Dependence Communities in Source Code

James Hamilton and Sebastian Danicic
Department of Computing
Goldsmiths, University of London
United Kingdom

***Abstract*—The concept of community structure arises from the analysis of social networks in sociology. Community structure can be found in many real world graphs other than social networks.**

We provide empirical evidence that dependence between statements in software also gives rise to community structure. This leads to the introduction of the concept of dependence communities in software. We give examples which suggest that the dependence communities reflect the semantic concerns of a program.

Furthermore, we show that there is a strong correlation between dependence communities and the previously studied dependence clusters.

I. Introduction

Software systems can be modelled as complex networks where vertices are typically components of a system and edges represent dependencies or interactions between components [55]. There a many different types of relationships in software including class or object dependence graphs, call graphs and package dependence graphs. Software graphs have been shown to have the non-trivial topological features of complex networks, such as a scale-free degree distribution, the small-world property and community structure [38, 42, 48, 52].

The problem of clustering software components has long been studied [30], for example the use of *hill climbing* [39] and *genetic* algorithms [16] applied to software modules. Previous research in the area has focused on the clustering of high-level components in a software system to recover a modular structure or re-modularise software. The Bunch tool [40, 41], for example, works on a Module Dependency Graph (MDG) which includes high-level system components such as Java classes or C files that are connected due to dependence. The purpose of the tool is to help maintain and understand existing software by clustering related modules together.

A strict form of a type of dependence community in software has previously been studied: a dependence cluster [3, 4, 11, 33, 35, 36] is a maximal set of mutually dependent program statements, or System Dependence Graph (SDG) vertices. It has been shown [3] that large dependence clusters may hinder software maintenance as a change in any of the members in a dependence cluster could affect any other member. Although some dependence clusters may naturally occur in a program, unwanted dependence clusters may be the result of bad programming practice which could be refactored away. Binkley and Harman [3] use the term 'dependence pollution' for such unwanted and avoidable dependence clusters.

A. Community Structure

The concept of community structure arises from the analysis of social networks in sociology [53]. Community structure can be found in many real world graphs other than social networks [47]. Random graphs, on the other hand, such as those produced by the Erdős–Rényi model [17], do not exhibit community structure.

Informally, a graph is said to have community structure if vertices grouped into groups that are densely connected internally, but sparsely connected to other groups [22]. A community can also be described as a sub-graph which is more tightly connected than average, i.e. cohesive [18]. Figure 1Community Structure is an example of a graph that exhibits community structure; the graph contains three natural communities.

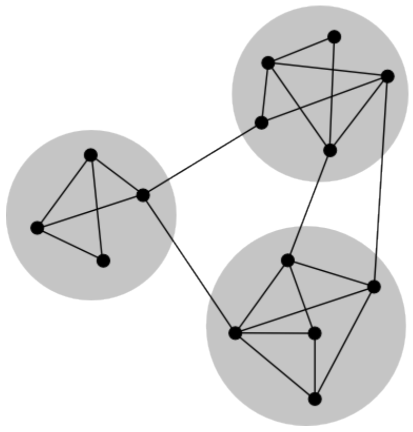


Fig. 1: A graph, with highlighted communities. $Q = 0.489$.

B. Community Structure in Software

Community structure has previously been found to exist in software networks. Šubelj and Bajec [51] found that Java class dependency networks show a clear community structure and that the detected communities do not exactly correspond to defined packages.

Valverde and Solé [49] analysed class dependency networks and found several highly frequent network motifs (sub-graphs that exhibit certain connectivity patterns) that appear to be a consequence of network heterogeneity and size, rather than as a result of the functionality of software.

Paymal et al. [46] applied a community detection algorithm to a class dependency network and examined changes in communities of interacting classes, over different versions of the software. The analysis showed that this technique provides important insights in understanding how code evolves over time.

This paper is the first to investigate community structure at the statement-level in software.

C. The Structure of this paper

In section IICommunity Structure in Graphs we describe modularity and the *Louvain method* used for community detection.

In section IIIDependence Communities in Backward Slice Graphs we introduce the notion of Backward Slice Graphs (BSGs) and introduce the concept of a *Dependence Community* by applying the *Louvain method* to BSGs.

In section IVEmpirical Study we report on an empirical study that applied the *Louvain method* to the BSGs of 44 open-source programs. The result of the empirical study was positive – each of the 44 BSGs exhibited community structure. The analysis involved 464,621 lines of code and BSGs that totalled 1,725,800 program slices.

In section VThe Semantic Nature of Dependence Communities we ask "what do dependence communities in software mean?" A dependence community is a set of statements in a program that have high dependence between them. It seems plausible that such collections of statements will be part of the same functional behaviour or semantics of the program. We manually inspect a number of programs to confirm this hypothesis.

In section VIDependence Communities vs Dependence Clusters we compare dependence communities with the previously studied *dependence clusters* and show that dependence communities give a stronger modularity i.e. the *Louvain method* partitions the BSG into areas of 'tighter' dependence. We show, however, that there is a strong correlation between the size of the largest dependence community and the size of the largest dependence cluster in a program.

In section VIIConclusion and Future Work, we conclude by highlighting the high potential for future applications of dependence communities in software.

II. Community Structure in Graphs

A. Modularity

Modularity is a measure of the quality of a particular division of a network [43–45], calculated as:

$$Q = \begin{array}{c}\text{(fraction of edges that}\\ \text{fall within}\\ \text{communities in the}\\ \text{given graph)}\end{array} - \begin{array}{c}\text{(expected number of}\\ \text{edges within those}\\ \text{communities in the null}\\ \text{model)}\end{array}$$

The value of the modularity lies in the range $[-1, 1]$. A modularity greater than 0 means that the graph does exhibit community structure. It is positive if the number of edges within communities exceeds the number expected on the basis of chance; the higher the modularity the stronger the communities [13]. An un-partitioned graph (a graph of 1 community) always has a modularity of 0 and a graph where each node is in its own community always has a negative modularity.

