Empirical Investigation of SEA-Based Dependence Cluster Properties

Árpád Beszédes*, Lajos Schrettner*, Béla Csaba†, Tamás Gergely*, Judit Jász* and Tibor Gyimóthy*

*Department of Software Engineering, University of Szeged, Hungary
E-mail: {beszedes, schrettner, gertom, jasy, gyimothy}@inf.u-szeged.hu

†Department of Set Theory and Mathematical Logic, University of Szeged, Hungary
E-mail: bcsaba@math.u-szeged.hu

***Abstract*—Dependence clusters are (maximal) groups of source code entities that each depend on the other according to some dependence relation. Such clusters are generally seen as detrimental to many software engineering activities, but their formation and overall structure are not well understood yet. In a set of subject programs from moderate to large sizes, we observed frequent occurrence of dependence clusters using Static Execute After (SEA) dependences (SEA is a conservative yet efficiently computable dependence relation on program procedures). We identified potential linchpins inside the clusters; these are procedures that can primarily be made responsible for keeping the cluster together. Furthermore, we found that as the size of the system increases, it is more likely that multiple procedures are jointly responsible as sets of linchpins. We also give a heuristic method based on structural metrics for locating possible linchpins as their exact identification is unfeasible in practice, and presently there are no better ways than the brute-force method. We defined novel metrics and comparison methods to be able to demonstrate clusters of different sizes in programs.**

***Index Terms*—Source code dependence analysis, dependence clusters, linchpins and linchpin sets, Static Execute After.**

I. Introduction

Dependences in computer programs are natural and inevitable. We can talk about dependences among any kind of artifacts such as requirements, design elements, program code or test cases, but dependences within the source code capture the physical structure as implemented best. A dependence between two program elements (*e. g.* statements or procedures) basically means that the execution of one element can influence that of the other, hence the software engineer should be aware of this connection in virtually any software engineering task involving the two elements. One of the fundamental tasks of program analysis is to deal with source code entities and the dependences between them [1].

Dependences cannot be avoided, but they do not always reflect the original complexity of the problem. Sometimes unnecessary complexity is injected into the implementation, which may cause significant problems. A relatively new research area explores *dependence clusters* in program code, which are defined as maximal sets of program elements that each depend on the other [2]. The current view is that large dependence clusters are detrimental to the software development process; in particular, they hinder many different activities including maintenance, testing and comprehension [3], [4], [5], [6], [7]. The primary problem is that in any dependence-related examination, encountering any member of a cluster forces us to consider all other cluster members. If large clusters covering much of the program code exist in a system, then it is very likely that one cluster member is encountered and consequently a large portion of the program code should be enumerated eventually.

978-1-4673-5739-5/13/$31.00 © 2013 IEEE

The root causes of this phenomenon are not well understood yet; it seems to be an inherent property of program code dependence relationships. As apparently dependence clusters cannot be easily avoided in the majority of cases, research should be focused on understanding the causes for the formation of clusters, and the possibilities for their removal or reduction. Previous work revealed that in many cases a highly focused part of the software can be deemed responsible for the formation of dependence clusters [4], [5]. Namely, program elements called *linchpins* are seen as central in terms of dependence relations, and are often holding together the whole program. If the linchpin is ignored when following dependences, clusters will vanish, or at least decrease considerably.

Of course, it is useful if one is aware of such linchpins, let alone be able to remove them by refactoring the program. However, currently even the first step (identifying linchpins) is largely an unexplored area. We still do not understand fully what makes a particular program point a linchpin, how they can be identified, or whether there is always a single element to be made responsible in the first place. The possibilities for linchpin removal by program refactoring are even harder to assess. Sometimes, dependence clusters are avoidable because they actually introduce unnecessary complexity to the implementation; this is what Binkley *et al.* call "dependence pollution" [2]. In such cases the program can be refactored using reasonable effort, but this is not always the case.

In this work, we present the results of our empirical investigation of dependence clusters in a range of programs of moderate (up to 200 kLOC) and large sizes. In the latter category we investigated two industrial size open source

software systems, the GCC compiler [8] and the WebKit web browser engine [9], each consisting of over a million LOC.

We are dealing with procedure-level program dependences computed using the *Static Execute After (SEA)* approach (on C/C++ functions and methods). The SEA relation between two procedures is a conservative type of dependence that takes into account the possible control-flow paths and call-structures in the program elements [10]. SEA-based dependences can be used, among others, in software change impact analysis. The main advantage of SEA is that it achieves acceptable accuracy, yet can efficiently be computed even for large systems of millions of LOC. We computed SEA-based dependences of all procedures in our subject programs and investigated the resulting dependence clusters in terms of their frequency of occurrence and by identifying potential linchpins in them. We summarize our findings as follows:

- We introduce the term *clusterization* to indicate the extent programs exhibit dependence clustering, and define novel metrics that indicate this property quantitatively.
- We computed SEA-based dependence clusters for realistic size programs. Among the moderate-size ones there were many clusters, but only one of the big programs included significant clusters, which is an interesting result.
- We were able to identify linchpin elements in most of the clusters using a naïve approach that enumerates all possibilities. In many cases however, especially with the big programs, it is not to be expected that only one program element (procedure) is responsible for the formation of clusters.
- We give a heuristic method based on local procedure metric for linchpin approximation. We found that the number of outgoing invocations from a procedure was quite a good estimator for linchpins.

The rest of the paper is organized as follows. Sections II and III provide relevant background information about the motivation, related work and our experimental environment. Cluster identification is discussed in Section IV, while the topics related to linchpin determination are given in Section V. Section VI discusses threats to validity, and finally we conclude in Section VII.

II. Background, motivation and goals

The phenomenon of *dependence clusters* was first described by Binkley and Harman in 2005 [2] based on program slices and Program Dependence Graphs [11], [12]. Initially, they were defined as maximal sets of statements that all have the same backward slice, which also means that the elements of a dependence cluster each depend on the other. The notion of dependence clusters can be generalized to other kinds of dependence types and different program elements at various granularity. Furthermore, it seems that dependence clusters are independent of the programming language and the type of the system [3], [13], [14].

