Dependence Cluster Visualization

Syed S Islam† Jens Krinke† David Binkley∗
†Centre for Research on Evolution, Search and Testing (CREST)
King’s College London,
London, UK
∗Loyola University Maryland
Baltimore Maryland, USA
{syed.s.islam, jens.krinke}@kcl.ac.uk, 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 view to a concrete source-level view. At the high-level, the visualization uses a heat-map (a color scheme) to efficiently overview the clusters found in an entire system. It also employs 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 the industrial program copia, which performs signal processing. The studies consider qualitative evaluations of the visualization. From the results, it is apparent that the visualization reveals interesting aspects of the program’s high-level structure. The results also show that the visualization highlights potential candidates (functions/files) for re-factoring in bc and finds dependence pollution in copia.

1. INTRODUCTION
Program dependence analysis, a key component of source code analysis [8], explores the dependence relationships between program statements. Real-world code has been shown to contain large clusters of mutually dependent statements [21]. Such clusters can impede comprehension [13], maintenance and evolution [20], testing [7], and analysis [15]. For these reasons, dependence clusters can be regarded as anti patterns [9], pollution [10] or a bad code smell [18].

Prior work has shown that dependence cluster visualization can help an engineer by providing a quick summary of how difficult a program will be to work with [9]. However, previous visualizations are size-graphs aimed solely at showing a statistical summary of a program rather than aiding in understanding its structure.

In contrast, a visualization of dependence clusters that reveals the high-level structure of a program helps an engineer form a mental model of this structure 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 decluvio. 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 copia. 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 copia, dependence pollution is identified, which can cause problems during software maintenance.

The remainder of this paper is organized as follows: Section 2 provides background on dependence clusters and previous dependence cluster visualization techniques, while Section 3 introduces the new visualization. Section 4 presents two case studies and evaluation. Section 5 describes related work, while Section 6 highlights future work, and finally, Section 7 summarizes the work presented.

2. BACKGROUND
This section provides background information on dependence clusters, existing visualizations of dependence clusters, and the data acquisition process used to gather the data from which they are generated. It first formalizes mutually dependent sets and dependence clusters together with a specialized form called coherent dependence clusters. Finally, current dependence cluster visualization techniques are reviewed.

2.1 Dependence Clusters
Harman et al. [21] defined a dependence cluster as a maximal set of program statements that mutually depend upon one another.

Definition 1 (Mutually-Dependent Set and Cluster [21])
A Mutually-Dependent Set (MDS) is a set of statements, $S$, such that $\forall x, y \in S : x$ depends on $y$. 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.
The above definition is parameterized by an underlying depends-on 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 [34]: 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) [22]. 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 \( \text{BSlice}(v) \), is the set of vertices reaching \( v \) via a path of control and data dependence edges [27]. The second kind of slice, a forward slice, is also taken with respect to vertex \( v \). Denoted \( \text{FSlice}(v) \), it includes the set of vertices reachable from \( v \) via a path of control and data dependence edges [22]. 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 [22].

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

**Definition 2 (Slice-based MDS/Cluster [21])**

A Slice-based MDS is a set of vertices, \( V \), such that
\[
\forall x, y \in V : x \in \text{BSlice}(y).
\]
A Slice-based Cluster is a slice-based MDS contained within no other slice-based MDS.

Being based on slice inclusion, the above definition for a slice-based cluster permits vertices of different 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 by replacing the slice inclusion relationship of Definition 2 with same-slice:

**Definition 3 (Same-Slice MDS/Cluster [21])**

A Same-Slice MDS is a set of vertices, \( V \), such that
\[
\forall x, y \in V : x \in \text{BSlice}(y) \land x \in \text{FSlice}(y).
\]
A Same-Slice Cluster is a Same-Slice MDS contained within no other Same-Slice MDS.

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 in terms of the coherent MDS:

**Definition 4 (Coherent MDS/Cluster [23])**

A Coherent MDS is a set of vertices \( V \), such that
\[
\forall x, y \in V : x \text{ depends on } a \text{ implies } y \text{ depends on } a \text{ and } a \text{ depends on } x \text{ implies } a \text{ depends on } y.
\]
A Coherent Cluster is a Coherent MDS contained within no other Coherent MDS.

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

**Definition 5 (Coherent-Slice MDS/Cluster [23])**

A Coherent-Slice MDS is a set of vertices, \( V \), such that
\[
\forall x, y \in V : x \in \text{BSlice}(y) \land x \in \text{FSlice}(y).
\]
A Coherent-Slice Cluster is a Coherent-Slice MDS contained within no other Coherent-Slice MDS.

Hereafter, Coherent-Slice Clusters are referred to simply as Coherent Clusters.

### 2.2 Cluster Visualizations with Size-Graphs

Three size-based graphs have been considered previously as views of dependence clusters: the Monotone Slice-Size Graph, the Monotone Cluster-Size Graph, and the Slice/Cluster-Size Graph. First, as illustrated in Figure 1, the Monotone Slice-Size Graph (MSG) [10] 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 1a shows a large plateau depicting a same-backward-slice cluster spanning almost 70% of the program \( bc \).

