Uncovering Dependence Clusters and Linchpin Functions

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

Árpád Beszédes
University of Szeged
Szeged, Hungary
beszedes@inf.u-szeged.hu

Syed Islam
University of East London
London, UK
syed@uel.ac.uk

Judit Jász
University of Szeged
Szeged, Hungary
jasy@inf.u-szeged.hu

Béla Vancsics
University of Szeged
Szeged, Hungary
vancsics@inf.u-szeged.hu

***Abstract*—Dependence clusters are (maximal) collections of mutually dependent source code entities according to some dependence relation. Their presence in software complicates many maintenance activities including testing, refactoring, and feature extraction. Despite several studies finding them common in production code, their formation, identification, and overall structure are not well understood, partly because of challenges in approximating true dependences between program entities. Previous research has considered two approximate dependence relations: a fine-grained statement-level relation using control and data dependences from a program's System Dependence Graph and a coarser relation based on function-level control-flow reachability. In principal, the first is more expensive and more precise than the second.**

Using a collection of twenty programs, we present an empirical investigation of the clusters identified by these two approaches. In support of the analysis, we consider hybrid cluster types that works at the coarser function-level but is based on the higher-precision statement-level dependences. The three types of clusters are compared based on their slice sets using two clustering metrics. We also perform extensive analysis of the programs to identify linchpin functions – functions primarily responsible for holding a cluster together. Results include evidence that the less expensive, coarser approaches can often be used as effective proxies for the more expensive, finer-grained approaches. Finally, the linchpin analysis shows that linchpin functions can be effectively and automatically identified.

I. Introduction

The need to repair and improve software involves a number of challenging tasks such as impact analysis [1], defect detection [2], software reuse [3], [4], and regression testing [5]. Such tasks are facilitated by source code that is easily separated (e.g., loosely coupled). For example, the complexity of understanding the impact of a change is reduced if only a subset of the code need be considered. Easily separated code also simplifies more complex tasks such as extracting software product lines from legacy applications [6]. In general, the software maintenance and evolution process is aided by separable software.

Unfortunately, a natural and inevitable aspect of all programs are dependences between components (e.g., statements, functions, or classes). A dependence between two program components means that the execution of one component influences the other [7]. Both software engineers and the tools they use must be aware of the connections dependences cause in virtually every software engineering task involving the multiple components.

A decade ago, Binkley and Harman demonstrated that software was often *not* easily separable [8] when they empirically observed that programs often include large clusters of mutually dependent components. The presence of these *dependence clusters* complicates software maintenance and evolution. For example, if the impact of a change involves any part of a cluster then it will involve the entire cluster. Furthermore, because large clusters include much of a program's code, it is very likely that some member of the cluster will be encountered. Thus, clusters, which can encompass 80% or more of a program [8], can severely inhibit the effectiveness of engineers and the tools that support them.

During the ensuing decade, studies have replicated the initial findings with C programs, uncovering clusters in Java codes [9]–[11] and in legacy Cobol systems [12]. Dependence clusters are also known to be detrimental to the software development process where they hinder a variety of activities including maintenance, testing, and comprehension [13]–[17].

While still not well understood, clusters of dependence seem to be an inherent property of source code; thus, current research has focused on understanding their formation and the possibilities for their removal or reduction. One of the challenges in studying dependence clusters is the complexity of the underlying program dependence analysis. The original work used dependences from a program's System Dependence Graph (SDG) [18] whose computation involves solving several difficult whole-program data-flow problems such as determining the modified global variables that result from a procedure call [19] and determining the points-to set for each pointer variable [20].

This complexity raises an interesting question: are there approximations that provide better scalability without significant loss of precision? The answer appears to be "yes." One recent example is *static execute before* (SEB) [21]. This coarser relation is based on function-level control-flow reachability. Preliminary experiments indicated that the SEB approximation works well in practice [21]. Extending this work, this paper makes the following contributions:

a) Cluster Comparison: It presents a more careful and extensive look at the comparison of slice-based [8] dependence clusters and the recently introduced SEB-based [22] dependence clusters. To do so, it considers two hybrid cluster types that are computed at the coarser function-level but based on the higher-precision dependences from the SDG. All three types of clusters are empirically compared using two different clustering metrics.

b) Cluster Identification: This paper also explores the identification of *linchpins* – a single program element (e.g., a statement, a global variable, or a single function) that holds a dependence cluster together. Specifically linchpin functions are investigated. When the dependences of a linchpin function are removed, the cluster(s) of the program vanish. At present, it is

not well understood what makes a particular program element a linchpin, how linchpins can be identified, or when there is a refactoring that removes a linchpin. As a first step, it is useful to be aware of a program's linchpins. The experiments show how linchpin functions can be effectively and automatically identified.

After introducing the types of clusters considered, the bulk of the paper presents an in depth analysis of the clusters identified by each type of cluster and the linchpins found within each. This is followed by a discussion of threats to validity, related work, and finally a summary.

II. Background

The goal of this section is to define the five cluster types studied in the paper. These five include the three mentioned in the introduction plus two additional variations. To do so, we first introduce the five slicing operators upon which the cluster definitions are built. These range from fine-grained statement-level relationship using control and data dependences to the coarser relation based on function-level control-flow reachability. Finally, we define the five cluster types.

A. Program Slicing

This section introduces the five slicing operators used to create five instantiations of the general definition of a dependence cluster. The first four compute (backward) slices as the solution to a reachability problem over a program's *System Dependence Graph* (SDG) [18]. An SDG is comprised of vertices, which essentially represent the statements of the program and two kinds of edges: data dependence edges and control dependence edges. A data dependence connects a definition of a variable with each use of the variable reached by the definition [23]. Control dependence connects a predicate p to a vertex v when p has at least two control-flow-graph successors, one of which can lead to the exit vertex without encountering v and the other always leads eventually to v [23]. Thus p controls the possible future execution of v. When slicing an SDG, a slicing criterion is a vertex from the SDG.

The four SDG-based slicing operators $Slice^{VV}$, $Slice^{VF}$, $Slice^{FV}$, and $Slice^{FF}$ differ on the slicing criteria considered and on the set of elements returned as the result of the slice. The first one, $Slice^{VV}$, is the traditional vertex-level slicing [16] where the slicing criterion is an SDG vertex and the result is a set of vertices, V. With such a slice the criteria is dependent on each of the vertices found in V. The second, $Slice^{VF}$, has the same slicing criteria, a vertex, but produces a set of functions F rather than a set of vertices. In this case, the criteria is dependent on the (entry-point vertices of the) functions in F.

Parallel to the first two, the output of the final two operators, $Slice^{FV}$ and $Slice^{FF}$, is a set of vertices V and a set of functions F, respectively. These two differ in that the slice is taken with respect to an entire function instead of a single SDG vertex. For function f, this is done by taking the union of the slices for each vertex that represents source code from f.