The current view is that large dependence clusters hinder many different software engineering activities, including impact analysis, maintenance, program comprehension and software testing [3], [6], [7]. It has been suggested that large dependence clusters leading to "dependence pollution" should be refactored [7], [2], but for such opportunities the identification of the dependence cluster causes is essential. Specifically, the identification and possible removal of *linchpins*, the directly responsible program elements, is an active research area. Virtually, the only existing approach to identify linchpins is based on a brute-force method that tries all possibilities. Binkley and Harman predict in their work [4] that it will be possible to pinpoint certain program elements that cannot be identified as linchpins, hence the search could be optimized this way. We employ heuristic methods to identify linchpins, and we are not aware of any previous work that used a similar approach. Global variables can play special role in the formation of dependence clusters, which has been investigated by Binkley *et al.* [5].

In a previous work [15], we investigated the concept of dependence clusters on procedure-level program dependences computed using the *Static Execute After (SEA)* approach. The SEA relation between two procedures is a conservative type of dependence that takes into account the possible control-flow paths and call-structures in procedures [10]. This approach is more efficient at the expense of being a bit less accurate, and is defined as follows. For program elements (procedures, in our case) f and g, we say that $(f, g) \in$ SEA if and only if it is possible that any part of g is executed after any part of f in any one of the executions of the program. Similarly to the definition of slice-based dependence clusters, we regard two procedures to be in the same SEA-based cluster if their dependence sets coincide. This kind of cluster definition is usual as the maximal mutual dependence-based cluster definition is prohibitively expensive to compute. This definition has the additional good property that it gives a partitioning of the procedures into clusters.

Note, that there is no obvious relationship between SEA-based and slice-based dependence clusters. From our preliminary investigations we found that in many cases similar cluster structures will be formed, but we plan to investigate this more systematically in the future, and see if our findings can be applied to other notions of dependence clusters.

We computed SEA-based dependence clusters in the WebKit system – one of the subjects in the present work as well – and found that it exhibits a certain level of clusterization. In the same work [15], we then used the identified clusters to verify the connection to the performance of change impact analysis in a practical situation, and to enhance our test case prioritization method based on code coverage analysis.

Monotone Size Graphs (MSG) and the related "area under the MSG" metric [2] are often used to characterize dependence clusters in programs. An MSG of a program (see Figure 1 for examples) is a graphical representation of all dependence sets belonging to the procedures of the program by drawing the sizes of the sets in monotonically increasing order along the x axis from left to right. Then the area metric mentioned above is the total sum of all dependence set sizes. In the case of SEA-based dependences the total number of dependence sets equals

the number of procedures in a program, and this number is also the maximal dependence set size. In Figure 1, we can see MSGs of such dependences in which – despite its rectangular shape – the same number of procedures is represented on both axes. The most straightforward way of interpreting this graph is to observe dependence clusters as plateaus, whose width corresponds to the cluster size and the height is the dependence set size. Note, that it may happen that the same dependence set size incidentally corresponds to different sets which will not be noticeable in this graphical representation.

It has not yet been investigated thoroughly whether the MSG and its associated area metric are good enough as descriptors of the level of clusterization. We further elaborate on this concept with SEA-based clusters and verify the ways of characterizing dependence clusters using this and other kinds of metrics. A related investigation was performed by Islam *et al.*, who defined alternative descriptions of the clusterization in form of various graphical representations [16]. These approaches, however, resort to visual investigation only.

As noted above, it is believed that a single linchpin can be associated to a program with dependence clusters in many cases. In this work we investigate the effect of joint removal of linchpin candidates in cases where the removal of a single element does not produce the desired result.

An additional problem is linchpin identification itself. The naïve linchpin identification algorithm – a brute-force method trying all possible solutions one by one – is not scalable. Hence, previous research that employed fine grained analysis could deal with programs of up to 20 kLOC only [4]. In a similar fashion, our SEA-based analysis makes it possible to investigate programs with sizes of a magnitude larger thanks to the higher level granularity and a simpler, albeit less precise, analysis method. However, this method is still not usable for bigger programs as is the case with our two large systems.

We articulate the following **Research Questions**:

- RQ1 How typical are SEA-based dependence clusters in a variety of programs of different sizes and how can we categorize the programs more objectively in terms of their clusterization?
- RQ2 How typical are cluster structures that are held together by at most a few clearly identifiable procedures (linchpins), *i. e.* whose removal could reduce program clusterization significantly?
- RQ3 Currently the exact linchpins can be determined by brute-force only, which is infeasible for bigger programs; how closely can low-cost heuristic methods approximate linchpins?

III. Experiments setup

We collected a set of programs that served as the subjects following these principles: the set had to be comparable to other researchers' and our own previous results, our existing tools had to be able to handle them with ease, the programs had to be from different domains, and their sizes had to vary in a wide range. Based on these requirements, we fixed the language of the programs to C/C++, and in the first instance started with the collection of programs Harman *et al.* used in their experiments [6]. We could reuse 60% of these programs but also extended this set to finally arrive at 29 programs written in C (we will refer to this set as the *moderate size* programs). The basic properties of these programs can be seen in the first three columns of Table I. We provide names, lines of code (LOC) and the number of procedures (NP). The purpose of the other columns will be explained later.

TABLE I
Moderate size subject programs with clusterization information, sorted by Visual class and NP

Program name	LOC	NP: # of procedures	Visual class	Clusterization metrics AREA	ENTR	REGU	REGX
lambda	1766	104	▼ low				
epwic	9597	153	▼ low				
tile-forth	4510	287	▼ low				
a2ps	64590	1040	▼ low				
gnugo	197067	2990	▼ low				
time	2321	12	■ med				
nascar	1674	23	■ med				
wdiff	3936	29	■ med				
acct	7170	54	■ med				
termutils	4684	59	■ med				
flex	22200	153	■ med				
byacc	8728	178	■ med				
diffutils	17491	220	■ med				
li	7597	359	■ med				
espresso	22050	366	■ med				
findutils	51267	609	■ med				
compress	1937	24	▲ high				
sudoku	1983	38	▲ high				
barcode	5164	70	▲ high				
indent	36839	116	▲ high				
ed	3052	120	▲ high				
bc	14370	215	▲ high				
copia	1168	242	▲ high				
userv	8009	255	▲ high				
ftpd	31551	264	▲ high				
gnuchess	18120	270	▲ high				
go	29246	372	▲ high				
ctags	18663	535	▲ high				
gnubg	148944	1592	▲ high				

The second part of our data set consisted of two large industrial software systems from the open source domain. The first one was the WebKit system, which we have already used in some of our previous investigations [15], [17]. WebKit is a popular open source web browser engine integrated into several leading browsers by Apple, KDE, Google, Nokia, and others [9]. It consists of about 2.2 million lines of code, written mostly in C++, JavaScript and Python. In this research we concentrated on C++ components only, which attributes to about 86% (1.9 million lines) of the code. In our

measurements we used the Qt port of WebKit called QtWebKit on x86_64 Linux platform. We performed the analysis on revision $r91555$, which contained 91,193 C++ functions and methods as the basic entities for our analysis.