Modularity, of a weighted undirected graph, is defined as

$$Q = \frac{1}{2m}\sum_{i,j}\Big[A_{ij} - E_{ij}\Big]\delta(c_i, c_j)$$

where A_{ij} is the weight of the edge incident to i and j, $k_i = \sum_j A_{ij}$ is the sum of the weights of the edges incident to vertex i, c_i is the community to which vertex i is assigned, $\delta(u, v)$ is the Kronecker δ-function so the value is 1 if i and j are in the same community and 0 otherwise and $m = \frac{1}{2}\sum_{i,j} A_{ij}$. E_{ij} is the expected number of edges between i and j in a chosen *null model*.

The null model used is based on the configuration model [50] which is a randomised realisation of a particular graph that retains the original degree distribution. Given a graph where each vertex i has degree k_i, each edge is divided into two halves called *stubs*. Each stub is randomly connected to one of the other l_{n-1} stubs resulting in a random graph with the same degree distribution. The total number of stubs l_n is $\sum_{k=1}^{n} k_i = 2m$.

In the null model a vertex can be attached to any other vertex, by connecting two stubs together. The probability p_i of picking at random a stub attached to vertex i is $\frac{k_i}{2m}$ since there are k stubs attached to i out of a total of $2m$ stubs. The probability of a connection between i and j is $p_i p_j = \frac{k_i k_j}{4m^2}$ since edges are placed independently of each other. Therefore the expected number of edges E_{ij} between i and j is $2m p_i p_j = \frac{k_i k_j}{2m}$ [20].

Modularity, can therefore be written as

$$Q = \frac{1}{2m}\sum_{i,j}\Big[A_{ij} - \frac{k_i k_j}{2m}\Big]\delta(c_i, c_j)$$

The modularity for partition of the graph in fig. 1Community Structure is 0.489, indicating a good community structure.

B. Community Detection

The *Louvain method* [12] is a fast algorithm for detecting communities in large networks based upon modularity maximisation. The algorithm combines neighbouring nodes until a local maximum of modularity is reached and then creates a new network of communities; these two steps are repeated until there is no further increase in modularity. This creates a hierarchical decomposition of the network - at the lowest level all nodes are in their own community, and at the highest level nodes are in communities which gives the highest gain in modularity. This technique is simple, fast and has good accuracy. It has been tested on networks with millions of nodes/edges which is particularly important as software systems can be large.

Modularity optimisation, in general, has the so-called resolution limit problem where the modularity optimisation fails to identify communities smaller than a certain scale [21]. Using the *Louvain method* the problem is only partially relevant because the algorithm provides a decomposition of the network into communities for different levels of organization [12]; thus, in the resulting hierarchical community structure levels in between the highest and lowest may contain communities of interest.

The *Louvain method* is divided into two phases that are repeated iteratively. Starting with a network with N nodes we first put each node into it's own community, giving N communities of size 1. Then each neighbour j of vertex i is considered and the gain in modularity after moving i to the community of j is evaluated. Vertex i is placed into the community that gives the greatest increase in modularity but only if the increase is a positive value. If there is no positive increase obtained by place i into the community of any of it's neighbours then i remains in it's original community. This process is repeated and applied sequentially for all vertices in a randomised order until no further improvement in modularity is possible. The second phase of the algorithm involves building a new network whose vertices are the communities discovered in the first phase. The weights of the edges between two communities are the sum of the weights of the edges between the vertices in the corresponding communities. After the nodes have been combined into communities the first phase can be re-applied to the new network.

The two phases are iterated until there are no more changes and a maximum modularity is achieved.

III. Dependence Communities in Backward Slice Graphs

Program slicing [1, 5, 6, 8, 9, 31, 37, 54] is a technique which computes a set of program statements, known as a *slice*, that may affect a point of interest known as the *slicing criterion*.

Informally, a program P is sliced with respect to a *slicing criterion* which is a pair (V, i) where V is a set of program variables and i is a program point. The slice s of P is obtained by removing statements from P such that s is semantically equivalent to P with respect to the slicing criterion (V, i).

In this paper, we model the dependencies of a program as a complex network using the notion of a BSG; we build BSGs from SDGs using the mature and widely used CodeSurfer [26] software.

An SDG [27, 34] is an inter-connected collection of Procedure Dependence Graphs (PDGs) [19]. The vertices of a PDG represent the statements and predicates of a procedure; the edges of an PDG represent the intra-procedural control and data dependencies between statements and predicates. An SDG connects a collection of PDGs by inter-procedural control and data dependence edges.

Definition 1 (Backward Slice Graph (BSG))**.** A Backward Slice Graph (BSG) G consists of a set of SDG vertices V and a set of edges E of the form $e_{i,j}$ where $\forall e_{i,j} \in E, v_j \in BackwardSlice(v_i)$

The corresponding BSGs were built from the set of backward slices computed by CodeSurfer for each vertex in a subset of the SDG vertices. The subset of SDG vertices included those vertices which had a corresponding unnormalised Abstract Syntax Tree (AST) vertex and source-code characters. This allows us to tie the results back to the source-code of a program as well as greatly reducing the number slices required to be computed.

Listing 1: Sum Product Program with 3 communities highlighted in different colours

```
int main() {

  const int N = 10;
  int sum = 0;
  int product = 1;
  int i = 1;

  while(i < N) {
    sum = sum + i;
    product = product * i;
    i = i + 1;
  }

  printf("%d\n" , product );
  printf("%d\n" , sum );
}
```

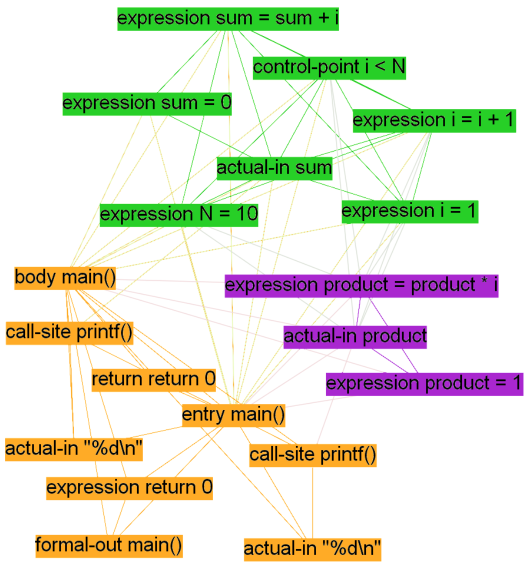


Fig. 2: BSG for **??** 1Sum Product Program with 3 communities highlighted in different colours, with 3 communities highlighted in different colours.