The final slicing operator, $Slice^{SEB}$, is based on the SEB relation [21]. SEB captures a conservative type of dependence on functions that does not require the computation or use of data dependences. Rather it uses only the possible control-flow paths and call-structures inside functions. This approach is more efficient but less accurate than SDG-based slicing, and is defined as follows. For functions f and g, we say that $(g, f) \in SEB$ if and only if it is possible that any part of f is executed before any part of g in any one of the executions of the program.

More formally, SEB is defined as

$$SEB = CALL \cup SEQ \cup RET \cup ID$$

where

$$\left.\begin{array}{l}(f,g) \in CALL \\ (g,f) \in RET\end{array}\right\} \iff f \text{ (indirectly) calls } g \text{ (or, } g \text{ (indirectly) returns into } f)$$

$$(f,g) \in SEQ \iff \exists h : f \text{ (indirectly) returns into } h, \text{ and there is a control-flow path to where } h \text{ (indirectly) calls } g$$

$$(f,g) \in ID \iff f = g$$

SEB is computed using a lightweight program representation called the *Interprocedural Component Control Flow Graph* (*ICCFG*) [21], which is composed of individual Component Control Flow Graphs (CCFGs) for each procedure of the program. Each CCFG represents a procedure's intraprocedural Control Flow Graph (CFG) [24] but only call site nodes and corresponding flow edges are retained. The ICCFG consists of the CCFGs of each procedure connected by call edges from each call site (in a component) to the entry node of the called function. A reachability algorithm is then used on the ICCFG, similar to the SDG reachability algorithm, to compute a slice. For any function f, $Slice^{SEB}$ on the criterion f produces the set of functions on which f depends (i.e., the functions that are predecessors of f according to the SEB relation).

B. Dependence Clusters

Having defined the five slicing operators, the next step is to introduce the five dependence cluster kinds studied in the empirical analysis. The term *Dependence Cluster* was introduced by Binkley and Harman [8] as a set of program elements that mutually depend upon one another. Dependence Clusters are formalized in terms of mutually dependent sets:

Definition 1 (Mutually-Dependent Set and Cluster): A *mutually-dependent set (MDS)* is a set of elements, E, such that $\forall x, y \in E : x$ depends on y. A *dependence cluster* is a maximal MDS; thus, it is an MDS not properly contained within another MDS.

Definition 1 implicitly builds on an underlying *depends-on* relation. Ideally, this relation would precisely capture the impact, influence, and dependence between program elements. Unfortunately, such a relation is not computable [25]. However, it can be approximated using program slicing [8]. Empirically it has been shown that it is sufficient to test if two elements have the same *slice size* as done in the following definition [8]:

Definition 2 (Same-Slice-Size MDS and Cluster): Given a slicing operator $\mathcal{S}$, a Same-Slice-Size MDS is a set of elements,

E, such that $\forall x, y \in E : |s(x)| = |s(y)|$. A *same-slice-size dependence cluster* is a same-slice-size MDS contained within no other same-slice-size MDS.

Instantiating in Definition 2, each of the five slicing operators produces the following same-slice-size cluster types shown in Table I. For ease of reading, the same-slice-size clusters defined in Definition 2 are referred to simply as "clusters" parameterised by the above notations in the rest of the paper.

TABLE I. SLICING OPERATORS AND CLUSTER TYPES

Slicing Operator (s)	Cluster Type (c)
$Slice^{VV}$	c^{VV}
$Slice^{VF}$	c^{VF}
$Slice^{FV}$	c^{FV}
$Slice^{FF}$	c^{FF}
$Slice^{SEB}$	c^{SEB}

III. EMPIRICAL STUDY

The five slicing operators are related by containment: given a criteria c, $Slice^{VV}(c) \subseteq Slice^{VF}(c)$ and $Slice^{FV}(c) \subseteq Slice^{FF}(c) \subseteq Slice^{SEB}(c)$ (provided that vertex-level slicing results are aggregated to function-level). However, these subsumptions may not translate to the corresponding dependence cluster types. Similarly, the existence of a linchpin in one cluster type does not necessarily imply a linchpin in another type.

The experiment presented in this section empirically explores the relationships between the clusters produced by the different slicing operators and the corresponding linchpins. We first compare the underlying slices and the clusters by objective measurements (*RQ1*). Then, we use a manual classification of cluster structures and associated linchpins to investigate the clusters and their breaking (*RQ2*). Finally, we consider how two metrics, *AREA* and *REGX* [21], perform at cluster and linchpin analysis with respect to a manual classification (*RQ3*). Based on the results we verify to what extent a less expensive dependence analysis, such as SEB, can be used as a proxy for a more precise and more expensive analysis with respect to dependence clustering. In greater detail, we frame the investigation using the following research questions:

- *RQ1* How well can SEB-based cluster analysis be used as a proxy for the more expensive slice-based cluster (vertex-level and function-level) analysis?
- *RQ1.1* How do the slice sets compare to SEB sets?
- *RQ1.2* What are the differences between cluster structures produced by slice sets and SEB sets?
- *RQ1.3* How well do the different clusterization metrics perform for the clusters?
- *RQ1.4* How does vertex-level clustering compare to function-level clustering?
- *RQ2* What patterns of clusterization (slice and SEB-based) indicate the presence of linchpin functions?
- *RQ3* How well could the clusterization metrics be used for the identification of linchpins and is there any difference in terms of different slice types?
- *RQ3.1* How different are rankings produced by different metrics used to characterize clusterization?
- *RQ3.2* What is the Average Precision of metric-based cluster identification with respect to the manually identified linchpins?

TABLE II. SUBJECT PROGRAMS

Subject	C Files	LOC	SDG Vertex Count	Function Count	Description
acct-6.3.2	26	6,278	6,032	100	Process monitoring
barcode-0.98	16	4,204	5,574	67	Barcode generator
bc-1.05	20	5,077	5,361	103	Calculator
byacc-1.9	13	6,626	9,669	178	Parser generator
compress-4.0	3	1,937	1,010	24	File compressor
copia	1	1,168	3,824	242	ESA signal processing
ctags-5.0	49	16,015	20,251	530	C tagging
diffutils-2.7	25	15,775	15,698	218	File differencing
ed-1.6	16	9,765	14,334	98	Line text editor
epwic-1.0	30	7,943	9,020	126	Wavelet image encoder
flex-2.4.7	18	14,336	10,832	153	Lexical analyzer
ftpd-1.0.29	49	18,804	19,368	263	File transfer daemon
gnuchess-5.08	38	16,866	15,145	267	Chess player
go-3.0.0	36	29,629	30,679	372	Go player
indent-2.2.9	24	11,539	9,819	116	Text formatter
sudoku-1.11	1	1,983	2,362	38	Sudoku player
time-1.7	9	1,315	988	13	CPU resource meter
userv-0.95.0	17	7,908	13,805	247	Access control
wdiff-0.5	6	1,862	1,554	29	Diff front end
wu-ftpd-2.5.0	8	7,674	9,309	74	File transfer daemon
Sum	405	186,704	204,634	3,258	
Average	20	9,335	10,232	163	

A. Environment and Experiment Process

For the different slice types to be maximally comparable we used a common analysis tool setup to the last point where the different slice computation methods diverge. Particularly, we used GrammaTech's CodeSurfer [26] to compute the common internal program representation, the System Dependence Graph (SDG). The SDG was used to compute the four SDG-based slice types using the two pass traversal algorithm by Horwitz et al. [18]. To compute $Slice^{SEB}$, we created the ICCFG from the same SDGs and applied a reachability algorithm [21].

