axis: Cluster Size (percent of program)  
*x*-axis: Cluster Rank (percent of program)

**Figure 3: MCGs for the program bc.**

order with the sizes (as a percentage of the entire program) plotted on the *y*-axis. MCGs 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). In an MCG a program's (same-slice/coherent-slice) clusters are clearly identified as steps. For example, MCGs in Figure ?? show the presence of two large same-backward-slice clusters, three same-forward-slice clusters and three coherent-slice clusters.

Finally, a combination of the MSG and MCG, the Slice/Cluster-Size Graph (SCG) [?] links slice and cluster sizes. As illustrated in Figure ?? an SCG plots three landscapes, one for increasing slice sizes (solid black line), one for the corresponding same-slice cluster sizes (light gray line), and the third for the corresponding coherent-slice cluster sizes (dashed red (gray in black & white) line). 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 the second and third. 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 built from the sizes of forward slices, same-forward-slice clusters, and coherent-slice clusters. SCGs of *bc* (Figure ??) show three coherent clusters along with the same-slice clusters. The backward and forward slice size for the vertices of the clusters are also shown, providing a link between clusters and the underlying slice sizes.

### 3. IMPROVED VISUALIZATION

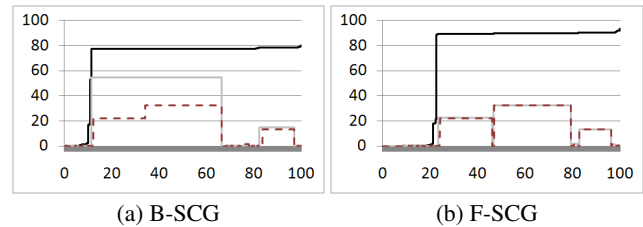
Building on previous visualizations that show only slice and cluster statistics, this section first presents design considerations for the new visualization. It then describes the proposed visualization's four views of a program: the heat-map and three code-based views. Finally, the section ends with a discussion of the data acquisition process used to gather data from which the visualizations were generated.

#### 3.1 Design Consideration

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

##### Tasks - why will the visualization help?

The visualization helps to quickly identify computations involved in clusters of dependence also shows interactions between these computations. This identification makes it easier to understand and modify a program. The visualization also



*y*-axis: Slice/Cluster Size (percent of program)  
*x*-axis: Slice/Cluster Rank (percent of program)

**Figure 4: SCGs for the program bc where slice sizes are shown in solid black line, same-slice cluster sizes are shown in light gray line, and coherent-slice cluster sizes in dashed red (gray in black & white) line.**

helps identify files and functions where multiple clusters are involved. Here re-structuring will improve cohesion and design abstraction.

##### Audience - who will make use of the visualization?

Maintainers will use the visualization to gain overall understanding of the program and estimate the impact of changes. Developers will also use the visualization to check if their implementation matches their documented architecture and to identify potential problems in the structure.

##### Target - what data source is to be represented?

Details of dependence clusters calculated from program.

##### Medium - where to represent the visualization?

The visualization will involve highly interactive computer graphics being displayed on a color monitor.

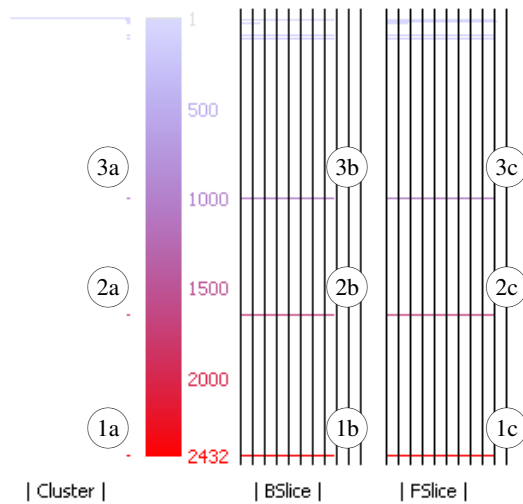
##### Representation - how to represent the data?

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

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

#### 3.2 Heat-Map View

The *Heat-Map View* is the first of the four views which work together to aid an engineer in creating a mental model of system components. The high-level understanding provided by this view can be traced to the source code using the other three views. Together, the four views assist with locating dependence cluster causes, allowing an engineer to decide whether the clustering is necessary or a form of 'pollution'. The views also highlight files and functions with multiple embedded functionalities; suggesting possible loca-



**Figure 5: Heat-Map View for the program bc showing the color spectrum (shades of gray in black & white) used to represent cluster size.**

tions on which to focus re-factoring efforts in order to improve the logical separation, cohesion, abstraction, and reduce code deterioration during evolution.