The second view, illustrated in Figure 2, is the Monotone Cluster-Size Graph (MCG) [23], 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 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 shown in Figure 2a show the presence of two large same-backward-slice clusters, three same-forward-slice clusters and three coherent-slice clusters.

Finally, a combination of MSG and MCG, the Slice/Cluster-Size Graph (SCG) [23] links slice and cluster sizes. As illustrated in Figure 3 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 3) 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 the various clusters and the slice sizes.

2.3 Data Acquisition

The visualization is generated from (backward/forward) slices extracted using the Scheme API of the mature slicing tool CodeSurfer [4]. The mapping between the SDG vertices and the actual source code is also extracted. Decluvi measures cluster sizes in terms of the SDG vertices; this excludes 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.

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 considering the proposed visualization’s four views of a program: the heat-map and three code-based views.

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. [25] and second the interface requirements proposed by Shneiderman [31]. 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 will aid in program comprehension, maintenance and reverse engineering by revealing the high-level structure of programs and also aid in program re-engineering by identifying artifacts with low cohesion.

Audience - who will make use of the visualization?

Both maintainers and developers will make use of the visualization.

Target - what data source is to be represented?

Details of dependence clusters are to be represented.

Medium - where to represent the visualization?

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

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 a heat map can be traced to the source code using the other three views. Together, the four views assists with locating dependence clusters causes, allowing an engineer to decide whether the clustering is necessary or a form of ‘pollution’. For example, they highlight files and functions with multiple embedded functionalities; suggesting possible locations on which to focus re-factorings 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. Figure 4 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 2nd largest clusters (Cluster 2) and the 3’s the 3rd 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.

The center of the Heat-Map shows 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 encompass complex functionality making them more important, or “hot topics”. 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.

On the right of the cluster counts is the actual heat-map showing the
Decluvi provides option for filtering and relative coloring. Filter-
ing 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 incorpo-
rates a feature to hide non-executable lines of code as well as clusters
outside the specified range. These features help to isolate particu-
lar clusters of interest. In addition, relative coloring allows the
heat-map colors to be automatically adjusted to fit within a defined
cluster size range. Relative color along with filtering overcomes
the problem where clusters of similar sizes are represented using
similar colors making them indistinguishable. Figure 5a and Fig-
ure 5b shows unfiltered and filtered System View respectively 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 5b in red (dark gray) and the second largest clusters
is in blue (medium gray). Comparing the columns for number.c,
in Figure 5a it is seen that there are multiple clusters displayed in
color from the heat-map (shades of gray) as opposed to that in Fig-
ure 5b 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 easy visual separation of clusters (of interest), allowing one
to easily examine the artifacts (lines/functions/files) that are part of
the cluster. The system also helps understand interaction between
different clusters of interest. In addition, relative coloring allows the
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, illustrated in Fig-
5a, 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 5a. 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 the column rep-
resents the length of the source file. To keep the view compact,
each line of pixels in a column represent multiple source lines. For
moderate sized system, such as the two case studies, each line rep-
resents eight source code lines. The color of each line reflects the
largest cluster summarized 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 clus-
ters in the file (1 denotes the largest cluster of the program whereas
10 is the 10th largest cluster).

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 size 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
(the separation of the vertical bars represent 10% of the program’s
size). Cluster 1’s BSlice (1b) and FSlice (1c) include approxi-
ately 80% and 90% of the program’s SDG vertices.

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, pro-
grammers 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 abstract
view of the entire system allow engineers to ascertain level of clus-
tering, understand overall system structure and identify artifacts for
re-engineering. As a compromise between these conflicting needs,
the visualization provides three different views that allow a pro-
gram’s clusters to be viewed at varying levels of detail: System View, File View, and Source View.

The File View, illustrated in Figure 6, is at a lower level of abstrac-
tion 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 lines are also indented to mimic the in-
dention of the source code and the number of pixels used to draw
each 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 line depicts the size of the
largest cluster to which any of the SDG vertices representing the

Figure 3: SCGs for the program bc where increasing slice sizes are shown using a solid black line, same-slice cluster sizes are shown using a light gray line, and the corresponding coherent-slice cluster sizes are shown using a dashed red (gray in black & white) line.

Figure 4: Heat-Map View for the program bc showing the color spectrum (shades of gray in black & white) used for displaying different sized clusters, number of clusters of each size, backward/forward slice sizes for each clusters size.
source code line belong. Figure 6 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 Cluster 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 7 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 7 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). The lines that do not include any executable code or whose SDG vertices are not part of the two largest clusters 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 7 lines 250 and 253 are both part of the $20^{th}$ largest cluster (clusters with same size have the same rank) as indicated by the value of $b$; however they belong to different clusters as indicated by the differing values of $c$ in their line tags.

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 decluri’s user-interface.