We computed dependence clusters, clusterization metrics, and other program parameters based on the slices. To aid linchpin determination we used the brute-force method first employed by Binkley and Harman [14]. For each slice type, we excluded each potential linchpin function one by one, and then recompute the dependence clusters and the clusterization metrics. Big relative changes in metric value is taken as an indicator of the presence of a linchpin.

We then used the clustering information from both the unreduced programs and the versions where potential linchpins had been removed to create input data for the manual and automatic characterization of the programs in the subsequent phases of our study. We used *Monotone Slice-size Graphs* (MSGs) [8] and the relative changes in the clusterization metrics for this purpose.

B. Subjects

Table II lists the subject programs used in the experiments. Columns 2–6 provide basic metrics and a description of the programs. The programs cover various domains and are taken from previous studies on dependence clusters. The source codes for the subject programs are available online (see Section VII).

C. Dependence Sets

The first experiment seeks to verify the relationship among the different slice types in terms of their relative precision and recall (*RQ1.1*). By definition, slice types with the same kind of

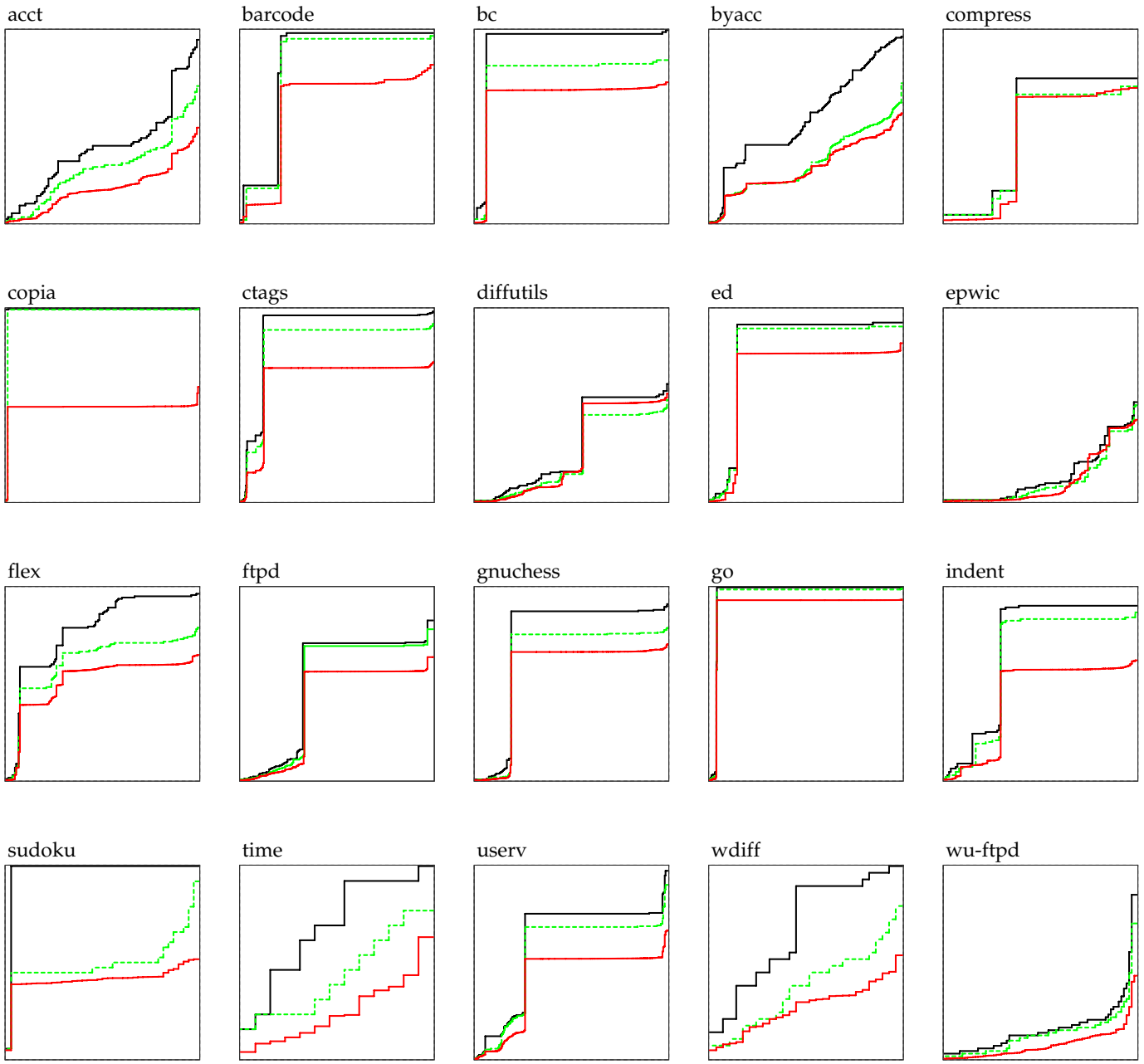

Fig. 1. Subject program MSGs. The lowest solid red line shows $Slice^{FV}$, the middle dotted green line shows $Slice^{FF}$, and the upper black line shows $Slice^{SEB}$. Both axes on the graphs follow the number of program elements (vertices or functions, depending on the slice type) relative to all elements.

criteria (vertices or functions) differ only in how we determine the elements of the slice since they are computed by the same underlying algorithm. Hence $Slice^{VV}(c) \subseteq Slice^{VF}(c)$ and $Slice^{FV}(c) \subseteq Slice^{FF}(c)$ for any slicing criterion, c (with vertices aggregated to functions in the results). $Slice^{FF}(c) \subseteq Slice^{SEB}(c)$ should also hold due to the definition of SEB relation and the fact that we used the same underlying program representation in all cases. We empirically verified these properties, which hold for all possible criteria in all programs.

In earlier work [21], we showed that the SEB relation is less precise than the SDG-based slices, but only to a small amount. This suggests that SEB may be a viable alternative for the analysis of large and complex systems (for which the traditional slicing algorithms are difficult to scale). These early experiments were done using a different set of tools and subject programs. In the present work, we replicate and extend the comparison to include all five slice types.