A. Example

We show the result of applying the *Louvain method* to the BSG of a simple program. In this small example, we found the results far from arbitrary: the algorithm appeared to partition the graph into communities each of which approximated to different ‘semantic’ concerns of the program.

This initial promising result was the evidence which led us to investigate these communities more extensively. We call a community in a BSG a *Dependence Community*.

The *sumproduct* program is shown in **??** 1Sum Product Program with 3 communities highlighted in different colours. The communities found in the program are depicted in fig. 2Example. There are 3 communities detected in this program: the ‘sum’ community, the ‘product’ community, and the ‘support’ community.

The ‘support’ community consists of the parts of the program which are not directly involved in calculating the sum or product, such as the procedure body, exit, return, and the calls to *printf*. This community is separate because there are fewer dependencies between itself and the sum/product code.

The ‘product’ community consists of the parts of the program which compute the product, including the initialisation of the product variable, the updating of the product variable and the `actual-in` to the *printf* call.

The ‘sum’ community consists of the parts of the program which compute the sum but also the loop code and the counter N. Communities cannot overlap because the algorithm partitions the graph therefore the loop code, which should intuitively be in both communities, must be placed in one or the other; the choice of the sum community is arbitrary.

The modularity of this partition of the graph is 0.227 - a positive value indicates that the graph has community structure.

The simple example clearly exhibits community structure; in the next section we investigate whether real-world programs also contain dependence communities.

IV. Empirical Study

The study in this section is based on the analysis of 44 open-source programs. The programs studied are a collection of open-source software that cover a range of application domains including games, small and large utilities and operating system components.

In our empirical study, we applied the *Louvain method* to BSG of each of the 44 programs and measured a number of quantities. As well as measuring the modularity we calculated the number of dependence communities, the size of the smallest, the size of the largest and the average size of communities measured as a percentage of program size.

A. The Program Sample

The 44 programs that are used in the empirical study are listed in table IResults. The smallest program has 71 Analysed Lines of Code (ALoC)[1], while the largest has 76,369. The total ALoC for the set of 44 programs is 464,621 and the average ALoC per program is 10,559.57.

[1]ALoC stands for the number of lines of code obtained by counting the unique number of lines of code attached to SDG vertices generated by CodeSurfer (each line of code will have one or more corresponding SDG vertices).

The smallest SDG, in terms of vertices, has 499 vertices and the largest has 2,954,718 vertices. In terms of edges, the smallest has 1,173 edges and the largest has 12,800,065 edges. The total number of vertices for the set of 44 program SDGs is 13,949,332 and the average number of vertices per SDGs is 39,222.73. The total number of edges for the set of 44 program SDGs is 61,878,708 and the average number of edges per SDGs is 1,406,334.25.

The program with the smallest number of slices had 172 slices and the program with the largest number of slices had 305,037 slices (number of slices is equal to number of vertices in the BSG). The smallest BSG, in terms of edges, has 9,296 edges and the largest has 16,449,266,842 edges.

The total number of slices computed for the set of 44 programs was 1,725,800 and the average number of slices per program is 317,030.28.

B. Results

The results of the empirical study are listed in table IResults and fig. 3Results shows the modularity for each of the 44 programs.

The most promising result of the empirical study is that all 44 programs have a positive modularity which indicates that BSGs exhibit community structure. The modularity for the set of programs varies, with some programs showing a stronger community structure; although it is not yet clear what ‘stronger community structure’ means in terms of software dependence.

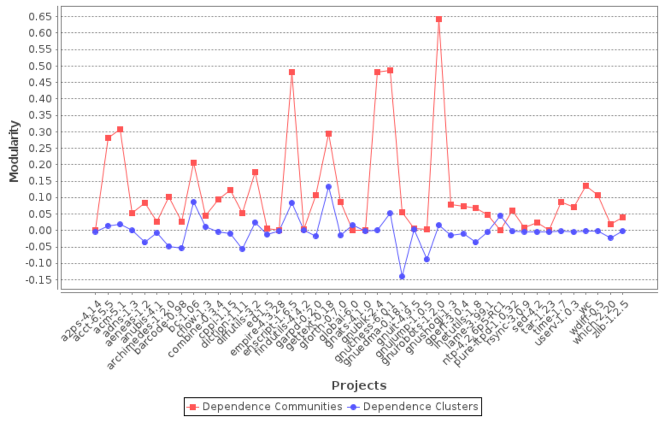


Fig. 3: Modularity calculated using dependence community and dependence cluster partitions of the BSGs.

The highest modularity is 0.644 for *gnurobots-1.2.0*, the lowest is 7.47×10^{-5} for *global-6.0* and the average modularity for the 44 programs is 0.12.

The program *ed-1.5*, although not the smallest, has only 2 communities – one consuming about 10% and the other 90% of the program. This suggests that there is a large section of code with a high dependence between statements. This result is similar to the discovery of a large dependence cluster in the same program by Binkley and Harman [3].

The largest community, on average, consumes 53.41% of a program; this echoes the results of Binkley and Harman who found that programs contain large dependence clusters [4]. Two programs (*global-6.0* and *gnujump-1.05*) have a dependence community that is $> 90\%$ of the program code. The larger program of the two has a community that is 97.8% of the program and 57 other communities; and the smaller has a community that is 92% of the program and only 11 other communities.

As program size increases (in terms of ALoC) so does the number of communities detected; the Pearson correlation is $R = 0.82$ with p-value < 0.000001. This could be due to larger programs performing more sub-tasks as part of the overall function of the program. The largest program, *gettext-0.18*, contains 264 dependence communities with a modularity of 0.294. The average number of dependence communities in this set of programs is 32.91.