The *Heat-Map View* is the central starting point to the visualization that displays an overview of all the clusters using color to distinguish clusters of varying sizes. The same color scheme is also used to represent the cluster size metric in other abstract and concrete views. The view also displays additional statistics such as the size of the backward and forward slices for each coherent cluster and number of clusters for each cluster size. Figure ?? shows the Heat-Map View for bc, which has been annotated for the purpose of this discussion. The three labels 1a, 1b, and 1c highlight statistics for the largest cluster (Cluster 1) of the program, whereas 2a, 2b, and 2c highlight statistics of the 2<sup>nd</sup> largest cluster (Cluster 2) and the 3<sup>rd</sup> the 3<sup>rd</sup> largest cluster (Cluster 3). Starting from the left of the heat-map, using one pixel per cluster, horizontal lines (capped at 100 pixels) show the number of clusters that exist for each cluster size. This helps identify cases where there are multiple clusters of the same size. For example, the single dot next to the labels 1a, 2a and 3a show that there is one cluster of each of the three largest sizes. A single occurrence is common for large clusters, but not for small clusters as illustrated by the long line at the top left of the heat-map. This line indicates multiple (uninteresting) clusters having size one.

On the right of the cluster counts is the actual heat-map (color spectrum) showing dependence cluster sizes from small to large going from top to bottom using colors varying from blue to red. In gray-scale this appear as shades of gray, with lighter shades (corresponding to blue) representing smaller clusters and darker shades (corresponding to red) representing larger clusters. Red is used for larger clusters as they are more likely to encompasses complex functionality making them more important, or “hot topics”.

A numeric scale runs on the right of the heat-map, where the numbers show the cluster size (measured in SDG vertices) represented by the color at the same horizontal level as the number. In Figure ??, the scale for bc runs from 1 - 2432, depicting the sizes from the largest to the smallest clusters, displayed using bright red (dark gray) to light blue (light gray), respectively.

Finally on the right of the number scale, two more statistics are displayed: |BSlice| (labeled 1b, 2b and 3b) and |FSlice| (labeled 1c, 2c and 3c). These represent the sizes of the backward slice and

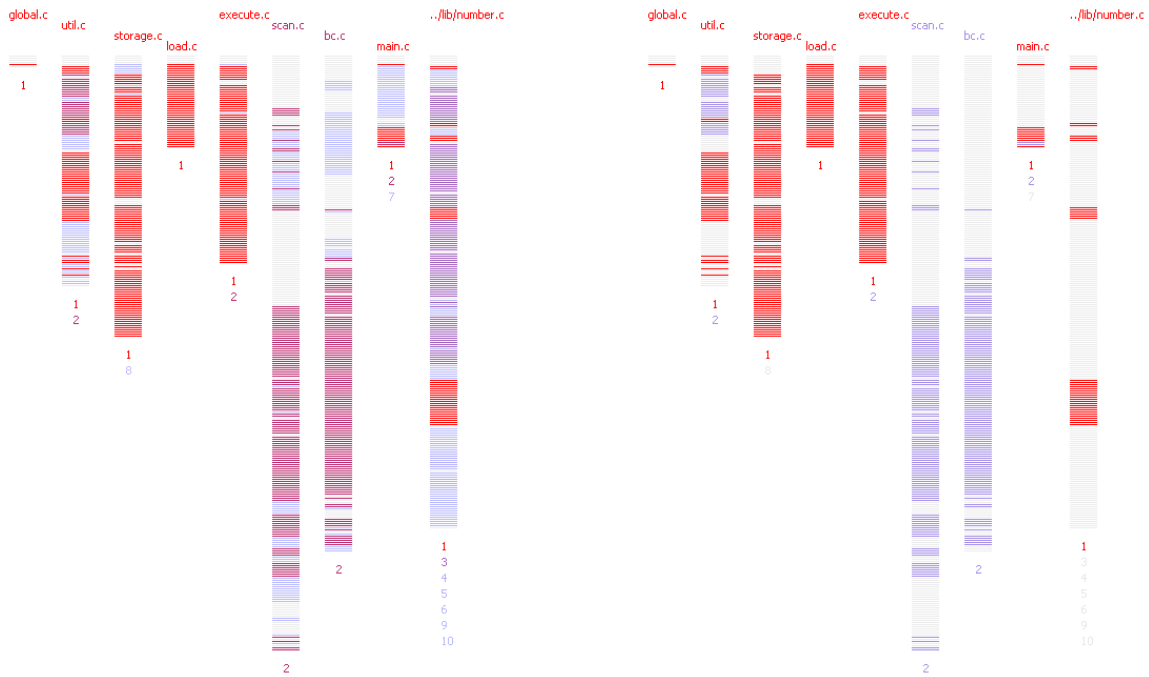
the forward slice for the vertices that form a coherent cluster. The sizes are shown as a percentage of the entire program’s SDG size (vertex count), with the separation of the vertical bars representing 10% of the SDG size. For example, Cluster 1’s BSlice (1b) and FSlice (1c) include approximately 80% and 90% of the program’s SDG vertices. For this particular example, the FSlice’s and BSlice’s that make up the top three clusters are very similar in size, resulting in bars of almost the same length.