As expected, slice sizes were comparable with the different slice types. This experiment confirmed our earlier findings that the SEB slices are not much larger than $Slice^{FF}$ slices: the difference is between 3%–48%, the average being 11%, and the outliers are all in the very small programs. The difference

TABLE III. CLUSTERIZATION METRICS AND DEPENDENCE CLUSTER CLASSIFICATION

Subject	Clusterization level			*AREA*			*REGX*		
	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$
acct	none	none	small	0.18	0.28	0.39	0.02	0.19	0.52
barcode	high	high	high	0.60	0.79	0.82	0.00	0.91	0.94
bc	high	high	high	0.65	0.77	0.93	0.04	0.77	0.92
byacc	small	small	medium	0.29	0.31	0.55	0.09	0.27	0.57
compress	small	small	medium	0.43	0.45	0.50	0.22	0.57	0.70
copia	huge	huge	huge	0.49	0.99	1.00	0.94	0.99	1.00
ctags	large	large	large	0.62	0.80	0.88	0.02	0.84	0.93
diffutils	medium	medium	medium	0.26	0.24	0.29	0.10	0.59	0.75
ed	large	large	large	0.66	0.78	0.80	0.08	0.88	0.90
epwic	none	none	small	0.11	0.11	0.13	0.30	0.27	0.37
flex	small	medium	medium	0.51	0.61	0.78	0.05	0.56	0.76
ftpd	medium	medium	medium	0.39	0.49	0.51	0.03	0.69	0.74
gnuchess	large	large	large	0.54	0.62	0.72	0.13	0.70	0.82
go	huge	huge	huge	0.90	0.95	0.96	0.01	0.96	0.98
indent	large	large	large	0.43	0.62	0.68	0.00	0.78	0.88
sudoku	small	large	huge	0.41	0.50	0.98	0.00	0.41	0.97
time	none	none	none	0.25	0.43	0.69	0.00	0.08	0.50
userv	large	large	large	0.41	0.54	0.60	0.10	0.72	0.85
wdiff	none	none	medium	0.26	0.37	0.68	0.07	0.07	0.68
wu-ftpd	none	none	none	0.07	0.13	0.16	0.03	0.22	0.21
Average				0.4230	0.539	0.6525	0.1115	0.5735	0.7495

between the two function-level slices $Slice^{FF}$ and $Slice^{FV}$ are also very similar: on average it was 12%.

The other interesting observation from the data is that the difference between pairs of slices that differ in the criteria used, but not the counting granularity ($Slice^{VV}$ vs. $Slice^{FV}$ and $Slice^{VF}$ vs. $Slice^{FF}$, respectively) was small, about 2% on average for both cases. Based on this observation and to reduce the complexity of further analyses we limit our further investigations to the three slice types that take functions as their criteria, namely $Slice^{FF}$, $Slice^{FV}$ and $Slice^{SEB}$. We expect that this simplification does not materially influence our findings.

D. Dependence Clusters

To answer *RQ1.2*, we performed the following study. A proven approach to investigate cluster structures is to use the Monotone Slice-size Graphs (MSGs). An MSG for a program is a graphical representation of all the program's slices drawn in monotonically increasing order along the x-axis from left to right. An MSG allows easy visual identification of cluster structures as horizontal plateaus in the graph. Figure 1 shows MSGs for the original (unreduced) versions of the investigated programs. In this work, we show a combined MSG in which all three slice types are shown on the same graph. Note, that since these slice types share the same slicing criteria, the x-axis of the MSG is common, but due to the different granularity of the slice elements, the slices of $Slice^{FV}$ were scaled on the y-axis. The MSGs enable a manual identification of various clusterization patterns. The first observation one can make about the graphs in Figure 1 is that the three slice types typically produce MSGs with very similar shape. In only a few cases is there a significant difference (e.g., sudoku). A more careful investigation revealed some subtle differences though, which we will address in a later section.

At this point we manually classified each program according to its clusterization level. Following the practice used in previous research, we used the five-level Likert scale ("none", "small", "medium", "large" and "huge"), which allowed us a systematic and relatively fine grained analysis of the cluster structures. The result of the assessment is shown in the columns 2–4 of Table III. As mentioned, clusterization was quite similar in many cases, but there are significant differences as well. For instance, while slice-based clusters for wdiff cannot be identified, a clearly visible cluster can be found with SEB.

Note, that it is to be expected that the lines do not cross in the combined MSGs because of the underlying slice-set containment relationship; however, in a few cases we can observe such phenomenon. In each such case, the crossing is an artifact of the scaling used on the y-axis.

E. Clustering Metrics

To characterize programs with dependence cluster formations in an objective way, some objective measurements are required (this addresses *RQ1.3*). The traditional approach for this purpose is to use the area under MSG, that is the sum of sizes of all slices [27]. This metric, used as an indicator of relative change after the removal of a program element may be used to detect linchpins. However, it has the drawback that it does not take into account the structure of cluster formations, so it produces false positives. For example, when there is a uniform reduction in slice size, which has no effect on the cluster structures. Thus it may be misleading. In previous work [22] we experimented with several other measurements that in certain situations are better indicators of clusterization. In this paper we will rely on two clusterization metrics: the area under MSG, denoted *AREA*, and the regularity metric *REGX*, defined in the mentioned article.

We want the two metrics to be comparable for a given program, so we normalize them to the interval $[0, 1]$, where 0 means no clusterization and 1 indicates maximum clusterization. For the *AREA* metric, we normalize using the maximum possible area. More formally, for a given program P and slice type $\mathcal{S}$:

TABLE IV. MANUAL CLASSIFICATION OF LINCHPINS

	Number of Gold-Standard linchpins		Number of Silver-Standard linchpins		Most obvious linchpin(s)
Subject	$\mathcal{C}^{\mathcal{FV}'}$ & $\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$	$\mathcal{C}^{\mathcal{FV}'}$ & $\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$	All cluster types
acct	–	–	–	2	
barcode	1	1	1	1	`Barcode_Encode`
bc	1	1	1	1	`dc_func`
byacc	–	–	3	3	
compress	–	–	3	3	`spec_select_action`
copia	2	2	2	2	`seleziona, scegli`
ctags	–	–	2	4	`createTagsForFile`
diffutils	–	–	2	3	
ed	–	1	1	1	`exec_command`
epwic	–	–	–	–	
flex	–	–	1	1	
ftpd	1	1	3	3	`parser`
gnuchess	2	2	3	3	`main, parse_input`
go	–	–	4	4	`get_reasons_for_moves`
indent	2	2	3	3	`indent_main_loop`
sudoku	–	–	1	1	
time	–	–	–	–	
userv	–	–	4	4	
wdiff	–	–	–	2	
wu-ftpd	–	–	–	–	

$$AREA = \frac{1}{|C| \cdot |E|} \sum_{c \in C} |\mathcal{S}(c)| \ ,$$

where C is a set of slicing criteria, $\mathcal{S}(c)$ denotes the slice on criterion $c \in C$, and E is the set of program elements that are possible members of a slice. The *AREA* metric is equivalent to the average slice size for a program.