The other large system we used was the GNU Compiler Collection (GCC), the well-known open source compiler system [8]. It includes front ends for C, C++, Objective-C, Fortran, Java, Ada, and Go, as well as libraries for these languages. The GCC system is large and complex, and its different components are written in various languages. It consists of approximately 200,000 source files, of which 28,768 files are in C, which was the target of our analysis. In terms of lines of code, this attributes to about 13% of the code, 3.8 million lines in total (note that in C, the size of individual functions is usually larger than that of an average C++ method, hence this difference in lines of code compared to WebKit). We chose revision $r188449$ (configured for C and C++ languages only) for our experiments, in which there were 36,023 C functions as the basic entities for our analysis.

In our experiments we used our custom build tools as well as some existing components. To extract base program representations, as parser front ends we used Grammatech CodeSurfer [18] in the case of moderate size programs and Columbus [19] for the big programs. For the SEA dependence computation, our existing implementation of the SEA algorithm using ICCFG graphs [13] was applied. The benefit of using Columbus with ICCFG graphs for the bigger programs is that it is more scalable due to higher granularity of analysis.

We modified the SEA computation tool by adding the capability to ignore one or more procedures, which was required for linchpin determination. Since we needed to process and store a large number of dependence sets, we implemented additional tools (for MSG computation, cluster metrics computation, etc.) that employ efficient specialized data structures and algorithms. The calculation of the final results and the whole measurement procedure was performed using shell scripts.

We will describe our set of experiments and additional details regarding the measurements and tools in the corresponding sections.

IV. Existence of Dependence Clusters

A. Identification of clusters

To obtain the dependence clusters first we needed to compute the SEA-based dependence sets for all procedures in our subject programs. Similarly to the definition of slice-based dependence clusters, we regard two procedures to be in the same SEA-based cluster if their dependence sets coincide. This is a sensible definition because the SEA relation is reflexive, so if two procedures have the same dependence set, then they depend on each other as well. Note, that we did not apply the approximation of comparing dependence set sizes only as other studies suggested [2], [6].

In the following, we will investigate the *clusterization* of programs, which is the extent they exhibit dependence clustering. To express clusterization we followed two approaches:

Visual classification is carried out by (subjective) visual inspection of the MSGs, and assigns one of three levels to each program: *low*, *medium* and *high*.

Clusterization metrics are rigorously defined measures that are designed to express clusterization in easily quantifiable numerical form (values from $[0, 1]$).

For the moderate size programs, the fourth column (Visual class) of Table I shows the results of the visual classification we performed by inspecting the MSGs of the programs. As an illustration, Figure 1 shows MSGs of programs that are typical for each category. A cluster reveals itself as a wide plateau consisting of a number of equal-sized dependence sets. Typically, in a low class we cannot identify any plateaus, while for the high class there are one or two big ones, the rest being medium.[1] Visual classification reveals the following:

- The first program (`epwic`) does not show any plateaus, the landscape ascends in small increases.
- The second one (`findutils`) contains some moderately wide plateaus. They are not significant individually, but altogether cover much of the width of the landscape.
- For the last program (`gnubg`) we can see a single plateau occupying nearly the whole width of the landscape.

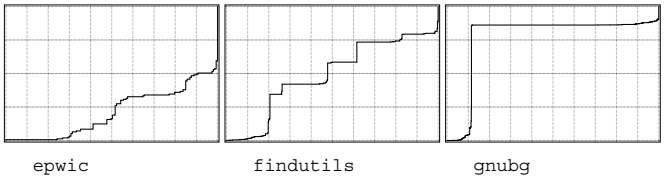

Fig. 1. Example MSGs for the visual classification (`epwic`: *low*, `findutils`: *medium*, `gnubg`: *high*)

As can be seen in Table I, overall we found 5 programs to be low, 11 medium, and 13 highly clusterized.

B. Measuring clusterization

Beyond visual interpretation, we will need an exact numerical expression (metric) of the level of clusterization and its relative change for two reasons. First, this way an automatic classification of programs could be made with the help of appropriate thresholds. Second, the metrics can be applied for the analysis of linchpins and measuring the effect of their removal (discussed in subsequent sections).

The obvious choice of metric to be used in these kinds of experiments is based on Binkley and Harman's work [2], who measured the area under MSG and used the change of this metric to analyze linchpins (we also rely on this metric and denote it by AREA in the following). The apparent weakness of AREA is that it increases if all dependence sets are increased by the same amount, although intuitively clusterization should not be different in such cases. Programs with no dependence clusters can have both small and large dependence sets, and vice versa.

[1]Note, that the same dependence set size may incidentally correspond to different sets, however this is very unlikely except for the smallest sets.

We experimented with alternative metrics to express clusterization in programs that are independent of the actual dependence set sizes and could reflect this property, these are presented below in a more formal way.