### 3.3 Three Code-Based Views

The level of abstraction offered by a visualization must cope with the volume of code being visualized, which ranges from thousands of lines in a moderately sized system to millions of lines in a larger system. If a view is too detailed it becomes incomprehensible for the analysis of large systems. On the other hand, if the level of abstraction is too high, low-level detail is lost. In addition, programmers are most comfortable in the spatial structure in which they have written the program. Therefore, a view of the source is often preferred. However, a single glance at a high-level view of the entire system allow engineers to ascertain the level of clustering and understand overall system structure. As a compromise between these conflicting needs, the visualization provides three different views that allow a program’s clusters to be viewed at varying levels of detail: *System View*, *File View*, and *Source View*.

The bird’s eye view shown by the *System View* is at the highest level of abstraction. Each file of the system (containing executable source code) is abstracted into a column. This yields the nine columns for bc, seen in Figure ??a. The name of the file appears at the top of each column, color coded to reflect the largest cluster in the file. The vertical length of a column represents the length of the corresponding source file. To keep the view compact, each line of pixels in a column summarize multiple source lines. For moderate sized systems, such as the two case studies presented, each pixel line typically represents eight source code lines. The color of each pixel line reflects the largest cluster found among the summarized lines of code, with light gray denoting source code lines that do not include executable code. Finally, the numbers at the bottom of each column indicate the presence of the top 10 clusters in the file (1 denotes the largest cluster of the program whereas 10 is the 10<sup>th</sup> largest cluster).

*Decluvi* provides options for *filtering* and *relative coloring*. Filtering allows a range of cluster sizes of interest to be defined. Only clusters whose size fall within the range are shown using the heat-map colors. Those outside the specified range along with non-executable lines of code are shown in light gray, on grayscale these appear in the lightest shade of gray. The filtering system incorporates a feature to *hide* non-executable lines of code as well as clusters outside the specified range. These features help to isolate particular cluster of interest. In addition, relative coloring allows the heat-map colors to be automatically adjusted to fit within a defined cluster size range. Relative coloring along with filtering overcomes the problem where clusters of similar sizes are represented using similar colors making them indistinguishable. Figure ??a and Figure ??b shows *unfiltered* and *filtered* System View for the program bc. The filtered view is configured to show only the two largest clusters using heat-map colors. With absolute coloring these two clusters have very similar colors because they are close in size. Relative coloring is used to display the two clusters using colors from opposite ends of the spectrum; the largest cluster is seen in Figure ??b in red (dark gray) and the second largest clusters is in blue (medium gray). Comparing the columns for number.c, in Figure ??a it is seen that there are multiple clusters displayed in color from the heat-map (shades of gray) as opposed to that in Fig-



(a) *Unfiltered*: Displaying all clusters using heat-map colors (b) *Filtered*: Displaying only Cluster 1 and 2 using heat-map colors

**Figure 6: System View for the Program bc showing each file using one column, and each line of pixels summarizing cluster size data from eight source lines.**

ure ??b where only Cluster 1 is shown in red (dark gray), the rest of the cluster are filtered out (displayed in light gray). The filtering enables visual separation of a set of clusters (of interest), allowing one to easily examine the artifacts (lines/functions/files) that make up the clusters, and understand their interaction.

The *File View*, illustrated in Figure ??, is at a lower level of abstraction than the System View. It essentially zooms into a column of the System View. In this view, each pixel line corresponds to one line of source code. The pixel lines are also indented to mimic the indentation of the source lines and the number of pixels used to draw each pixel line is the same as the number of characters in the represented source code line. This makes it easier to relate this view to actual lines of source code. The color of the pixel line depicts the size of the largest cluster to which any of the SDG vertices representing the corresponding source code line belong. Figure ?? shows the Line View of bc’s file util.c, filtered to show only the two largest clusters using colors of the heat-map, while smaller clusters and non-executable lines are shown in light gray. The lines shown in red (dark gray) and blue (medium gray) belong to Clusters 1 and 2, respectively.

While the first two views aid in locating parts of the system involved in one or more clusters, the *Source View* allows a programmer to see the actual source code that makes up the cluster. This can be useful in addressing questions such as: *Why are the clusters formed? What binds the cluster together? Is there dependence pollution?* The *Source View* illustrated in Figure ?? is a concrete view that maps the clusters onto actual source code lines. The lines are displayed in the same spatial context in which they were written, line color depicts the size of the largest cluster to which the SDG vertices representing the line belong. Figure ?? shows lines 241-257 of bc’s file util.c which has also been filtered to show only the largest two clusters using colors from the heat-map. The lines of code whose corresponding SDG vertices are part of the largest cluster are shown in red (dark gray) and those lines whose SDG

vertices are part of the second largest cluster are shown in blue (medium gray). Other lines that do not include any executable code or whose SDG vertices are not part of the two largest cluster are shown in light gray. On the left of each line is a *line tag*, which denotes as  $a : b|c/d$ : the line number ( $a$ ), the cluster number ( $b$ ), and an identification  $c/d$  for the  $c^{th}$  of  $d$  clusters having a given size. For example, in Figure ?? lines 250 and 253 are both part of the 20<sup>th</sup> largest cluster (clusters with same size have the same rank) as indicated by the value of  $b$ ; however they belong to different clusters as indicated by the differing values of  $c$  in their line tags.