The intuition behind *REGX* is that the number of different slice sets formed when slicing a program captures how "regular" the slices are. If there are many different sets, the program probably does not have dependence clusters, whereas a very low number of different sets potentially indicates a large cluster. *REGX* is normalized over the largest possible number of slice sets and assumes that $|C| > 1$:

$$REGX = \frac{|C| - |D|}{|C| - 1} \ ,$$

where $D = \left\{ \mathcal{S}(c) \,\middle|\, c \in C \right\}$ is the set of unique slices.

In the experiments, elements of C and E are either SDG vertices or program functions, so both *AREA* and *REGX* can be interpreted for all slice types by substituting the respective criteria and program element sets. Table III provides the *AREA* and *REGX* metrics for each unreduced program and the three investigated slice types. These metrics serve as a baseline when searching for linchpin functions. Comparing the manual cluster classifications from Table IV to the metric values, generally it is not obvious which metric best reflects the level of clusterization. There are obvious cases such as programs copia and go, where large dependence clusters can be easily identified by large *AREA* values. However, there are cases, such as wdiff, byacc, and ctags, where *AREA* alone is not enough to determine clusterization. Here, *REGX* could provide additional information about the regularity of different set sizes. For instance, in the case of wdiff we observe a medium SEB-based cluster, which is reflected by the relative high value of *REGX* while the same metric for the other two slice types is low, indicating the absence of clusters. Another example is ctags, which is classified as highly clustered. In this case the *REGX* metrics are bigger than *AREA* metrics, which indicates that in these cases *REGX* may be a better indicator than *AREA*.

F. Vertex-Level vs. Function-Level Analysis

RQ1.4 deals with the difference between vertex-level and function-level clustering. To summarize our findings from previous sections, the different clustering levels generally tend to show similar behavior. This is especially true for programs where there are large to huge dependence clusters. In these cases the difference is merely in the average slice size but the clusterization structures are very similar. Also, with programs which are obviously clusterless, this property is found by all slice types (and the corresponding metrics). There are only a few cases where there is observable difference between clusterization patterns at different levels (examples include bc, sudoku, wdiff, byacc), which suggests at this point (together with the overall small difference in the slice sets themselves) that function-level analysis is indeed a good approximation to vertex-level analysis.

G. Manual Linchpin Identification

Our next set of experiments dealt with *RQ2*, the search for patterns in the clusterization. As a first step, we wanted to find out if we can identify linchpin functions in the programs by manually investigating the changing of cluster formations as we apply the brute-force method of linchpin identification. More precisely, for each program we performed a series of code analyses, slice and cluster computations by removing one function at a time from a program. This way we got a series of cluster structures represented in form of MSG graphs, which we investigated visually. To be able to handle the large number of graphs to look at we ordered them according to the different associated clusterization metrics. We thus got 2×3 different orderings which we interchangeably used, but most of the times ordering by *AREA* for $\mathcal{C}^{\mathcal{SEB}}$ was most helpful. The graphical representation of metric changes in form of combined 3D bar charts as exemplified with Figure 4 were a useful aid as well.

To make the classification of linchpins more structured we applied a 5-level scheme: 5 (cluster broken), 4 (almost a 5),

3 (a bit of breaking is evident), 2 (almost a 1) and 1 (clearly no breaking or just a drop in slice size). For each program we then identified all potential linchpin functions (separately for each slice type), resulting in the total of 69 functions for the whole subject set. We then further classified the linchpins into what we call a "gold standard" and a "silver standard," which form the basis for further statistical analysis. These two classifications capture the author's intuition as to the level of clusterization with the gold standard being more rigorous and representing very obvious cases, while silver standard is more relaxed (and is a superset of the gold standard). We decided not to include any of linchpin candidates falling into categories 1–2, thus sliver standard included candidates of categories 3–5, and gold standard consisted of only category 5 linchpins. Note, that in this step we deliberately did not use any predefined thresholds of the metric values to decide on the categories, we relied on visual inspection only.

Four columns in the middle of Table IV contain the number of manually identified linchpins organized by gold and silver standard and by the different slice types. It can be observed that $\mathcal{C}^{\mathcal{FV}}$ and $\mathcal{C}^{\mathcal{FF}}$ did not produce any differences, while in several cases $\mathcal{C}^{\mathcal{SEB}}$ clusters resulted in different linchpins (typically more could be identified with this slice type). In the last column of the table we listed the most obvious linchpin functions, the ones that can be identified with all cluster types, and are typically parts of the gold standard.

In Figures 2 and 3 we present examples of gold and silver standard functions, respectively. If we compare the MSGs shown with their unreduced versions from Figure 1, we can clearly see the effect of linchpin removal: in the case of gold standard it is significant, while it is less pronounced with silver standard. The example, in the case of barcode all slice types produce significant cluster break, however with ed only $\mathcal{C}^{\mathcal{SEB}}$ produces a break, the slice-based clusters do not vanish, they are just reduced. Considering example silver standard reductions (Figure 3), the reduction for go is significant, however, a big cluster remains. Program byacc is the least evident: in fact, with $\mathcal{C}^{\mathcal{FV}}$ and $\mathcal{C}^{\mathcal{FF}}$ different linchpins could be identified than with $\mathcal{C}^{\mathcal{SEB}}$-based ones. The example shows a function that resulted in the slight break of clusters with $\mathcal{C}^{\mathcal{SEB}}$ but slice-based clusters were unaffected. This last example is also a good demonstrator where the drop in *AREA* metric could not indicate a linchpin, while the change in the cluster pattern could be captured by the *REGX* metric.

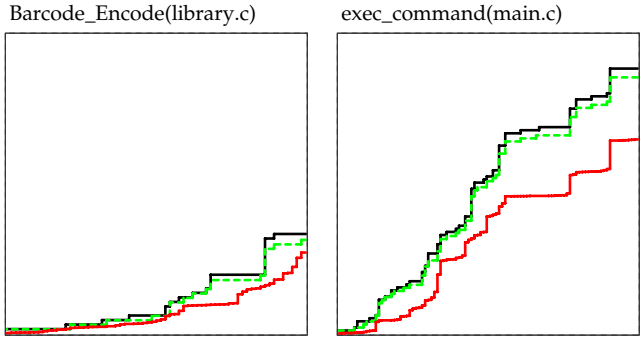

Fig. 2. Gold Standard Patterns. Functions from barcode (left) and ed (right)