We define the measures so that they are comparable to each other and yield a value in the interval $[0, 1]$. For all metrics, 0 is set to mean that the level of clusterization is close to none, while 1 means that clusterization is maximal.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of procedures in a program X (for simplicity, we assume $n \geq 2$). The SEA relation of program X is a binary relation defined on its set of procedures, *i. e.* $SEA \subseteq P \times P$. With $\overline{n} = \{0, 1, \ldots, n\}$ we give the following auxiliary definitions:

$SEA(p)$ is the dependence set of a procedure $p \in P$ defined as

$$SEA(p) = \{q \in P \mid SEA(p, q)\}$$

The weight function w gives the sizes of the dependence sets as

$$w : P \to \overline{n}, \quad w(p) = |SEA(p)|$$

Formally, the clusterization of a program based on SEA dependences is in fact a partitioning of the procedure set P (not being transitive, the SEA relation itself does not exhibit partitions). We define two kinds of clusters, first by assigning two procedures to the same cluster (partition) if they have the same dependence set, *i. e.* they have the same SEA image:

$$\mathcal{I} = \big\{\, \{q \in P \mid SEA(q) = SEA(p)\} \,\big|\, p \in P \big\}$$

The second kind of cluster is defined by considering only the sizes (weights) of the dependence sets of the procedures:

$$\mathcal{S} = \big\{\, \{q \in P \mid w(q) = w(p)\} \,\big|\, p \in P \big\}$$

For any $c \in \mathcal{S}$ the weight of its members are equal, so we can assign the same weight to cluster c itself. Clearly $SEA(q) = SEA(p)$ implies $w(q) = w(p)$, so $\mathcal{I}$ is a refinement of $\mathcal{S}$. This also means that the weight function can be extended naturally to $\mathcal{I}$ as well.

Now we can define the different clusterization metrics as follows (the metrics always have to be interpreted in the context of a given program).

Consistently with earlier descriptions, AREA has three equivalent definitions:

$$\text{AREA} = \frac{1}{n^2} \sum_{p \in P} w(p) = \frac{1}{n^2} \sum_{c \in \mathcal{I}} |c| \cdot w(c) = \frac{1}{n^2} \sum_{c \in \mathcal{S}} |c| \cdot w(c)$$

Our next metric is based on an analogy of *entropy* and measures the "(dis)order" in the system of dependence sets in terms of their sizes. We consider a program more clusterized in this respect if there is a greater number of equal-sized dependence sets, *i. e.* when the entropy is lower (note, that this inverse relationship is required to obtain comparable metric intervals with the other metrics). Our entropy-based clusterization measure is formally defined as:

$$\text{ENTR} = 1 - \frac{\sum_{c \in \mathcal{S}} f(c) \cdot \log_2 f(c)}{\log_2 1/n}, \text{ where } f(c) = \frac{|c|}{n}$$

Finally, our two metrics referred to as *regularity* metrics are based on the number of partitions. The idea is that the fewer partitions there are, the larger their size must be, so there have to be more large clusters among them. Inversely, more partitions have to take more "regular" different sizes hence they will represent low clusterization. This metric has two variants, the first is based on $\mathcal{S}$, the other (extended, REGX) is based on $\mathcal{I}$.

$$\text{REGU} = \frac{n - |\mathcal{S}|}{n - 1} \qquad \text{REGX} = \frac{n - |\mathcal{I}|}{n - 1}$$

As noted earlier, all metrics are normalized (*i. e.* their value is a real number from the interval $[0, 1]$), which is useful to be able to compare the metrics of programs with different size to each other.

We computed all four metrics for all of our moderate size subject programs and compared the rankings of the procedures based on these individual metrics to the visual classification of the programs. These values are shown in the last four columns of Table I (to ease interpretation, the gray areas inside the small rectangles are set to be proportional to the metric values). Note that the ordering of the programs in this table was done based on the visual ranking first, then on the number of procedures inside each rank group.

Visual clusterization created three groups with 5 (low), 11 (medium), and 13 (high) elements, respectively. We would expect an ideal clusterization metric to yield values in such a way that the 5 smallest would be assigned to the "low" level, the middle 11 would be assigned to "medium", and the largest 13 to the "high" level group. Based on these criteria, the clusterization metrics can be characterized by counting how many programs they fail to assign to the group given by visual ranking. The counts are as follows: AREA $\to$ 10, ENTR $\to$ 7, REGU $\to$ 15, REGX $\to$ 8. The differences in these counts can also be observed by visual inspection of the metric values. It can clearly be seen that AREA and REGU are significantly worse than the other two metrics, while the difference is not so great regarding ENTR and REGX. ENTR is more precise on low and medium clusterization levels, while REGX performs better on highly clustered programs (see the last four columns of Table I).

Based on the above observations we will use ENTR and REGX to measure the degree of clusterization in the rest of the paper, and will mostly rely on ENTR where low or medium clusterization is concerned and use the other for high clusterization.

C. Dependence clusters in the big programs

So far, we have been dealing only with the moderate size programs, but our dataset contains two big programs as well, which need more thorough investigation. The MSGs for these two programs, GCC and WebKit, can be seen in Figure 2.

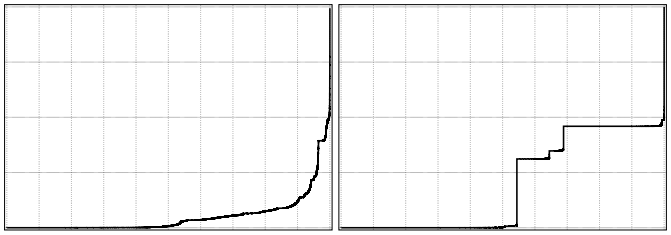

Fig. 2. MSGs for GCC and WebKit

The differences between the two programs are clear. GCC belongs to the low level clusterization category, while WebKit exhibits some clusterization (it would belong to the medium category in the visual ranking). The ENTR values are 0.4347 for GCC and 0.6980 for WebKit, while REGX is 0.3134 and 0.3552, respectively, which supports our initial (visual) classification for these two systems. While ENTR shows a notable difference, in the case of REGX it is not so significant, which may also reflect our finding from above that ENTR was better for low or medium clusterization.

It would be interesting if we could find any properties of these systems that justify their classification in terms of dependence clusters. In other words, what makes GCC not having significant dependence clusters as opposed to WebKit? In previous work [15], we analyzed the structure of source code and the dependences in WebKit in a slightly different context. After consulting with some key WebKit developers and showing them the members of the clusters, we came to the conclusion that clustarization is related to architectural concepts in the system.