### 3.4 Data Acquisition

The visualization is generated from backward and forward slices taken with respect to each vertex of a program’s SDG. The slices along with the mapping between the SDG vertices and the actual source code is extracted from the mature and widely used slicing tool *CodeSurfer* [?]. The cluster sizes measured by *decluvi* is in terms of the SDG vertices; which exclude pseudo vertices introduced into the SDG, to represent, for example, global variables, which are modeled as additional pseudo parameters by *CodeSurfer*. Measuring size in terms of vertices is more accurate than using lines of code because it is not influenced by blank lines, comments, statements spanning multiple lines, multiple statements in one line, or compound statements.

## 4. USE-CASE AND EVALUATION

This section presents a qualitative evaluation of coherent dependence clusters and their visualization. Two case-studies based on the programs *bc* and *copia* are presented, followed by a discussion of threats to validity. The section ends with the evaluation of *decluvi*’s user-interface.

### 4.1 Case Study: bc





**Figure 7: File View (filtered) for the file util.c of Program bc.** Each line of pixels correspond to one source code line. Blue color (medium gray in black & white) represents 2<sup>nd</sup> largest cluster, and red color (dark gray) represents the largest cluster. The rectangle in the first column marks function `init_gen` which contains both clusters.

This subsection presents a case study of the program `bc`. It starts with a brief description of the program followed by the results of applying the visualization to the program. The program `bc` is an open-source calculator, which consists of 16,763 LOC (lines of code as counted by the Unix utility `wc`) and 11,173 SLoC (non-comment non-blank lines of code as counted by `sloc` [?]). The program has nine C files for which CodeSurfer produces 15,076 slices (backward and forward).

The Heat-Map View for `bc` (Figure ??) shows the presence of three large clusters and three smaller clusters which are readily distinguishable from each other. We will focus on these six clusters (detailed in Table ??) as the rest of the clusters are extremely small: while the largest clusters consists of 2432 vertices, the 7<sup>th</sup> largest cluster consists of a mere 25 vertices.

As seen in Figure ??, Cluster 1 spans all files in `bc` except for `scan.c` and `bc.c`. This cluster encompasses the core functionality of the program – *loading and handling of equations, converting to bc’s own number format, performing calculations, and accumulating results*.

Cluster 2 spans five files, `util.c`, `execute.c`, `main.c`, `scan.c`, and `bc.c`. The majority of the cluster is distributed over the last two

```

241:0|0|0      /* Initialize the code generator the parser. */
242:0|0|0
243:0|0|0
244:1|1|1      void
                init_gen ()
245:0|0|0      {
246:0|0|0          /* Get things ready. */
247:2|1|1          break_label = 0;
248:2|1|1          continue_label = 0;
249:2|1|1          next_label = 1;
250:20|257|1645 out_count = 2;
251:1|1|1          if (compile_only)
252:19|20|51          printf ("8i");
253:20|49|1645          else
254:1|1|1              init_load ();
255:1|1|1          had_error = FALSE;
256:2|1|1          did_gen = FALSE;
257:0|0|0      }

```

**Figure 8: Source View (filtered) showing function `init_gen` in file `util.c` of Program `bc`.** Blue color (medium gray in black & white) represents 2<sup>nd</sup> largest cluster and red color (dark gray) represents the largest cluster.

```

1211:6|1|1      while ( !bc_is_zero(exponent) )
1212:0|0|0      {
1213:6|1|1          (void) bc_divmod (exponent, _two_, &exponent, &parity, 0);
1214:6|1|1          if ( !bc_is_zero(parity) )
1215:0|0|0      {
1216:6|1|1          bc_multiply (temp, power, &temp, &scale);
1217:6|1|1          (void) bc_modulo (temp, mod, &temp, &scale);
1218:0|0|0      }
1219:0|0|0      }
1220:6|1|1          bc_multiply (power, power, &power, &scale);
1221:6|1|1          (void) bc_modulo (power, mod, &power, &scale);
1222:0|0|0      }

```

**Figure 9: Source View for the file `number.c` of Program `bc` showing the lines whose vertices form the 6<sup>th</sup> largest cluster.**

Cluster	Cluster Size		Number of files spanned
	%	vertices/lines	
1	32.3%	2432/1411	7
2	22.0%	1655/999	5
3	13.3%	1003/447	1
4	1.6%	117/49	1
5	1.4%	102/44	1
6	0.4%	32/7	1

**Table 1: Top Six coherent clusters of `bc`**

files. Even more interestingly, these two files contain only Cluster 2 from the set of the top 6 clusters, which indicates a clear purpose to the existence of the files. The files are solely used for *lexical analysis and parsing* of equations. To aid in this task some utility functions from `util.c` are employed. Only five lines of code in `execute.c` are part of Cluster 2 and are used for *flushing output and clearing interrupt signals*.