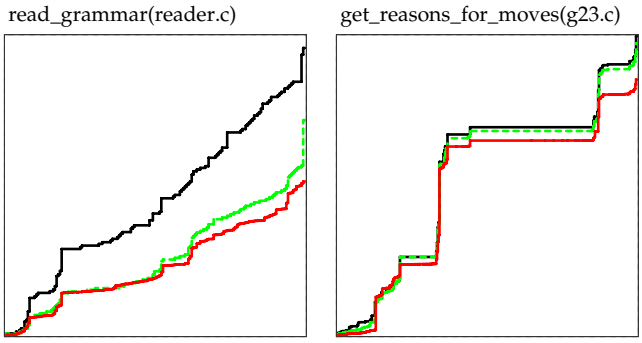

Fig. 3. Silver Standard Patterns. Functions from byacc (left) and go (right)

H. Metric-Based Linchpin Identification

In our final set of experiments our goal was to verify to what extent clusterization metrics can be used to guide the linchpin identification process (*RQ3*). In the previous section we explained how the brute-force method of linchpin identification results in a set of reduced cluster structures and associated metric values. We use these "reduced" metric values in form of their relative change percentage compared to the unreduced ones. Thus we obtain six (potentially) different rankings of functions for each program. Figure 4 shows examples of how different the rankings can be. The series of bars are given in decreasing order of a relative decrease of the metric compared to the unreduced program. Note, that since the rankings are computed individually, the bars at the same rank position may represent different functions. The upper part of the figure shows the results for program bc, where we can clearly observe that the first element in the rank list is much higher than the rest in all three *AREA* metrics and in one of the *REGX* metrics (in this case the first element was the same), which clearly indicates a linchpin. However, the other example in this figure (from program ctags) exemplifies a different pattern. Here, just by looking at the ranks we could not clearly say if the first one or more elements might be linchpins. In fact, in the case of this program it turned out that the ranking according to *REGX* was closer to manual assessment than the *AREA*-based.

To address the research question *RQ3*, we assessed these rankings from two angles: (*RQ3.1*) how do they compare to each other (i.e. is there any significant difference among them), and (*RQ3.2*) how well do the rankings reflect the manual assessment presented in the previous section?

To answer the first question, we compared the different rankings using correlation analysis. We computed Kendall and Spearman correlations between the rank lists for each program and each combination of metrics. Results are shown in Tables V and VI, respectively. Since the two sets of results are very similar we discuss only Kendall in detail.

First thing to observe is that the *AREA*-based rankings are very similar: $\mathcal{C}^{\mathcal{FV}}$ and $\mathcal{C}^{\mathcal{FF}}$ show a correlation of 0.74, and $\mathcal{C}^{\mathcal{SEB}}$ is aligned with the other two as well. Specifically, results for $\mathcal{C}^{\mathcal{SEB}}$ are more strongly correlated with $\mathcal{C}^{\mathcal{FF}}$, which is to be expected because these two slice types are more similar. The *REGX*-based rankings show a different picture. Only the two slice-based metrics are correlated, while $\mathcal{C}^{\mathcal{SEB}}$ shows a negative

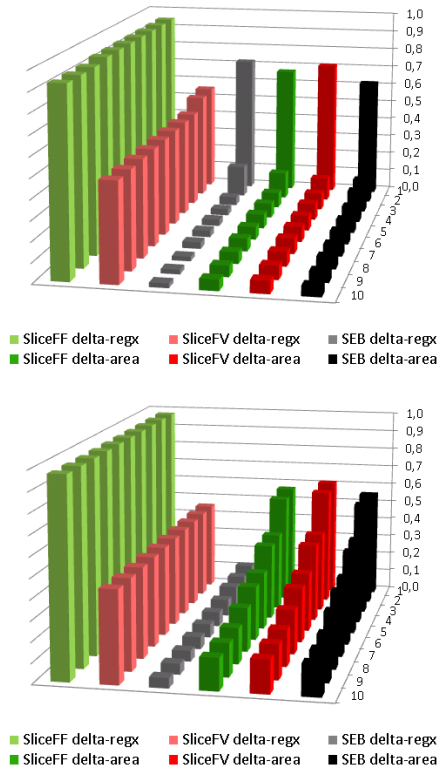

Fig. 4. Linchpin Rankings by Different Metrics (shown are bc and ctags)

TABLE V. KENDALL CORRELATIONS AMONG RANKINGS

		AREA			*REGX*		
		$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$
AREA	$\mathcal{C}^{\mathcal{FV}}$	1.0	0.74	0.6	-0.44	-0.28	0.41
	$\mathcal{C}^{\mathcal{FF}}$	0.74	1.0	0.8	-0.44	-0.3	0.49
	$\mathcal{C}^{\mathcal{SEB}}$	0.6	0.8	1.0	-0.42	-0.3	0.54
REGX	$\mathcal{C}^{\mathcal{FV}}$	-0.44	-0.44	-0.42	1.0	0.32	-0.37
	$\mathcal{C}^{\mathcal{FF}}$	-0.28	-0.3	-0.3	0.32	1.0	-0.28
	$\mathcal{C}^{\mathcal{SEB}}$	0.41	0.49	0.54	-0.37	-0.28	1.0

correlation. Based on earlier observations, where *REGX* metrics performed best with $\mathcal{C}^{\mathcal{SEB}}$, we conclude that the slice-based *REGX* metrics are typically less useful. This can be supported also by looking at the correlation between *AREA* and *REGX* metrics: only in the case of $\mathcal{C}^{\mathcal{SEB}}$ can we observe positive correlation. As noted earlier, *AREA*-based metrics are good predictors in most of the cases of the existence of linchpins, however in less evident cases, *REGX* could be used in addition. This experiment supported this observation and revealed that this is typically the case with $\mathcal{C}^{\mathcal{SEB}}$ slice types.

Finally, we address *RQ3.2* by comparing the rankings to the manually identified linchpins. For this purpose, we used the Average Precision (AP) and Mean Average Precision (MAP)

TABLE VI. SPEARMAN CORRELATIONS AMONG RANKINGS

		AREA			*REGX*		
		$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$
AREA	$\mathcal{C}^{\mathcal{FV}}$	1.0	0.85	0.73	-0.53	-0.34	0.49
	$\mathcal{C}^{\mathcal{FF}}$	0.85	1.0	0.87	-0.51	-0.34	0.55
	$\mathcal{C}^{\mathcal{SEB}}$	0.73	0.87	1.0	-0.48	-0.34	0.59
REGX	$\mathcal{C}^{\mathcal{FV}}$	-0.53	-0.51	-0.48	1.0	0.33	-0.4
	$\mathcal{C}^{\mathcal{FF}}$	-0.34	-0.34	-0.34	0.33	1.0	-0.29
	$\mathcal{C}^{\mathcal{SEB}}$	0.49	0.55	0.59	-0.4	-0.29	1.0

measures, typically used in Information Retrieval for similar purposes [28]. Average Precision is more appropriate for ranked information retrieval than traditional precision and recall, which do not take the ranks into account, because it does not require the selection of an arbitrary set of retrieved documents. In short, AP is precision (at each rank position) at each relevant document (linchpin in our case) averaged over the number of relevant documents, while the MAP value is the mean of the Average Precision values over all sets of queries (programs in our case).