V. The Semantic Nature of Dependence Communities

The empirical study found that dependence communities do exist in software but what do these mean? Recall that a community is a sub-graph with a higher than expected number of internal connections. In our situation this means sets of statements in a program that have high dependence between them. It seems plausible that such collections of statements will be part of the same functional behaviour of the program.

In this section, we investigate this by manually inspecting a number of programs and their dependence communities to see if, like the *sumproduct* example, the communities closely approximated the separate semantic concerns of the program.

Figure 5Watermark Communities shows the BSG for the GNU *wc* program and fig. 4Watermark Communities shows the partitions for the BSGs for the further 3 programs that we discuss in this section. In the outer ring, each section corresponds to a community with its size as a percentage of the total program size.

A. GNU wc

In the GNU *wc* program - that counts lines, characters and words in a text file - there are two communities detected; this can be seen graphically in fig. 5Watermark Communities. The two communities are, broadly speaking, the 'counting community' and the 'input/output community'.

The 'counting community' consists of the parts of the program which deal with counting the values of lines, characters and words in a file; this includes vertices that iterate through the characters in a file, vertices that increment counters, and vertices that deal with checking if a string is a word.

The 'input/output' community contains vertices which deal with the opening of the file, printing of error messages and printing of the results of the 'counting community'.

It is clear from the diagram that there are two distinct communities - the advantage of using a force-directed layout algorithm is that the nodes that are more tightly connected are placed close together; this is exactly the same idea behind community detection. Vertices that have more edges between them than with the rest of the graph will be put into a community.

B. GNU bc

The program *bc* is an "arbitrary precision numeric processing language" [23] which is a utility included in the POSIX standard. The program parses input from the user, translates it into bytecode and executes the bytecode. In the program there are two main communities detected – the parser and the calculator. These two communities combined make up 96% of the program; the parser community is 51% of the program and the calculator community is 45%.

C. GNU Chess

GNU Chess [24] is another programming that exhibits a clearly defined communities which correspond to syntactic modules of the program. The program is composed of three loosely-coupled modules: the fronted, adapter and engine; the adapter sits in between the front-end and the engine. The three main communities detected in the BSG correspond to the three components of the software. The developers of the software intended the components to be loosely-coupled which has resulted in the distinct communities. If coupling was high there would be more edges in the BSG between components and it would therefore be less likely that they would be separate communities. This indicates that community detection can be applied to the problem of software metrics for the calculation of coupling between modules in software. If the program shows a community that expands across functional areas it may indicate that the program has high coupling; on the other hand, if the detected communities closely match the functional areas of a program we can be confident that the modules are loosely-coupled.

D. GNU Robots

GNU robots is a program with low coupling between procedures. The user interface is written in C which interacts with an external Scheme program. This causes the coupling between procedures to be very low because many procedures communicate with the external program rather than with each other. In turn, this causes a large number of communities as there are many areas of highly dependent code with few edges between them.

E. Watermark Communities

Software watermarking involves embedding a unique identifier within a piece of software, to discourage the copying of software [28, 29]. Watermarking does not prevent copyright infringement but instead discourages it by providing a means to identify the creator of a piece of software and/or the origin of copied software. A hidden watermark can be extracted, at a later date, to prove ownership.

Watermark code can be protected by *opaque predicates* [14] which attempt to force the original program code to depend on the watermark and the watermark code to depend on the original code. However, it is difficult to make the watermark fully integrated into the original program. Preliminary work suggests that community detection could be used to uncover