We speculate that the most notable difference between the two systems in this respect is that while WebKit is essentially a library consisting of highly coupled elements for the distinct functional areas, GCC is a complex application but with much clear behavioural paths that are independent of each other. In WebKit, most complex functionalities are implemented in a set of highly interacting procedures (for example, webpage rendering is performed by several hundred procedures calling each other recursively). On the other hand, GCC implements functionalities like compiler optimization passes that are more isolated from each other. In addition, the two systems are written in different programming paradigms (C vs. C++) which may influence their internal structure. More detailed analysis of the causes for this difference remains for future work.

In the remaining parts of this paper we will be concerned with the identification of linchpins, however for programs of this size only the heuristic approaches are feasible. Hence we will subsequently use WebKit only to verify the effect of our heuristic methods, while GCC will play no role from now on.

V. LINCHPIN DETERMINATION

First, we identified the linchpins for our moderate size programs using the brute-force method that enumerates all procedures. As the biggest challenge in this topic is how to locate linchpins using more efficient methods, in the next experiment we investigated approximate heuristic methods for this task and compared their results to the exact results of the brute-force method. Since we could not apply the brute-force method to our large program WebKit, we verified the successfulness of the heuristic method on it in the final step.

A. Linchpin identification by brute-force

The simplest way to identify a possible linchpin in a program is to remove procedures one by one and see which one brings the biggest gain according to some metric. In the following, *gain* will mean the amount the respective metric is reduced in percentage: $\frac{m-m'}{m}$ [%], m being the original metric value and m' the value after linchpin removal.[2]

Specifically, we computed all SEA dependence sets for a program by ignoring the candidate procedure and all of its dependences. We compared the ENTR and REGX metrics of the reduced versions of the program to the corresponding metrics of the original program. This calculation was then repeated for all procedures in the program with these two metrics. For simplicity, we will present our results for REGX in the cases when the results were similar for both metrics, and will note explicitly in other situations. For the purposes of the remaining discussion, the procedure that caused the biggest reduction in the REGX metric was considered to be the linchpin.

TABLE II
REGX THRESHOLDS FOR LINCHPIN PROCEDURES

	Minimum gain	Maximum gain	Typical gain
Low clusterization	5%	20%	—
Medium clusterization	18%	55%	20%
High clusterization	13%	99%	37%

Table II summarizes results of linchpin determination for different clusterization classes of moderate sized programs. It shows how much reduction in the REGX clusterization value could be attained in the worst case (*minimum gain*) for a given clusterization class, and similarly how much was the *maximum gain*. It also indicates how much reduction could be attained if the few outlier programs with least gain—4 in the medium and 3 in the high class—are ignored (*typical gain*).

One would expect that programs with low clusterization do not contain linchpins, *i. e.* there is no procedure whose removal significantly reduces the already low clusterization. This was not entirely supported by our findings, as there were cases when as much as 20% gain could be achieved. This is not negligible, but as noted earlier, REGX performs better at high clusterization levels, so results for the low clusterization level are not entirely relevant. However, the achieveable gain is definitely larger for the medium and high classes, and gains for highly clusterized programs vary widely.

In Table III, we listed the results of linchpin calculation for moderate size programs with high clusterization. To get these

[2]Note, that we do not actually *refactor* linchpins and get equivalent programs but we merely remove the procedures in order to identify them.

results we had to compute over 15 million SEA sets altogether, but this was possible to complete in hours on an average server machine.

TABLE III
LINCHPINS FOR MODERATE SIZE PROGRAMS WITH HIGH CLUSTERIZATION

Program	REGX gain	Procedure name
barcode	57%	`Barcode_Encode`
bc	37%	`dc_func`
compress	37%	`compress`
copia	99%	`scegli`
ctags	13%	`createTagsForFile`
ed	53%	`exec_command`
ftpd	43%	`parser`
gnubg	52%	`HandleCommand`
gnuchess	53%	`main`
go	14%	`get_reasons_for_moves`
indent	47%	`indent_main_loop`
sudoku	41%	`rsolve`
userv	22%	`parser`

The second column shows the REGX gain after linchpin removal, which was quite significant (at least 37%) in almost all of the cases, 43.7% on average for this class of programs. The last column of the table shows the names of the respective procedures identified (which, except for `compress`, `gnubg` and `go`, were the same for ENTR as well). It is interesting to observe that, as names themselves suggest and a manual analysis of the programs confirms, most of the identified procedures indeed have central role in the programs. It is an open question, however, how many of these procedures could be deemed responsible for avoidable dependence clusters, in other words, dependence pollution [2]. Expectedly, procedures acting as the main procedures could not be easily refactored.

B. Heuristic determination of linchpins

We estimated that the brute-force method to determine the potential linchpin for the WebKit system would take about 70 years to complete using our strongest servers. So, obviously, we must find alternative methods to find (or at least approximate) the linchpins to enable practical application of dependence cluster related research.

The existence of dependence clusters and any related linchpins are determined by the structure of the dependences under investigation (SEA and the underlying ICCFG program representation in our case). Therefore, it is to be expected that by investigating the topology of the underlying dependence graph one could gain insight into what makes a program point a potential linchpin.

The problem does not have an obvious solution, so we wanted to investigate whether local properties of the dependence graph nodes (procedures) could be leveraged to approximate lichpins. We used the following heuristic metrics as potential indicators: NOI (Number of Outgoing Invocations from the procedure), NII (Number of Incoming Invocations to the procedure), sum of the former two (SOI=NOI+NII), and their product (POI=NOI·NII). We tried the sum and the product because we expected that in linchpin formation both incoming and outgoing dependences could be important.

To compare the actual linchpins identified by the brute-force method to the performance of the heuristic metrics, we related two values for each procedure in the programs: a clusterization metric (ENTR or REGX) after removing the procedure and one of the heuristic metrics (NOI, NII, SOI, POI) associated with the procedure. Then we used Pearson and Kendall correlation checks between the corresponding vectors of these values. We do not provide detailed data for these measurements because they all pointed out the same best heuristic estimation.

TABLE IV
PEARSON CORRELATION BETWEEN HEURISTIC METRICS AND THE ENTR AND REGX METRIC. UNDERLINED NUMBERS INDICATE STRONGEST CORRELATION IN THE CORRESPONDING BLOCK.