We computed AP values for each combination of program and ranking (determined by the different metrics and slice types), using both the gold and silver standard linchpins as the relevant documents. Table VII shows the Average Precision values for the gold standard linchpins, while Table VIII contains the same for silver standard linchpins.

TABLE VII. GOLD STANDARD AVERAGE PRECISION

	AREA			*REGX*		
Subject	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$
barcode	1.00	1.00	1.00	1.00	1.00	1.00
bc	1.00	1.00	1.00	1.00	1.00	1.00
copia	1.00	1.00	1.00	1.00	0.32	1.00
ctags	1.00	1.00	–	1.00	0.27	–
ed	1.00	1.00	1.00	1.00	0.25	1.00
ftpd	1.00	1.00	1.00	1.00	0.50	1.00
gnuchess	1.00	1.00	1.00	1.00	1.00	1.00
indent	1.00	1.00	1.00	1.00	1.00	0.81
userv	0.50	0.50	–	0.33	0.00	–
MAP	**0.94**	**0.94**	**1.00**	**0.93**	**0.59**	**0.97**

TABLE VIII. SILVER STANDARD AVERAGE PRECISION

	AREA			*REGX*		
Subject	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$	$\mathcal{C}^{\mathcal{FV}}$	$\mathcal{C}^{\mathcal{FF}}$	$\mathcal{C}^{\mathcal{SEB}}$
barcode	1.00	1.00	1.00	1.00	1.00	1.00
bc	1.00	1.00	1.00	1.00	1.00	1.00
byacc	0.37	0.35	0.44	0.81	0.17	0.79
compress	1.00	1.00	1.00	1.00	0.92	1.00
copia	1.00	1.00	1.00	1.00	0.32	1.00
ctags	0.91	0.89	0.89	1.00	0.27	0.58
diffutils	1.00	1.00	0.92	0.83	0.58	1.00
ed	1.00	1.00	1.00	1.00	0.25	1.00
flex	–	–	0.50	–	–	1.00
ftpd	1.00	1.00	1.00	0.81	0.44	0.78
gnuchess	1.00	1.00	1.00	0.87	1.00	0.83
go	–	–	0.89	–	–	0.67
indent	1.00	1.00	1.00	1.00	1.00	0.81
sudoku	–	–	1.00	–	–	1.00
userv	0.50	0.50	0.92	0.33	0.00	0.84
wdiff	–	–	0.50	–	–	1.00
MAP	**0.90**	**0.90**	**0.88**	**0.89**	**0.58**	**0.89**

The most obvious thing to observe from the data is that the metric based ranking performs exceptionally well, especially for gold standard linchpins. In other words, if the existence of linchpins is evident, it can be found by metric-based approach with high success. Particularly, AP is 1.00 for most of the programs in the gold standard category with the *AREA* metrics. Typically, the *REGX* metric is also a good indicator in these cases as it usually "follows" *AREA* in the case of high clusterization and evident linchpins. For silver standard linchpins, the situation is slightly different. MAP values are again very high almost in all cases, but here we can distinguish interesting cases that require further discussion.

Such cases are byacc and ctags, where we can observe that *REGX* outperforms *AREA* in precision of identifying linchpin

functions. A typical linchpin for byacc can be seen in Figure 3. Comparing it to the unreduced MSG, we can observe that there was no significant drop in the average slice sizes, but the cluster structures changed so that the plateau representing a cluster in the C^{SEB} case disappeared. This change could not be captured by *AREA* metrics, only by *REGX*.

The final conclusion is that in the cases where the drop in clusterization metric *AREA* is significant it will indicate the existence of a linchpin function with great probability. However, where the metric changes and the associated ranking are less evident, *REGX* could be an additional source of information.

I. Discussion

As a conclusion to the set of experiments overviewed above, we may say that SEB could be a viable alternative to dependence cluster analysis, because:

1) SEB is a lightweight analysis and is more scalable than SDG-based slicing. This topic has not been covered in the present paper, but previous works (e.g., [21], [22], [29]) showed the viability of the method for very large systems as well.
2) The SEB sets are only slightly less precise than slice sets.
3) Typically, there are no big differences between the cluster structures formed by SDG and SEB-based slices.
4) Linchpins can be identified in both approaches using a metrics based method with *AREA* and *REGX*, and the identified linchpins are aligned in most of the cases in slice and SEB-based analyses.

The other conclusion we may make from the experiments is that *AREA* is a good indicator of clusterization in many cases, especially where the average slice sets are big as well. However, *REGX* may be called upon in less evident situations, which can highlight subtle differences in cluster structures.

IV. Threats to Validity

Although we chose a set of twenty open-source and industrial programs of various sizes and from different domains, external threat arises from the possibility that the programs selected are not representative of programs in general, causing uncertainty of the generality of the conclusions. The manual identification of linchpin functions is subjective raising threats to validity of the metrics-based linchpin identification results. The other possible threat to construct validity arises from the potential faults in the slicers. A mature and widely used slicing tool (CodeSurfer) and slicing algorithm implementations used in previous research were used to mitigate this concern.

V. Related Work

Binkley and Harman [8] introduced the notion of dependence clusters and employed program slicing at the vertex-level to identify them. Harman et al. [16] later extended this initial study with a large-scale study of 45 programs and gave clustering definitions based on the direction of dependence (backward/forward). They found that vertex-level slice-based clusters were a common phenomenon in systems and empirically identified patterns of clustering. In a recent work, Islam et al. [30] introduced coherent dependence clusters, which combine the backward and forward slice-based dependence clusters, and then used them to identify logical functionality of the program. Recently, Beszédes et al. [22] introduced the instantiation of dependence clusters using function-level program dependences computed based on the Static Execute Before (SEB) relationship [21].

There has been much interest in dependence clusters and clusters have been identified in various programming languages and systems [12], [13], [21]. Jiang et al. [31] proposed the use of search-based program slicing to identify dependence structures in programs. Another interesting approach to locate interrelated program elements is based on applying community structure analysis on software dependence graphs [32], [33].

Lehnert [1] has considered the relationship between dependence clusters and impact analysis and found that clustering can be used to determine the ripple-effect during software maintenance. Beszédes et al. [9] looked at the relationship between SEB dependencies, and software maintenance and found that SEB can be used to identify hidden dependencies and thus help in many maintenance tasks, including change propagation and regression testing. Subsequent work [21] identified a connection between the SEB-based dependence clusters and performance of change impact analysis.

The current view is that large dependence clusters hinder many different software engineering activities, including impact analysis, maintenance, program comprehension and software testing [13], [16], [17]. In light of this view, this paper compares two approaches to clustering. We also consider two hybrid cluster definitions, which conceptually lie between the levels of abstraction offered by the previously proposed types.