![Figure 4: Heat-Map View for bc](image)

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

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Cluster Size</th>
<th>Number of files spanned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2432/1411</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1655/999</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>1003/447</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>117/49</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>102/44</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>32/7</td>
<td>1</td>
</tr>
</tbody>
</table>

4.1 Case Study: bc

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 [35]). The program has nine C files for which CodeSurfer produces 15,076 slices (backward and forward).

The Heat-Map View for bc (Figure 4) shows the presence of three large clusters and three smaller clusters which are readily distin-
Each line of pixels correspond to one source code line. Blue color (medium gray in black & white) represent lines with vertices belonging to the 2nd largest cluster, and red color (dark gray) represent lines with vertices belonging to the largest cluster, while all other lines with/without executable code are shown in light gray. The rectangle in the first column marks function \texttt{init\_gen} which contains both clusters.

As seen in Figure 5, Cluster 1 spans all files in \texttt{bc} except for \texttt{scan.c} and \texttt{bc.c}. This cluster encompasses the core functionality of the program – loading and handling of equations, converting to \texttt{bc}'s own number format, performing calculations, and accumulating results.

Cluster 2 spans five files, \texttt{util.c}, \texttt{execute.c}, \texttt{main.c}, \texttt{scan.c}, and \texttt{bc.c}. The majority of the cluster is distributed over the last two 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 \texttt{flushing output} and \texttt{clearing interrupt signals}.

The third cluster is completely contained within the file \texttt{number.c} and spans 1003 vertices. The cluster encompasses functions such as \_bc\_do\_sub, \texttt{bc\_init\_num}, \_bc\_do\_compare, \_bc\_do\_add, \_bc\_simp\_mul, \_bc\_shift\_add\_sub, and \_bc\_rm\_leading\_zeros, which are responsible for initializing \texttt{bc}'s number formatter, performing comparisons, \texttt{modulo} and other \texttt{arithmetic} operations.

Clusters 4 and 5 are also completely contained within \texttt{number.c}. These clusters encompass functions to perform \texttt{bcd} operations for base ten numbers and \texttt{arithmetic} division, respectively. Cluster 6 (Figure 8) of \texttt{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 \texttt{bc} as described above reveals its high-level structure. This aids an engineer in understanding 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.

\texttt{Util.c} contains 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 6 shows that distribution of the two clusters in red (dark gray) and blue (medium gray) within the file are well separated.
From the file level, making the code easier to understand and maintain.

The clusters do not intersect inside any function with the exception of \texttt{init\_gen} (rectangle in first column Figure 6). The Code View of this function shown in Figure 7 includes 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. The lines belonging to smaller clusters and those containing only non-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 program comprehension) and increase the likelihood of ripple-effects [13]. A common exception to guidelines are initialization functions such as \texttt{bc’s init\_gen}, which initializes the parser code generator. The remaining functions belonging to each cluster should be separated by splitting \texttt{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 \texttt{bc’s} SCGs (Figure 3) two interesting observations can be made. First, the program \texttt{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 \texttt{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 the artifact. Artifacts that are part of larger clusters are harder to test and change than those that are part of smaller clusters.

4.2 Case Study \texttt{copia}

This subsection presents a case study of \texttt{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 \texttt{copia} has a large coherent cluster spanning 40% of the SCG as shown by the dashed red (dark gray) line (running from 10% to 50% on the x-axis) in Figure 9a.

During the analysis of \texttt{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 10) 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 \texttt{copia} it was found that the program has 234 small functions that all call one large function, \texttt{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 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 \texttt{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 \texttt{f()} returned a 5 this causes the switch statement to eventually invoke function \texttt{g()}. Instead, the analysis makes the conservative assumption that a call to \texttt{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
This subsection provides an evaluation of decluvi’s interface against the list of features suggested by Shneiderman [31].

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 11, 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 all source code views 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 [10] were the first to introduce the notion of dependence clusters that looked into the fine grained structure of clustering based on vertices of a SDG. They deemed dependence clusters as problems for software maintenance and regarded them as anti-patterns [9], pollution [10] and bad code smells [18]. Black [14] 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 [11] and high-level [12]) of dependence clusters. Islam et al. [23] 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. [10] were the first to introduce a graph visualization for dependence clusters known as MSG. Islam et al. [23] 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** [17] 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 [6] 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 [5] 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 [24] 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 [33] 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 [26] 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 [19] 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) [36] 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 [30] uses the BEE/HIVE [29] 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 [32] is a reverse engineering tool that uses Simple Hierarchical Perspective (SHriMP) views, employs fisheye views of nested graphs. Creole [1] is an open-source plugin for the Eclipse (IDE) based on SHriMP. Tools such as GOOSE [2], Sotograph [3] and VizzAnalyzer [28] 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, Fuji-Java, GoVisual [16]) 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 form 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 *copia* 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 inter-play of clusters and help focus on re-engineering of artifacts containing multiple clusters.
- Improve the algorithm used to calculate color for each line of 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 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, refactoring which 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 *copia* 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](http://www.dcs.kcl.ac.uk/pg/syed/tools.html)

### 8. ACKNOWLEDGMENTS

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

### 9. REFERENCES