	ENTR				REGX			
Program	**NOI**	**NII**	**SOI**	**POI**	**NOI**	**NII**	**SOI**	**POI**
a2ps	-0.27	0.03	-0.16	-0.04	-0.57	-0.01	-0.39	-0.40
acct	-0.67	0.21	-0.52	-0.53	-0.67	0.13	-0.57	-0.46
barcode	-0.59	0.07	-0.55	-0.65	-0.71	0.06	-0.66	-0.74
bc	-0.72	0.04	-0.56	-0.57	-0.75	0.05	-0.58	-0.59
byacc	-0.11	-0.01	-0.08	-0.20	-0.72	0.05	-0.42	-0.40
compress	-0.72	0.04	-0.63	-0.49	-0.89	-0.09	-0.85	-0.63
copia	-0.72	-0.66	-0.98	-1.00	-0.70	-0.68	-0.98	-1.00
ctags	-0.42	0.03	-0.18	-0.23	-0.53	0.04	-0.23	-0.24
diffutils	-0.42	-0.02	-0.36	-0.51	-0.66	-0.02	-0.56	-0.57
ed	-0.67	0.03	-0.49	-0.56	-0.82	0.04	-0.59	-0.62
epwic	0.28	0.12	0.32	0.32	-0.50	-0.02	-0.48	-0.32
espresso	-0.55	0.03	-0.33	-0.46	-0.70	0.04	-0.42	-0.43
findutils	-0.25	0.07	-0.20	-0.04	-0.34	0.07	-0.29	-0.01
flex	-0.79	0.07	-0.70	-0.54	-0.88	0.08	-0.78	-0.62
ftpd	-0.74	0.03	-0.53	-0.40	-0.78	0.02	-0.57	-0.42
gnubg	-0.66	-0.07	-0.55	-0.68	-0.69	-0.07	-0.57	-0.71
gnuchess	-0.54	0.07	-0.47	-0.31	-0.55	0.06	-0.48	-0.29
gnugo	-0.45	0.04	-0.06	0.01	-0.53	-0.01	-0.13	-0.05
go	-0.49	0.03	-0.16	-0.31	-0.58	0.04	-0.18	-0.33
indent	-0.64	0.04	-0.45	-0.17	-0.69	0.05	-0.48	-0.16
lambda	0.30	0.53	0.50	0.58	-0.61	-0.49	-0.64	-0.57
li	-0.07	-0.17	-0.18	-0.18	-0.09	-0.15	-0.17	-0.18
nascar	-0.13	-0.18	-0.23	-0.41	-0.77	0.15	-0.76	-0.33
sudoku	-0.69	0.22	-0.26	-0.40	-0.79	0.20	-0.35	-0.52
termutils	-0.35	0.18	-0.21	-0.13	-0.46	0.17	-0.33	-0.20
tile	0.48	0.46	0.62	0.63	-0.27	-0.18	-0.29	-0.28
time	0.70	-0.29	0.47	0.70	-0.55	0.08	-0.47	-0.12
userv	-0.49	0.02	-0.35	-0.40	-0.57	0.04	-0.39	-0.39
wdiff	0.04	-0.23	-0.02	-0.50	-0.89	0.18	-0.89	-0.66
average	-0.36	0.03	-0.25	-0.26	-0.63	-0.01	-0.50	-0.42
strongest	17	0	1	11	23	0	2	4

In Table IV, we show Pearson correlation results for all programs. We marked the strongest correlation values for each program underlined; the last row shows the average correlation values for each metric. It can clearly be seen that the NOI metric (Number of Outgoing Invocations) is the best estimator for both ENTR and REGX. The best values are negative in the NOI columns, which means that for the procedures of a program there is a high correlation between a high NOI value and a low clusterization value resulting from the removal of that procedure. In other words, the higher NOI value a procedure has, the more likely it is that its removal would decrease the clusterization considerably, *i. e.* the more likely it is that the procedure is a linchpin.

In the case of ENTR and REGX metrics, in 59% and 79% of the cases NOI showed the strongest correlation; the average correlation was -0.36 and -0.63 (with standard deviations 0.4 and 0.18), respectively. The second best was POI showing strongest correlation in 38% and 14% of the programs with average correlation values -0.26 and -0.42. NII performed

poorly, which was surprising because we expected NOI and NII will perform similarly. The promising results for NOI are strengthened by the fact that *the highest NOI value predicts a linchpin correctly in most of the cases*: in the highly clustered group in 12 out of 13 programs, in the medium group in 7 out of 11 programs the procedure with the highest NOI value turned out to be a linchpin. What causes NOI to be the best estimator is not yet clear, we are going to investigate this question in the future.

As a statistical test to support our choice for NOI, we make a null-hypothesis that the other three metrics are at least equally good. For instance, consider NOI and NII with ENTR measure. Out of the 28 programs, in 22 instances NOI has stronger correlation than NII. Using Chernoff's bound we get that the probability of NII being at least as good as NOI is at most 0.01035. Chernoff's bound shows that NOI is in fact better with a probability of at least 0.9235 than any of the other three with respect to any of the considered measures.

Another interesting observation we made about the data is that for smaller programs the agreement between the NOI metric and both clusterization metrics was slightly better, suggesting that this heuristic will perform better for smaller programs. Figure 3 shows how the correlation values between NOI and ENTR as well as NOI and REGX change as a function of program size. Only programs with high clusterization are shown because in the other cases the relationship was not so evident. Particularly, from left to right, we can see the average correlation values for the programs ordered increasingly by their number of procedures. Although not drastically, but a tendency of worsening correlation can clearly be observed. This could also indicate the need for combined identification of linchpins as outlined at the end of this section. The exact causes of this phenomenon are not clear yet, they are probably related to the different topologies of small and bigger programs.