The third cluster is completely contained within the file `number.c` and spans 1003 vertices. The cluster encompasses functions such as `_bc_do_sub`, `bc_init_num`, `_bc_do_compare`, `_bc_do_add`, `_bc_simp_mul`, `_bc_shift_addsub`, and `_bc_rm_leading_zeros`, which are responsible for *initializing bc’s number formatter, performing comparisons, modulo and other arithmetic operations*. Clusters 4 and 5 are also completely contained within `number.c`. These clusters encompass functions to perform *bcd operations for base ten numbers and arithmetic division*, respectively. Cluster 6 (Figure ??) of `bc` spanning (32 vertices) 7 lines of code is formed because of a while loop which checks if the exponent is zero.

The results of the cluster visualization for `bc` as described above reveals its high-level structure. This aids an engineer in understand-

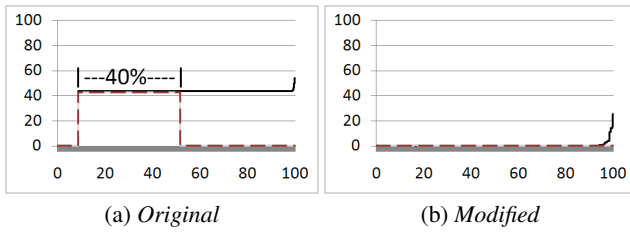


Figure 10: SCGs for the program copia

ing how the artifacts (functions/files) of the program interact to provide various functionalities. The visualization thus aids in program comprehension. The following illustrates how the visualization can also aid in finding potential problems and its causes.

Util.c consists of small utility functions called from various parts of the program. This file contains parts of Cluster 1 and 2. Both the separate functionalities identified previously (encompassed by each of the clusters) make use of the utility functions defined within the file. Figure ?? shows that two clusters in red (dark gray) and blue (medium gray) within the file are well separated. The clusters do not intersect inside any function with the exception of `init_gen` (rectangle in first column of Figure ??). The Code View of this function illustrated in Figure ?? shows lines 244, 251, 254, and 255 in red (dark gray) from Cluster 1, and lines 247, 248, 249, and 256 in blue (medium gray) from Cluster 2. Other lines belonging to smaller clusters and those not containing any executable code are shown in light gray. Functions should be refactored to avoid containing more than one cluster as such functions reduce code separation (hindering comprehension) and increase the likelihood of ripple-effects [?]. A common exception to guidelines are initialization functions such as `bc`'s `init_gen`, which initializes the parser code generator. The remaining functions belonging to each cluster should be separated by splitting `util.c` into two files, with each file dedicated to functions interacting with one of the two largest clusters. This would improve logic separation and cohesion at the file level, making the code easier to understand and maintain.

From `bc`'s SCGs (Figure ??) two interesting observations can be made. First, the program `bc` contains two large same-backward-slice clusters 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 dashed red (dark gray) landscape. This indicates that the same-backward-slice cluster splits into the two coherent-slice clusters, supporting the conjecture that coherent clusters are more suited to modeling components of a program than other forms of dependence clusters.

The visualization reveals the high-level structure for the program `bc` and shows how different artifacts (functions/files) of the program interact with each other. This makes it easier for engineers to understand the program. By identifying artifacts of the program which have multiple embedded functionalities, the visualization also identifies areas of low cohesion. These can be sources of code degradation during evolution. By highlighting such problems, the visualization successfully suggests artifacts that should be the focus of refactoring efforts. The slice/cluster sizes from the visualization provides an estimate of the level of difficulty likely to be faced by testers and maintainers when dealing with program. Artifacts that are part of larger clusters are harder to test and modify than those that are part of smaller clusters.

## 4.2 Case Study copia

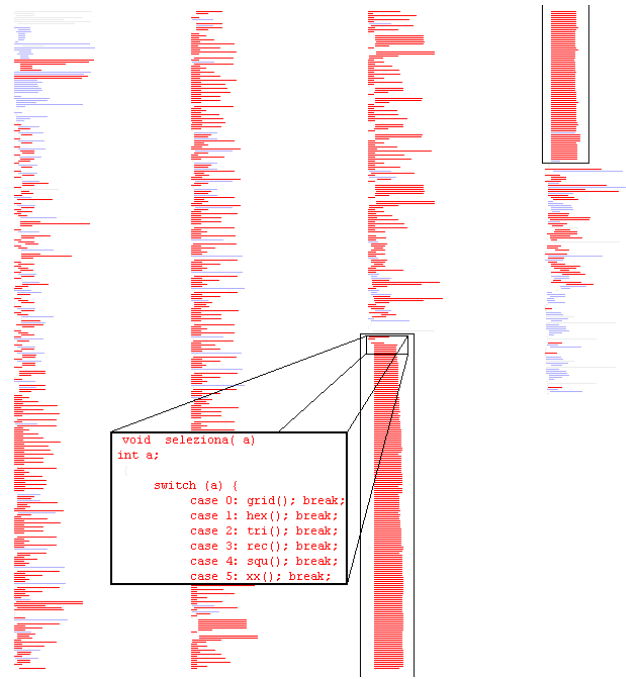


Figure 11: File View for the file `copia.c` of Program `copia`. Each line of pixels represent the cluster size data for one source code line.