Program	ALoC	SDG		BSG		Dependence Communities				
		Vertices	Edges	Vertices	Edges	Number	Smallest	Largest	Average	Modularity
a2ps-4.14	17,518	271,685	1,566,984	57,844	1,258,072,909	102	0.00346%	72.9%	0.980%	0.00101
acct-6.5.5	2,691	15,596	47,834	6,324	5,499,790	11	0.221%	28.2%	9.09%	0.282
acm-5.1	922	3,462	11,155	2,029	295,025	26	0.0493%	28.5%	3.85%	0.306
adns-1.3	5,865	102,855	347,055	20,895	177,781,311	10	0.0239%	42.2%	10.0%	0.0535
aeneas-1.2	803	17,016	61,044	8,315	20,658,335	3	22.2%	43.8%	33.3%	0.0844
anubis-4.1	6,618	68,510	247,986	19,510	149,296,932	25	0.0256%	48.9%	4.00%	0.0257
archimedes-1.2.0	766	17,665	68,781	7,957	16,881,616	11	0.126%	43.2%	9.09%	0.103
barcode-0.98	2,111	14,457	48,006	5,452	11,732,564	5	0.0183%	73.8%	20.0%	0.0277
bc-1.06	5,249	41,025	310,319	11,037	62,116,117	11	0.0634%	50.8%	9.09%	0.206
cflow-1.3	6,226	57,209	213,966	17,135	101,597,540	23	0.0117%	83.9%	4.35%	0.0454
combine-0.3.4	4,527	37,185	127,748	15,585	47,417,876	16	0.0321%	58.6%	6.25%	0.0935
cppi-1.15	1,949	12,904	46,162	5,471	6,661,891	24	0.0914%	46.6%	4.17%	0.123
diction-1.11	865	7,791	23,723	3,283	3,334,634	8	0.0305%	51.3%	12.5%	0.0539
diffutils-3.2	9,042	48,210	163,360	21,621	76,902,352	53	0.0278%	33.9%	1.89%	0.178
ed-1.5	1,883	30,779	110,126	7,274	37,419,735	2	10.3%	89.7%	50.0%	0.00692
empire-4.3.28	40,704	2,277,653	9,242,821	156,776	16,112,669,388	14	0.00319%	57.6%	7.14%	0.000247
enscript-1.6.5	9,896	107,684	539,478	33,681	287,940,746	25	0.00891%	48.5%	4.00%	0.481
findutils-4.4.2	17,206	157,518	534,727	52,843	1,191,798,635	37	0.00568%	43.7%	2.70%	0.00207
garpd-0.2.0	308	2,239	6,113	1,034	269,612	5	1.93%	43.2%	20.0%	0.108
gettext-0.18	76,369	2,954,718	12,800,065	305,037	9,961,065,414	264	0.00131%	53.1%	0.379%	0.294
gforth-0.7.0	7,457	26,851	4,456,673	13,907	52,707,360	10	0.0503%	61.3%	10.0%	0.0872
global-6.0	22,964	774,361	3,916,873	86,136	2,500,096,080	58	0.00348%	97.8%	1.72%	7.47e-05
gnats-4.1.0	13,425	130,337	551,887	35,295	666,919,670	24	0.0113%	64.2%	4.17%	0.000332
gnubik-2.4	2,936	13,542	31,977	9,417	805,652	39	0.0531%	16.2%	2.56%	0.483
gnuchess-6.0.1	12,064	103,525	336,793	28,072	85,105,593	79	0.0178%	27.8%	1.27%	0.486
gnuedma-0.18.1	12,485	1,334,533	3,921,091	63,560	1,141,664,053	94	0.00157%	56.1%	1.06%	0.0543
gnuit-4.9.5	11,522	181,333	785,979	29,792	356,982,854	32	0.0101%	68.0%	3.13%	0.00661
gnujump-1.0.5	4,979	42,620	136,880	15,345	77,794,411	12	0.0391%	92.0%	8.33%	0.00277
gnurobots-1.2.0	1,230	5,551	13,306	3,773	169,759	25	0.106%	18.9%	4.00%	0.644
gnushogi-1.3	4,417	24,079	79,091	9,597	34,527,979	18	0.0104%	59.1%	5.56%	0.0796
gperf-3.0.4	3,625	15,712	45,420	8,976	13,394,816	21	0.0334%	48.8%	4.76%	0.0730
inetutils-1.8	31,941	835,082	2,928,870	142,631	4,265,435,922	80	0.00210%	50.0%	1.25%	0.0694
lame-3.99.1	16,337	107,876	410,316	39,736	580,705,172	54	0.0126%	53.3%	1.85%	0.0482
ntp-4.2.6p5-RC1	41,232	1,177,368	5,501,985	167,394	8,053,207,729	72	0.000597%	49.7%	1.39%	8.38e-05
pure-ftpd-1.0.32	7,142	65,674	240,992	18,591	130,914,642	12	0.0377%	58.1%	8.33%	0.0595
rsync-3.0.9	21,099	2,145,312	9,327,181	142,962	16,449,155,234	30	0.00210%	67.7%	3.33%	0.00965
sed-4.2	6,527	75,658	276,319	24,377	374,571,561	11	0.0164%	53.5%	9.09%	0.0248
tar-1.23	22,002	535,824	1,974,361	97,435	4,946,266,842	53	0.00308%	52.9%	1.89%	0.00109
time-1.7	382	1,822	5,290	826	127,281	5	9.56%	37.7%	20.0%	0.0853
userv-1.0.3	3,629	69,978	231,217	14,184	97,915,761	7	0.00705%	79.4%	14.3%	0.0715
wc	71	499	1,173	172	9,296	2	48.8%	51.2%	50.0%	0.136
wdiff-0.5	658	3,242	9,326	1,301	385,953	4	15.1%	32.1%	25.0%	0.107
which-2.20	774	5,138	15,787	2,189	2,513,834	5	0.411%	67.6%	20.0%	0.0181
zlib-1.2.5	4,205	27,254	162,468	11,029	65,570,162	26	0.0272%	44.3%	3.85%	0.0407
Sum	464,621	13,949,332	61,878,708	1,725,800	69,426,360,038	1,448				
Average	10,559.57	39,222.73	1,406,334.27	317,030.28	1,577,871,819.05	32.91	2.49%	53.41%	9.54%	0.12

TABLE I: Dependence Community Statistics

hidden watermarks as the additional watermarking code tends to reside in its own communities.

For example, fig. 6Watermark Communitiesa shows the BSG of a Java program that has been injected with a dynamic watermark [15]; fig. 6Watermark Communitiesb shows the communities detected using the *Louvain method* in which the majority of the watermark code is in its own community.

VI. Dependence Communities vs Dependence Clusters

A dependence cluster [3, 33] is a maximal set of mutually dependent program statements, or SDG vertices. It is a maximal clique within a BSG in which every vertex is connected to every other vertex; thus a dependence cluster is a stricter form of dependence community in a BSG.

?? 2Example Dependence Cluster is a dependence cluster because slicing on any line will give the slice $\{1, 2, 3\}$.

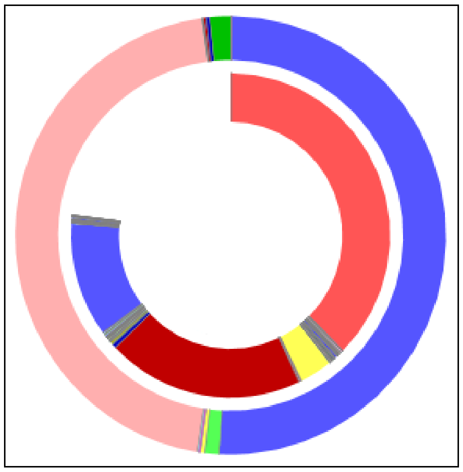
Listing 2: Example Dependence Cluster

```
1 while(i < 10)
2   if(A[i] > 0)
3     i = i + 1;
```

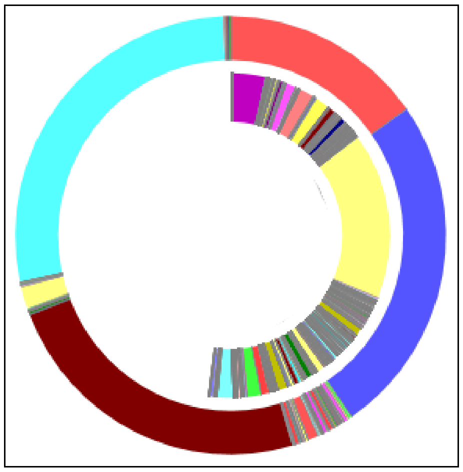
Finding dependence clusters is NP-hard which led Binkley and Harman to approximate to them by saying two vertices are in the same dependence cluster if and only if they have the same slice. These are cliques, but not, in general, maximal ones. A clique, being a fully connected subgraph, may be an overly strict requirement for what is required [33].