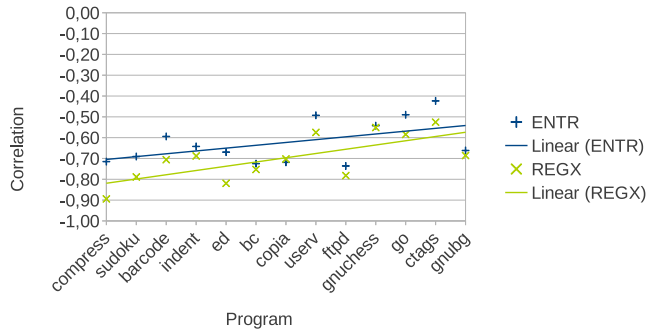

Fig. 3. Correlation change with program size of highly clusterized programs

Once we got these results about the best linchpin estimator heuristic metric, we applied it to WebKit to see whether we can achieve significant ENTR or REGX metric reduction and hence potentially find linchpins in that system too. In the first instance we calculated the NOI metrics for WebKit and applied the filtered dependence set calculation excluding the first 10 procedures with highest NOI values individually, thus obtaining a set of 10 clusterization reduction values. Unfortunately, after this experiment we could not observe any notable improvement in clusterization: even the largest ENTR and REGX reductions were negligible and visual inspection could not reveal anything either. Then we tried the other heuristic metrics as well in a similar way, but we got even worse results, so we decided to continue the research with the combined exclusion of procedures as detailed below.

C. Reducing clusterization by sets of linchpins

Linchpin identification and removal can be rephrased in graph theoretical terms as well. Given a graph G on an n element vertex set V, we say that $S \subset V$ is a separator set, if $G \setminus S$ contains only small connected components. If G does not contain certain substructures (large excluded minors, to be precise), then it must contain a small separator set. Still, in many cases the size of the separator set grows with n, usually it is about $O(\sqrt{n})$ [20].

While clusters in a SEA graph (or other graphs associated with a program) are more complex than connected components (for instance, most dependence graphs including SEA are not transitive), it is easy to see an analogy between the two kinds of problems. We think that analogously to separation of graphs, one cannot always expect to find a single linchpin vertex whose removal can significantly decrease clusterization. Rather, one should look for a hopefully small subset of vertices that somehow glue together the graph, and deleting them results in small clusters.

Graph theory also tells that not every graph has a small separator, for example, one has to delete a large number of vertices from a so called expander graph in order to make it disconnected. We expect that the graphs associated with programs are not expander graphs, and therefore the deletion of a relatively small subset can reduce clusterization. Finding such a linchpin set effectively seems to be challenging.

To verify this theoretical concept, we performed empirical measurements with sets of linchpins as opposed to only one on some representative programs from our moderate size subjects. In this series of measurements, three programs from the medium level clusterization group (`findutils`, `termutils`, `nascar`), and three other from the high clusterization group (`go`, `ed`, `sudoku`) were selected. Moreover, care was taken to include programs with different sizes in both groups. We took the first 10 procedures of a program with the highest NOI values, performed the calculation of the dependence sets while ignoring the first 0, 1, 2, ..., 10 procedures together and investigated the resulting clusterization metrics. We were interested to see whether there was indeed one linchpin that brought significantly more gain than the followers, or were the differences not so significant between these first 10 candidates.

Figure 4 shows the ENTR metric for the cumulative removal of the first k procedures (k-element linchpin sets), where the different k values are represented on the horizontal axis.

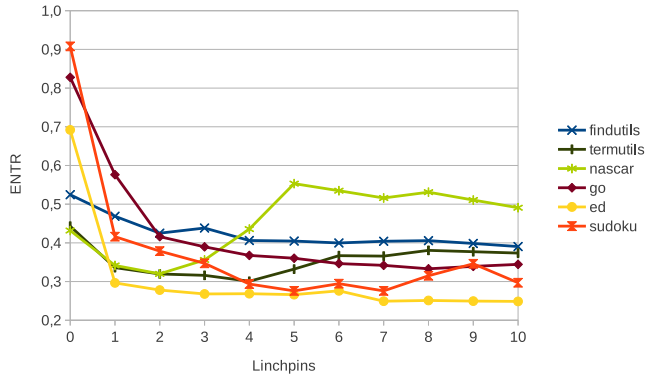

Fig. 4. Changing of the gains for the first 10 linchpins

One can observe that at most the sets with the first three procedures are notable and require further investigation. It can also be observed that for the three programs of medium level clusterization, the overall decrease in the metric is less than for the other group, as expected. More interestingly, it seems that regardless of the level of clusterization, program size plays an important role in the rate of decrease. Consider programs `sudoku` and the ten times larger `go`, for instance. We can see that for `sudoku` a significant decrease of clusterization occurs right after the first procedure, while for `go` even the second procedure contributes to the decrease significantly. As an example from the other group, the same effect can also be observed for `nascar` and `findutils`. The anomalous behavoiur of increasing clusterization can be expected in certain cases, the effect is more pronounced in the case of `nascar` due to its small size.

D. Linchpins in WebKit

Findings from above suggest that in many cases, especially for large programs, not only one program element (procedure) could be responsible alone for the formation of dependence clusters but a set of program elements together.

We performed similar experiments with the WebKit system in the hope to identify linchpin sets that significantly reduce clusterization. We expected that it would need more than only 2-3 procedures to achieve the same effect but we did not know how many. We tried removing several first procedures with the biggest NOI values but could not observe significant change in the clusterization up to until we removed about the first 200-250 procedures together. Removing 256 methods with the highest NOI values resulted in ENTR reduced by 7.2%, and REGX metric value reduced by 19.1% (AREA was reduced by 32.2%). In other words, the dependence sets collapsed this way, also changing dependence cluster formations, which can be observed in Figure 5.

Here, we can compare the original dependence clusters and the ones after removing these procedures (note, that in both cases the total number of procedures are represented on both axes, which is in the second case smaller by a comparatively

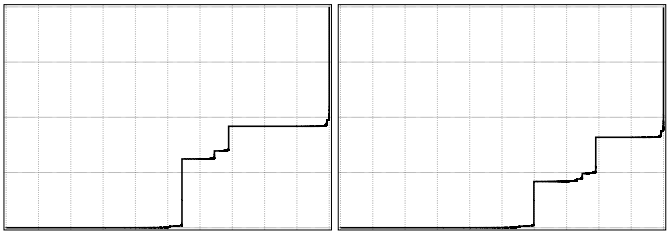

Fig. 5. WebKit MSGs before and after removing top 256 NOI procedures

neglible amount, hence the graphs are still comparable to each other). Although we cannot state that the clusters completely disappeared, the change is significant. To check if this result can indeed be attributed to the special role of the procedures with the top 256 NOI, we performed validation tests with three sets of randomly chosen 256 procedures. We found that randomly filtering procedures does not bring any improvement to the clusterization, change in ENTR was 0.04%, in REGX it was 0.25% on average.