VI. Summary and Future work

This paper studies the relation between several kinds of dependence-clusters including two existing kinds and a previously unstudied hybrids. This is done using a general framework for defining dependence clusters, which it instantiated using five different slicing operators. In summary, the coarser approximation works well and thus extends dependence cluster and linchpin analysis to a wider range of programs. Thus, our research expands the applicability of dependence cluster analysis in software engineering practice.

Looking forward, this line of research is promising. Future projects will consider finding more efficient linchpin identification methods, such as heuristic algorithms. Another interesting topic is to consider sets of potential linchpins rather than single linchpin functions. Finally, we could work towards forming a better understanding of the effects of dependence clusters on every day software engineering. To this end, we plan to empirically investigate the relationship between dependence clusters and software quality.

VII. Online Dataset

The source codes for the subject programs and animated graphics for linchpin functions can be found at `www.cs.ucl.ac.uk/external/s.islam/archives.html#icsme15`.

References

[1] S. Lehnert, "A review of software change impact analysis," Ilmenau University of Technology, Tech. Rep., 2011.

[2] T. Oyetoyan, D. Cruzes, and R. Conradi, "Transition and defect patterns of components in dependency cycles during softw are evolution," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2 014 Software Evolution Week - IEEE Conference on*, Feb 2014, pp. 283–292.

[3] A. Cimitile, A. De Lucia, and M. Munro, "Identifying reusable functions using specification driven program slicing: a case study," in *Proceedings of the IEEE International Conference on Software Maintenance*. Los Alamitos, California, USA: IEEE Computer Society Press, 1995, pp. 124–133.

[4] ——, "A specification driven slicing process for identifying reusable functions," *Software Maintenance: Research and Practice*, vol. 8, pp. 145–178, 1996.

[5] S. Black, S. Counsell, T. Hall, and P. Wernick, "Using program slicing to identify faults in software," in *Beyond Program Slicing*, ser. Dagstuhl Seminar Proceedings, no. 05451. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[6] M. Valente, V. Borges, and L. Passos, "A semi-automatic approach for extracting software product lines," *Software Engineering, IEEE Transactions on*, vol. 38, no. 4, pp. 737–754, July 2012.

[7] 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*, vol. 19, no. 5, pp. 177–184, 1984.

[8] D. Binkley and M. Harman, "Locating dependence clusters and dependence pollution," in *21st IEEE International Conference on Software Maintenance*, 2005, pp. 177–186.

[9] Á. Beszédes, T. Gergely, J. Jász, G. Toth, T. Gyimóthy, and V. Rajlich, "Computation of static execute after relation with applications to software maintenance," in *23rd IEEE International Conference on Software Maintenance*. IEEE Computer Society Press, October 2007, pp. 295–304.

[10] L. Savernik, "Entwicklung eines automatischen Verfahrens zur Auflösung statischer zyklischer Abhängigkeiten in Softwaresystemen (in german)," in *Software Engineering 2007 – Beiträge zu den Workshops*, ser. LNI, vol. 106. GI, 2007, pp. 357–360.

[11] A. Szegedi, T. Gergely, Á. Beszédes, T. Gyimóthy, and G. Tóth, "Verifying the concept of union slices on Java programs," in *11th European Conference on Software Maintenance and Reengineering*, 2007, pp. 233–242.

[12] Á. Hajnal and I. Forgács, "A demand-driven approach to slicing legacy COBOL systems," *Journal of Software: Evolution and Process*, vol. 24, no. 1, pp. 67–82, 2011.

[13] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Proceedings of the 33rd ACM SIGSOFT International Conference on Software Engineering (ICSE)*, 2011, pp. 746–765.

[14] D. Binkley and M. Harman, "Identifying 'linchpin vertices' that cause large dependence clusters," in *9th International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*, Edmonton, Alberta, Canada, 20th-21st September 2009, pp. 89–98.

[15] 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*, vol. 83, no. 1, pp. 96–107, 2010.

[16] M. Harman, D. Binkley, K. Gallagher, N. Gold, and J. Krinke, "Dependence clusters in source code," *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 1, pp. 1–33, Nov. 2009.

[17] S. Black, S. Counsell, T. Hall, and D. Bowes, "Fault analysis in OSS based on program slicing metrics," in *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE Computer Society, 2009, pp. 3–10.

[18] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–61, 1990.

[19] J. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables," in *Sixth ACM Symposium on Principles of Programming Languages, (San Antonio, TX, Jan. 29-31, 1979*, 1979, pp. 29–41.

[20] W. Landi and B. G. Ryder, "Pointer-induced aliasing: A problem classification," in *In Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*. Orlando, FL: ACM Press, Jan. 1991, pp. 93–103.

[21] J. Jász, A. Beszédes, T. Gyimóthy, and V. Rajlich, "Static Execute After/Before as a replacement of traditional software dependencies," in *IEEE International Conference on Software Maintenance*, 2008, pp. 137–146.

[22] Á. Beszédes, L. Schrettner, B. Csaba, T. Gergely, J. Jász, and T. Gyimóthy, "Empirical investigation of SEA-based dependence cluster properties," *Science of Computer Programming*, vol. 105, pp. 3 – 25, 2015, available online at http://dx.doi.org/10.1016/j.scico.2014.09.010.

[23] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[24] M. S. Hecht, *Flow Analysis of Computer Programs*. New York, NY, USA: Elsevier Science Inc., 1977.

[25] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[26] "Homepage of GrammaTech's CodeSurfer," http://www.grammatech.com/research/technologies/codesurfer, GrammaTech, Inc., 2013, last visited: 2013-05-08.

[27] D. Binkley and M. Harman, "Locating dependence clusters and dependence pollution," in *21st IEEE International Conference on Software Maintenance*. IEEE Computer Society Press, 2005, pp. 177–186.

[28] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[29] L. Schrettner, J. Jász, T. Gergely, Á. Beszédes, and T. Gyimóthy, "Impact analysis in the presence of dependence clusters using Static Execute After in WebKit," in *Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'12)*, Sep. 2012, pp. 24–33.

[30] S. Islam, J. Krinke, D. Binkley, and M. Harman, "Coherent clusters in source code," *Journal of Systems and Software*, vol. 88, pp. 1 – 24, 2013, available online at http://dx.doi.org/10.1016/j.jss.2013.07.040.

[31] T. Jiang, N. Gold, M. Harman, and Z. Li, "Locating dependence structures using search-based slicing," *Information and Software Technology*, vol. 50, no. 12, pp. 1189–1209, 2008.

[32] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the Bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 1–16, 2006.

[33] J. Hamilton and S. Danicic, "Dependence communities in source code," in *Proceedings of the 28th International Conference on Software Maintenance (ICSM'12), Early Research Achievements track*, 2012, pp. 579–582.