Figure 3Results shows the modularity for the 44 BSGs when partition using dependence communities and dependence clusters. It turns out that if we apply the *Louvain method* to the same graphs we get a partition with higher modularity. In other words it produces 'clusters' with a stronger 'internal interdependence' than those produced by Binkley and Harman's approximation.

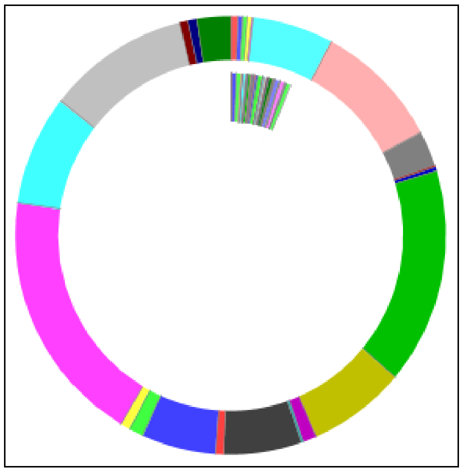
It could be argued, therefore, that dependence communities may be a better approximation to dependence clusters or at



(a) GNU bc



(b) GNU Chess



(c) GNU Robots

Fig. 4: BSG partitions, as a percentage of program size.

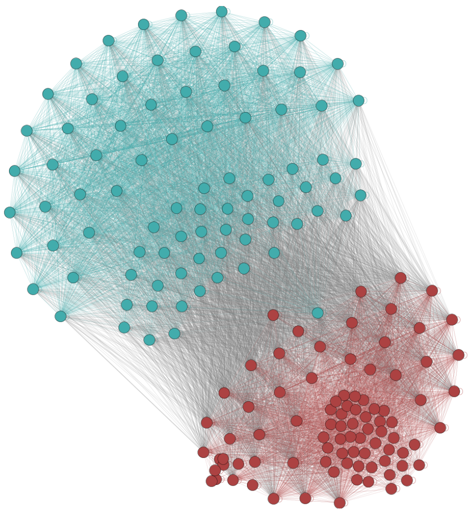
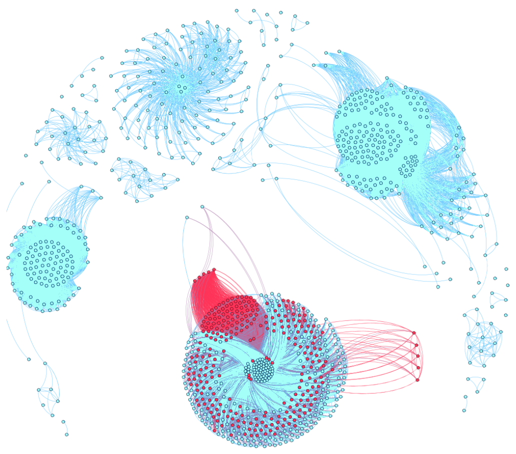
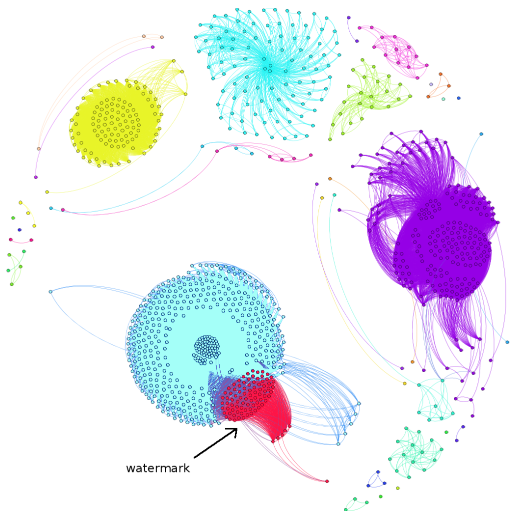


Fig. 5: Communities detected in the wc program. $Q = 0.136$.



(a) Highlighted Watermark



(b) Highlighted Communities

Fig. 6: A Java Program's BSG with highlighted watermark and with detected communities.

least a better approximation to the properties of programs that the authors are trying to capture using dependence clusters.

The inner rings in fig. 4Watermark Communities show the dependence cluster partitions for the BSGs for the 3 programs discussed in the previous section. Each section corresponds to a cluster with its size as a percentage of the total program size; only clusters with size > 1 are shown. It is interesting to note that the dependence clusters and dependence communities for *GNU bc* are quite similar, except that the communities are bigger. However, with *GNU Chess* there is only one 'large' dependence cluster and many small clusters; this is clearly a different result from dependence communities. In the *GNU Robots* program there are no large dependence clusters, most likely due to the low coupling between procedures; there are however, moderately sized communities. The difference results from the fact that clusters must be cliques whereas communities are not as strict – this means that there are more chances for the semantically related statements in a program to be grouped together.

There is a strong correlation between the size of the largest dependence community in a program and the size of the largest dependence cluster; the Pearson correlation between these is $R = 0.51$ with a p-value < 0.0001.

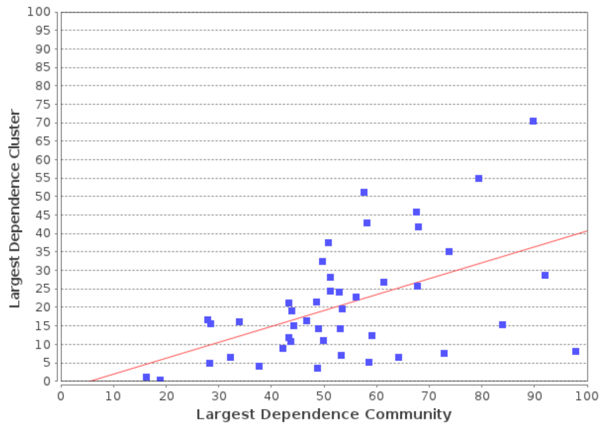


Fig. 7: Largest dependence community vs the largest dependence cluster.

It turns out that dependence communities tend to be larger than dependence clusters. A dependence cluster is a form of dependence community and a programs with large dependence clusters will have large dependence communities.