It remains for future work to analyze in depth these linchpin sets and their effect on clusterization. From preliminary investigation, it seems that the clusters did not really disappear, they just changed their structure. Some of the procedures with high NOI could represent some very general connecting procedures, which do not really bear significant functionality (for example, we noticed a procedure that consisted of only a big switch statement and a huge number of method calls). When removing such procedures, the core dependences responsible for the main functionalities will remain, although the overall size will be smaller.

VI. Threats to Validity

We believe that the range of subject programs we used is representative for C/C++ as it ranges over various sizes and the domains are different. However, it would be important to see whether these findings can be generalized to other languages and types of dependences. The imprecision of our SEA-based dependences could slightly distort the results due to dependences that could have been avoided using a more precise method. However, as previous research showed [10], such false dependences are expectedly tolerable. We did not investigate if similar results would have been obtained using different, for instance, slice-based clusters.

We generalized our findings regarding the heuristic method based on the NOI metric to the WebKit system. We could not verify how good this heuristic actually performs on this system as we do not know the exact linchpin data, we could only state that some improvement has been obtained.

Our findings related to the presence of linchpin sets instead of individual linchpins showed that this is more probable with bigger programs. However, this highly depends on the system itself so this observation should be generalized with caution.

VII. Discussion and conclusions

This paper presented an empirical investigation of SEA-based dependence clusters. We may now answer the research questions set forth in this paper, however, our contributions to the better understanding of dependence clusters raised a number of additional questions.

Answering RQ1, we found that dependence clusters occur frequently in programs regardless of their domain and size, however there are also programs which exhibit very little clusterization. We gave precise definitions for four clusterization metrics and used them throughout to evaluate the results of our experiments. The results so far enable us to expect that the clusterization of programs can be measured with greater precision in the future. Additionally, we plan to experiment with more complex metrics that are less sensitive to the cluster and dependence set sizes.

RQ2 dealt with identifying one or more linchpins in programs. We were able to identify linchpin procedures using the brute-force method for all moderate size programs, but it is still to be explored in which cases we must seek for multiple linchpins and not only one. What we found is that as the program size increases it is less probable that only one program element is responsible for the formation of clusters. Note, that in this work we did not systematically investigate to decide if a dependence cluster reflects a dependence pollution [2] or not, and if the associated linchpins could be actually refactored, which is one of our future research directions.

Exhaustive exploration of possible linchpin combinations is computationally even more hopeless than for a single case. Hence additional, more sophisticated heuristic methods should be explored. Specifically, analyzing certain properties of the dependence graphs in terms of the graphs' topology is promising. Answering RQ3, we found that a specific metric based on graph structure, the Number of Outgoing Invocations for a procedure (NOI) is quite a good heuristic estimator for the linchpin, and the highest NOI value indeed predicts linchpins correctly in most of the cases, so in the short term we plan to investigate what made NOI the best in these experiments. Investigating other more sophisticated but still efficient methods is among our future plans as well, including the use of machine learning classifiers based on graph topology properties. Involving the hierarchical structure of the programs (packages, classes and methods) could also be promising.

We are still looking for the causes of the dependence clusters in WebKit. Although our heuristics gave interesting insights into the structure of these clusters, further investigation of the internals of this software is required. For this task we will consult WebKit developers, who already noticed that there are some elements in the identified clusters which they cannot explain. We need to investigate whether these are the consequence of the imprecision of the SEA algorithm or they represent some more hidden dependences in the system.

Acknowledgements

The authors would like to thank Zoltán Herczeg, László Langó, Csaba Osztrogonác, John Taylor and Béla Vancsics for their supporting work in this research. This research was supported by the Hungarian national grant GOP-1.1.1-11-2011-0049.

References

[1] D. Binkley, "Source code analysis: A road map," in *Proceedings of 2007 Future of Software Engineering (FOSE'07)*. IEEE Computer Society, 2007, pp. 104–119.

[2] D. Binkley and M. Harman, "Locating dependence clusters and dependence pollution," in *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 177–186.

[3] 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.

[4] D. Binkley and M. Harman, "Identifying 'linchpin vertices' that cause large dependence clusters," in *Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*, 2009, pp. 89–98.

[5] 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.

[6] 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.

[7] 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.

[8] "GCC, the GNU Compiler Collection," http://gcc.gnu.org/, last visited: 2013-05-08.

[9] "The WebKit open source project," http://www.webkit.org/, last visited: 2013-05-08.

[10] J. Jász, Á. Beszédes, T. Gyimóthy, and V. Rajlich, "Static Execute After/Before as a replacement of traditional software dependencies," in *Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM'08)*, 2008, pp. 137–146.

[11] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, 1984.

[12] 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.

[13] Á. Beszédes, T. Gergely, J. Jász, G. Tóth, T. Gyimóthy, and V. Rajlich, "Computation of Static Execute After relation with applications to software maintenance," in *Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07)*, 2007, pp. 295–304.

[14] 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, Jan. 2012.

[15] 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.

[16] S. Islam, J. Krinke, and D. Binkley, "Dependence cluster visualization," in *Proceedings of the 5th international symposium on Software visualization (SOFTVIS'10)*, 2010, pp. 93–102.

[17] Á. Beszédes, T. Gergely, L. Schrettner, J. Jász, L. Langó, and T. Gyimóthy, "Code coverage-based regression test selection and prioritization in WebKit," in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12)*, 2012, pp. 46–55.

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

[19] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy, "Columbus – reverse engineering tool and schema for C++," in *Proceedings of the International Conference on Software Maintenance*, 2002, pp. 172–181.

[20] K. Kawarabayashi and B. Reed, "A separator theorem in minor-closed classes," in *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, Oct., pp. 153–162.