This subsection presents a case study of `copia`, an industrial program used for ESA signal processing. The program consists of 1,170 LOC, and 1,112 SLoC. It has only one C file from which CodeSurfer extracts 7518 slices (backward and forward). The program `copia` has a large coherent cluster spanning 40% of the B-SCG as shown by the dashed red (dark gray) line (running from 10% to 50% on the  $x$ -axis) in Figure ??a.

During the analysis of `copia`'s File View we were drawn towards an intriguing structure. There is a huge block of code with same spatial arrangement (bounded by black rectangle in Figure ??) that belongs to a single large cluster of the program. It is unusual for so many consecutive source code lines to have similar length and indentation. Source View of the the corresponding lines revealed that this unusual large block of code is a `switch` handling 234 cases. Upon inspection of `copia` it was found that the program has 234 small functions that all call one large function, `seleziona`, which in turn calls the smaller functions effectively implementing a finite state machine. Each of the smaller functions return a value that is the next state for the machine and is used by the switch to call the appropriate function. The primary reason for the high level of dependence in the program lies in the statement `switch(next_state)`, which controls the call to the smaller functions. This causes what might be termed 'conservative dependence analysis collateral damage' because the static analysis can not determine that when function `f()` returns a 5 this causes the switch statement to eventually invoke function `g()`. Instead, the analysis makes the conservative assumption that a call to `f()` might be followed by a call to any of the functions appearing in the switch statement, resulting in a mutual recursion involving most of the program.

This is a clear case of dependence pollution where the next-state value coupled with the mutual recursion is entirely avoidable. To show this, we did a simple refactoring (by hand) to simulate the replacement of the integer variable `next_state` with direct recursive function calls. This removed the potentially problematic `switch`



**Figure 12: decluvi’s Control Panel showing buttons to toggle views, configure filter range, and hide filtered regions.**

statement. The SCGs for both *original* and *modified* versions of the program are shown in Figure ???. As seen in the figure, removing the switch and its control dependence removes the cluster. As a result of this reduction, the potential impact of changes to the program will be greatly reduced, making it easier to understand and maintain.

In this particular case study, owing to peculiar coding structure, the visualization was used to identify the cause of the cluster. This however is not the primary purpose of the visualization. The visualization as presented can show the existence of clusters, helping in program comprehension, but cannot pinpoint a cause of the clusters. It currently can only assist in investigating the program for possible causes. However, it is not difficult to incorporate other techniques [?, ?] of identifying causes of dependence clusters and view them using the presented visualization.

Finally, in the B-SCG for *copla* (Figure ??a) it is seen that the plots for Backward-Slice Cluster Sizes (light gray line) and the coherent cluster sizes (dashed red (gray in black & white) line) are very similar. This has happened because, in this case the size of the coherent-slice clusters are restricted by the size of the same-backward-slice clusters. Although, the plot for the size of the backward slices (black line) seems to be the same from the 10% mark to 95% mark on the *x*-axis, the slices are not exactly same size. Only vertices plotted from 10% through to 50% have exactly same backward slice. The tolerance implicit in the visual resolution used to plot the slice sizes obscures this detail.

The case study for *copla* shows that the visualization can not only show the existence and distribution of coherent clusters, but the detailed views can also lead to identification of dependence pollution. Restructuring of such programs to remove the pollution, makes the program easier to understand, test and maintain.

### 4.3 Threats to Validity

In the study, the primary 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 as we present only two case studies. The programs presented are however from different domains: one open-source and another industrial, both of which have previously been the subject of many studies. Also, the programs studied to date, including the two case studies, are mid-sized. It is possible that the value of the visualization does not scale to larger systems. Finally, a threat arises from the potential for faults in the slicer. A mature and widely used slicing tool (CodeSurfer) was used to mitigate this concern.

### 4.4 decluvi’s Interface Evaluation

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

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

The abstract Heat-Map View and compact System View provide an overview of the clustering for an entire system.

**Zoom** - *Zoom in on items of interest.*

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

**Filter** - *Filter out uninteresting items.*

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

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

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

**Relate** - *Clear relationship between the various views.*

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

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

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

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

The tool provides support for exporting slice/cluster statistics.