VII. Conclusion and Future Work

This paper is the first to investigate community structure at the statement-level in software. We introduce the concept of a *dependence community* and have shown that, at this level, BSGs have community structure and that the *Louvain method* seems to place the program statements in communities that reflect the semantic concerns.

We have described an empirical study of 44 open-source programs. The result of the empirical study was positive – each of the 44 BSGs exhibited community structure. The analysis involved 464,621 lines of code and BSGs that totalled 1,725,800 program slices.

We manually inspected a number of programs to answer the question "what do dependence communities in software mean?" The analysis revealed that dependence communities seem to reflect the functional behaviour or semantics of the program.

Dependence communities perform a goal similar to that of other work on the extraction of code from a program that expresses domain-level concepts implemented by the program. *Concept assignment* [2] attempts to relate the high-level concepts implemented by a program to portions of the source-code. Several approaches have been taken to solve this problem including the unification of program slicing and concept assignment [7, 10, 25, 32].

Clearly, the fact that software does exhibit community structure and that the fact that community structure appears to reflect the semantic properties of the software leads us to believe that there is a huge number of potential applications of dependence communities including program comprehension, maintenance, debugging, software metrics, refactoring, testing and software protection.

The modularity for the set of 44 programs in the empirical study varies, with some programs showing a stronger community structure than others. It is not yet clear what 'stronger community structure' means in terms of software dependence and further work will investigate the relationship between modularity and software dependence.

We found that programs have large dependence communities and similar results have been found for *dependence clusters*. We have shown that dependence communities give a stronger modularity i.e. the *Louvain method* partitions the BSG into areas of 'tighter' dependence. It is clear that there is a relationship between dependence clusters and dependence communities and further work will involve undertaking a more detailed examination of the differences and similarities between these.

Further work will also investigate the relationship between cohesion, coupling and dependence communities. For example: do procedures that contain more than 1 community have a low cohesion? Do programs with high coupling have larger communities?

Additionally, we intend to investigate the relationship between standard graph metrics such as density, centrality, clustering coefficient, network diameter and the dependence in software graphs. Modelling dependence as a graph affords us the opportunity to apply such metrics which may have significance in the context of software dependence.

Preliminary results suggest that, unlike other software networks, BSGs are not scale-free and that this is caused by the presence of large *dependence clusters*. Further work will involve verifying this result and conducting an empirical study to investigate the effects that dependence clusters have on the topology of BSGs.

References

[1] Richard Barraclough, David Binkley, Sebastian Danicic, Mark Harman, Robert Hierons, Ákos Kiss, Michael Laurence, and Lahcen Ouarbya. A trajectory-based strict semantics for program slicing. *Theoretical Computer Science*, 411(11-13), March 2010.

[2] T.J. Biggerstaff, B.G. Mitbander, and D Webster. The concept assignment problem in program understanding. *Proceedings of the 15th International Conference on Software Engineering*, pages 482–498, 1993. doi: 10.1109/ICSE.1993.346017.

[3] David Binkley and Mark Harman. Locating Dependence Clusters and Dependence Pollution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 177–186. IEEE, September 2005. ISBN 0-7695-2368-4. doi: 10.1109/ICSM.2005.58.

[4] David Binkley and Mark Harman. Identifying 'Linchpin Vertices' That Cause Large Dependence Clusters. *Source Code Analysis and*, pages 89–98, 2009. doi: 10.1109/SCAM.2009.18.

[5] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. A formalisation of the relationship between forms of program slicing. *Sci. Comput. Program.*, 62(3):228–252, October 2006. ISSN 0167-6423. doi: 10.1016/j.scico.2006.04.007.

[6] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. Theoretical foundations of dynamic program slicing. *Theor. Comput. Sci.*, 360(1):23–41, August 2006. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.01.012.

[7] David Binkley, Nicolas Gold, Mark Harman, Zheng Li, and Kiarash Mahdavi. An empirical study of executable concept slice size. *13th Working Conference on Reverse Engineering*, pages 103–114, 2006. doi: 10.1109/WCRE.2006.11.

[8] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology*, 16(2):8–es, April 2007. ISSN 1049331X. doi: 10.1145/1217295.1217297.

[9] David Binkley, Mark Harman, and Jens Krinke. Empirical study of optimization techniques for massive slicing. *ACM Transactions on Programming Languages and Systems*, 30(1):3–es, November 2007. ISSN 01640925. doi: 10.1145/1290520.1290523.

[10] David Binkley, Nicolas Gold, Mark Harman, Zheng Li, and Kiarash Mahdavi. An empirical study of the relationship between the concepts expressed in source code and dependence. *Journal of Systems and Software*, 81(12):2287–2298, December 2008. ISSN 01641212. doi: 10.1016/j.jss.2008.04.007.

[11] David Binkley, Mark Harman, Youssef Hassoun, Syed Islam, and Zheng Li. Assessing the impact of global variables on program dependence and dependence clusters. *Journal of Systems and Software*, 83(1):96 – 107, 2010. ISSN 0164-1212. doi: 10.1016/j.jss.2009.03.038.

[12] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, October 2008. ISSN 1742-5468. doi: 10.1088/1742-5468/2008/10/P10008.

[13] Aaron Clauset and MEJ Newman. Finding community structure in very large networks. *Physical review E*, 70 (6):1–6, December 2004.

[14] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, January 1998. ISBN 0897919793.

[15] Christian Collberg, Clark Thomborson, and Gregg M. Townsend. Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.*, 29(6):35, 2007.

[16] D. Doval, S. Mancoridis, and B.S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Software Technology and Engineering Practice, 1999. STEP '99.*, pages 73–81. IEEE Comput. Soc, 1999. ISBN 0-7695-0328-4. doi: 10.1109/STEP.1999.798481.

[17] P Erdős and A. Rényi. On the strength of connectedness of a random graph. *Acta Mathematica Hungarica*, 12 (1-2):261–267, 1961. doi: 10.1007/BF02066689.

[18] T. Evans and R. Lambiotte. Line graphs, link partitions, and overlapping communities. *Physical Review E*, 80(1): 9, July 2009. ISSN 1539-3755. doi: 10.1103/PhysRevE.80.016105.

[19] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987. ISSN 01640925. doi: 10.1145/24039.24041.

[20] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, February 2010. ISSN 03701573. doi: 10.1016/j.physrep.2009.11.002.

[21] Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences of the United States of America*, 104(1):36–41, January 2007. ISSN 0027-8424. doi: 10.1073/pnas.0605965104.

[22] M Girvan and M E J Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America*, 99(12):7821–6, June 2002. ISSN 0027-8424. doi: 10.1073/pnas.122653799.

[23] GNU. bc, 2011. URL http://www.gnu.org/software/bc/.

[24] GNU. GNU Chess 6.0.0, 2011. URL http://www.gnu.org/software/chess/manual/gnuchess.html.

[25] N. E. Gold, M. Harman, D. Binkley, and R. M. Hierons. Unifying program slicing and concept assignment for higher-level executable source code extraction. *Software: Practice and Experience*, 35(10):977–1006, Au-

gust 2005. ISSN 0038-0644. doi: 10.1002/spe.664.

[26] GrammaTech Inc. CodeSurfer, 2011. URL www.grammatech.com.

[27] GrammaTech Inc. Dependence Graphs and Program Slicing. 2011. URL http://www.grammatech.com/research/papers/slicing/slicingWhitepaper.html.

[28] James Hamilton and Sebastian Danicic. An Evaluation of Static Java Bytecode Watermarking. In *Proceedings of the International Conference on Computer Science and Applications (ICCSA'10), The World Congress on Engineering and Computer Science (WCECS'10)*, volume 1, pages 1 – 8, San Francisco, USA, October 2010. ISBN 978-988-17012-0-6.

[29] James Hamilton and Sebastian Danicic. A Survey of Static Software Watermarking. In *Proceedings of the World Congress on Internet Security 2011*, pages 114–121, London, 2011. IEEE.

[30] Mark Harman. The Current State and Future of Search Based Software Engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.29.

[31] Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance*, 10(6):1–30, November 1998.

[32] Mark Harman, Nicolas Gold, R. Hierons, and David Binkley. Code extraction algorithms which unify slicing and concept assignment. In *Working Conference on Reverse Engineering*, pages 11 – 21, Los Alamitos, CA, USA, October 2002. IEEE Computer Society Press.

[33] Mark Harman, David Binkley, Keith Gallagher, Nicolas Gold, and Jens Krinke. Dependence clusters in source code. *ACM Transactions on Programming Languages and Systems*, 32(1):1–33, October 2009. ISSN 01640925. doi: 10.1145/1596527.1596528.

[34] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990. ISSN 01640925. doi: 10.1145/77606.77608.

[35] Syed S. Islam, Jens Krinke, and David Binkley. Dependence cluster visualization. In *Proceedings of the 5th international symposium on Software visualization - SOFTVIS '10*, page 93, New York, New York, USA, October 2010. ACM Press. doi: 10.1145/1879211.1879227.

[36] Syed S. Islam, Jens Krinke, David Binkley, and Mark Harman. Coherent dependence clusters. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '10*, PASTE '10, page 53, New York, New York, USA, 2010. ACM Press. ISBN 9781450300827. doi: 10.1145/1806672.1806683.

[37] J. Krinke. Advanced slicing of sequential and concurrent programs. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 464–468. IEEE. doi: 10.1109/ICSM.2004.1357836.

[38] Nathan LaBelle and Eugene Wallingford. Inter-Package Dependency Networks in Open-Source Software. *CoRR*, cs.SE/0411096, November 2004.

[39] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organizations of source code. *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98*, pages 45–52, 1998. doi: 10.1109/WPC.1998.693283.

[40] S Mancoridis, B S Mitchell, Y Chen, and E R Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 50–59, 1999. doi: 10.1109/ICSM.1999.792498.

[41] BS Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):1–16, 2006.

[42] Christopher Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68(4), October 2003. ISSN 1063-651X. doi: 10.1103/PhysRevE.68.046116.

[43] M E J Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences of the United States of America*, 103(23):8577–82, June 2006. ISSN 0027-8424.

[44] M E J Newman and M Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69 (2):26113, 2004. ISSN 01678655.

[45] M.E.J. Newman. Analysis of weighted networks. *Physical Review E*, 70(5), 2004.

[46] Prashant Paymal, Rajvardhan Patil, Sanjukta Bhowmick, and Harvey Siy. Empirical Study of Software Evolution Using Community Detection. Technical report, University of Nebraska, Omaha, 2011.

[47] Mason A. Porter, Jukka-Pekka Onnela, and Peter J. Mucha. Communities in Networks. *Notices of the American Mathematical Society*, 56(9), February 2009.

[48] Alex Potanin, James Noble, and Marcus Frean. Scale-free geometry in Object Oriented programs. *Communications of the ACM*, 48(5):1–8, 2002.

[49] Sergi Valverde and Ricard Solé. Network motifs in computational graphs: A case study in software architecture. *Physical Review E*, 72(2), August 2005. ISSN 1539-3755. doi: 10.1103/PhysRevE.72.026107.

[50] R. van der Hofstad. Random graphs and complex networks. Technical report, Eindhoven University of Technology, 2008.

[51] Lovro Šubelj and Marko Bajec. Community structure of complex software systems: Analysis and applications. *Physica A: Statistical Mechanics and its Applications*, 390(16):2968–2975, August 2011. ISSN 03784371.

[52] Lei Wang, Zheng Wang, Chen Yang, Li Zhang, and Qiang Ye. Linux kernels as complex networks: A novel

method to study evolution. In *2009 IEEE International Conference on Software Maintenance*, pages 41–50. IEEE, September 2009.

[53] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications (Structural Analysis in the Social Sciences)*. Cambridge University Press, 1994. ISBN 0521387078.

[54] Mark D Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. Phd, University of Michigan, Ann Arbor, 1979.

[55] Lian Wen, D Kirk, and R G Dromey. Software Systems as Complex Networks. In *Cognitive Informatics 6th IEEE International Conference on*, pages 106–115. Ieee, 2007. doi: 10.1109/COGINF.2007.4341879.