## 5. RELATED WORK

The work presented follows two separate sub-domains of software engineering. The first is dependence cluster analysis and the second is software visualization. The following subsections describe relevant work from these sub-domains.

### 5.1 Dependence Clusters

Binkley and Harman [?] 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. They deemed dependence clusters as problems for software maintenance and regarded them as anti-patterns [?], pollution [?] and bad code smells [?]. Black [?] has even hypothesized a direct relationship between the size of dependence clusters and number of software faults. Binkley et al. followed up their initial work and presented work on the causes (low-level [?] and high-level [?]) of dependence clusters. Islam et al. [?] have recently introduced coherent clusters suggesting that such clusters have the potential to reveal high-level structures of systems.

### 5.2 Software Visualization

This subsection first presents the current dependence cluster visualization techniques. It then goes on to describe tools and techniques used in aiding program comprehension.



Binkley et al. [?] were the first to introduce a graph visualization for dependence clusters known as MSG. Islam et al. [?] later extended this by introducing MCG and SCG. All these three visualizations were however size-graphs aimed solely at showing the presence of dependence clusters and their statistics. These visualizations do not aid in program understanding as they lack a mapping to source code.

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

The approach pioneered by *Seesoft* was also used in many other visualization tools. *SeeSys* System [?] was developed to localize error-prone code through visualization of ‘bug fix’ statistics. The tool extended the *Seesoft* approach by introducing treemaps to show hierarchical data. It displayed code organized hierarchically into subsystems, directories, and files by representing the whole system as a rectangle and recursively representing the various sub-units with interior rectangles. The area of each rectangle was used to reflect statistic associated with its sub-unit. Tarantula [?] also employs the “line of pixel” style code view introduced by *Seesoft*. The tool was aimed at visualizing the pass/fail of test cases. It extended the idea of using solid colors to represent statistics by using hue and brightness to encode additional information. *CVSscan* [?] also inherited and extended the “line of pixel” based representation by introducing “dense pixel display” to show the overall evolution of programs. The tool has a bi-level code display that provide views of both the contents of a code fragment and its evolution over time. *Source Viewer 3D* [?] is a software visualization framework that is based on *Seesoft* and adds a third dimension (3D) to the original approach allowing additional statistics to be visualized. *Augur* [?] is also based on the line-oriented approach of *Seesoft*. The primary view is spatially organized as in *Seesoft* with additional columns to display multiple statistics for each line. *Aspect Browser (Nebulous)* [?] provides a global view of how the various aspect entries cross-cut the source code using “line of pixels” view and uses *Aspect Emacs* to get the statistics and provide the concrete source code view. *BLOOM* [?] uses the *BEE/HIVE* [?] architecture, a powerful back-end that supports a variety of high-density, high-quality visualization one of which (*File Maps*) is based on the *Seesoft* layout.

The final set of systems discussed are those that aim to help in reverse engineering but are not based on the “line of pixels” approach. Most of these tools focus on visualizing high-level system abstractions (often referred to as ‘clustering’ or ‘aggregation’) such as classes, modules, and packages, using a graph-based approach. *Rigi* [?] is a reverse engineering tool that uses Simple Hierarchical Perspective (*SHriMP*) views, employs fisheye views of nested graphs. *Creole* [?] is an open-source plugin for the Eclipse (IDE) based on *SHriMP*. Tools such as *GOOSE* [?], *Sotograph* [?] and *VizzAnalyzer* [?] work on the class and method levels allowing information aggregation to form higher levels of abstractions. There

are tools (Borland Together, Rational Rose, ESS-Model, BlueJ, Fujaba, GoVisual [?]) which also help in reverse engineering by producing UML diagrams from source code.

## 6. FUTURE WORK

Future work will involve a wide-scale qualitative study into how well *decluvi* supports software comprehension and maintenance. The feedback and survey results from such work can be used to further improve *decluvi*. Preliminary evidence for the success of such study is found in the case studies of *bc* and *copla* where we identified several improvements that will make *decluvi* more effective:

- Addition of intermediate abstractions to visualize clusters at function, component and directory level. This will make it easier to understand the inter-play of clusters and help focus on re-engineering of artifacts containing multiple clusters.
- Improve the algorithm used to calculate color for the pixel lines in the System View by adding anti-aliasing features to incorporate cluster size statistics from all summarized lines of source code.
- Add 3D to visualize the number of clusters of each size to address cases where multiple clusters have the same size and cannot be readily distinguished using color.
- Add support for *history*, *undo*, and *replay* to allow users to backtrack their steps.

## 7. CONCLUSION

The paper introduced new multi-level dependence cluster visualization that aids in comprehension, maintenance, and reverse engineering tasks. The visualization is realized using *decluvi*, which allows dependence clusters to be viewed in terms of source code rather than statistics. The two case studies show that the new visualization is able to reveal high-level structure of programs and can also be used to ascertain interaction between the different components of a program. The case study for *bc* illustrates that the visualization identifies artifacts with low cohesion where refactoring will make the code easier to understand and also reduce code deterioration during software evolution. The visualization also highlights dependence pollution and its causes in *copla* that can hinder testing and maintenance.

The *decluvi* system along with scheme script for data acquisition and pre-compiled dataset for several open-source programs can be downloaded from:

<http://www.dcs.kcl.ac.uk/pg/syed/tools.html>

## 8. ACKNOWLEDGMENTS

We would like to thank GrammaTech Inc. (<http://www.grammatech.com>) for making CodeSurfer available.

## 9. REFERENCES

- [1] Chisel Group. Creole Homepage: <http://www.thechiselgroup.org/creole>.
- [2] GOOSE Homepage: <http://esche.fzi.de/PROSTextern/software/goose/index.html>.
- [3] Software-Tomography GmbH. Sotograph Homepage: <http://www.software-tomography.com/html/sotograph.htm>.
- [4] P. Anderson and T. Teitelbaum. Software inspection using CodeSurfer. In *First Workshop on Inspection in Software Engineering*, pages 1–9, 2001.

- [5] M. J. Baker and S. G. Eick. Space-filling software visualization. *Journal of Visual Languages & Computing*, 6(2):119–133, 1995.
- [6] T. Ball and S. Eick. Visualizing program slices. *Proceedings of 1994 IEEE Symposium on Visual Languages*, pages 288–295, 1994.
- [7] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23:498–516, 1997.
- [8] D. Binkley. Source code analysis: A road map. *ICSE 2007 Special Track on the Future of Software Engineering*, May 2007.
- [9] 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.
- [10] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In *21st IEEE International Conference on Software Maintenance*, pages 177–186, 2005.
- [11] 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.
- [12] 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.
- [13] S. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13:263–279, July 2001.
- [14] 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.
- [15] G. Canfora. An integrated environment for reuse reengineering C code. *Journal of Systems and Software*, 42:153–164, August 1998.
- [16] S. Diehl. Software visualization. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 718–719, New York, NY, USA, 2005. ACM.
- [17] S. Eick, J. Steffen, and E. Sumner. Seesoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18:957–968, 1992.
- [18] A. Elssamadisy and G. Schalliol. Recognizing and responding to “bad smells” in extreme programming. In *International Conference on Software Engineering*, pages 617–622, 2002.
- [19] J. Froehlich and P. Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 387–396, Washington, DC, USA, 2004. IEEE Computer Society.
- [20] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17:751–761, 1991.
- [21] 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.
- [22] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12:26–60, January 1990.
- [23] S. Islam, J. Krinke, D. Binkley, and M. Harman. Coherent dependence clusters. In *PASTE '10: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, New York, NY, USA, 2010. ACM.
- [24] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization for fault localization. In *Proceedings of the Workshop on Software Visualization, 23rd International Conference on Software Engineering*, May 2001.
- [25] J. I. Maletic, A. Marcus, and M. L. Collard. A task oriented view of software visualization. In *VISSOFT '02: Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, page 32, Washington, DC, USA, 2002. IEEE Computer Society.
- [26] A. Marcus, L. Feng, and J. I. Maletic. 3D representations for software visualization. In *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*, New York, NY, USA, 2003. ACM.
- [27] 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 Environments, SIGPLAN Notices*, 19(5):177–184, 1984.
- [28] T. Panas, J. Lundberg, and W. Lowe. Reuse in reverse engineering. In *IWPC '04: Proceedings of the 12th IEEE International Workshop on Program Comprehension*, page 52, Washington, DC, USA, 2004. IEEE Computer Society.
- [29] S. P. Reiss. Bee/Hive: A software visualization back end. *Proceedings of ICSE 2001 Workshop on Software Visualization, Toronto*, pages 44–48, 2001.
- [30] S. P. Reiss. An overview of BLOOM. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 2–5, New York, NY, USA, 2001. ACM.
- [31] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 336, Washington, DC, USA, 1996. IEEE Computer Society.
- [32] M.-A. D. Storey, K. Wong, and H. A. Müller. Rigi: a visualization environment for reverse engineering. In *ICSE '97: Proceedings of the 19th International Conference on Software Engineering*, pages 606–607, New York, NY, USA, 1997. ACM.
- [33] L. Voinea, A. Telea, and J. J. van Wijk. CVSScan: visualization of code evolution. In *SoftVis '05: Proceedings of the 2005 ACM Symposium on Software Visualization*, pages 47–56, New York, NY, USA, 2005. ACM.
- [34] M. Weiser and C. Park. Program slicing. In *International Conference on Software Engineering*, 1981.
- [35] D. A. Wheeler. SLOC count user’s guide. <http://www.dwheeler.com/sloccount/sloccount.html>, 2004.
- [36] W. G. Yoshikiyo, W. G. Griswold, Y. K. Y, and J. J. Yuan. Aspect Browser: Tool support for managing dispersed aspects. In *First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems—OOPSLA 99*, 1999.