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. Recently, efficient community detection algorithms have been developed which can cope with very large graphs with millions of nodes and potentially billions of edges. So, for the first time, there is the potential for investigating communities in real industrial-strength software at the statement level. We provide empirical evidence that dependence between statements in software does, indeed, give rise to community structure. Initial findings suggest that the separate *dependence communities* are far from arbitrary. They appear to decompose systems into areas of distinct functionality. This new approach to system decomposition has tremendous potential in many areas of software engineering, particularly in reverse engineering of legacy software and program comprehension.**

I. Introduction

Many naturally occurring phenomena can be represented in terms of graphs with edges and vertices. An understanding of the *community structure* of such graphs can give us a deeper understanding of the phenomena themselves. This idea has already been applied in many areas including sociology [1–3], biology [4] and, of course, computing [5]. This paper is the first to investigate the community structure in dependence graphs of program statements in software.

Informally, a set of vertices of a graph is a *community* if the number of internal edges is *more than expected* given the degree distribution of the vertices [6].

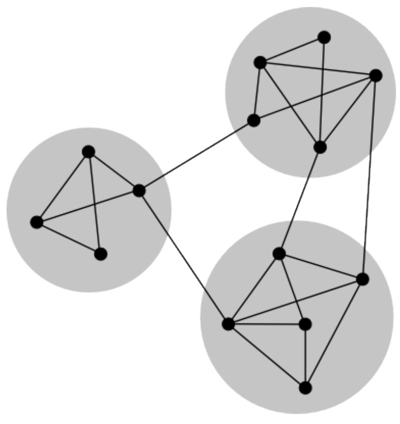


Figure 1: A graph, with highlighted communities.

The size and complexity of industrial strength software systems are constantly increasing. This means that the task of managing large software projects is becoming even more challenging. Software clustering approaches can have an important role in the task of understanding large, complex software systems by automatically decomposing them into smaller, easier-to-manage subsystems. In this paper, we present an initial investigation into whether partitioning software into graph-theoretic *communities* of statements can be usefully applied to this problem.

The graphs we consider, called Backward Slice Graphs (BSGs), are made up of vertices which represent the program statements and edges that represent dependencies between these statements. In industrial software, such graphs will typically contain millions of vertices and possibly billions of edges. New community detection algorithms such as the *Louvain method* [7] have successfully been applied to graphs of this size. There is now, therefore, the possibility that such techniques can be applied to real-world industrial software.

We believe that for a clustering technique to be useful in software engineering it must provide what we call *Semantic Separation*, i.e. the clusters must in, some sense, partition the system into its different *functionalities*. In this paper, we provide initial evidence that the communities produced by applying the Louvain method to BSGs do, indeed, give rise to semantic separation.

Community structure has previously been found to exist in software, but not nearly at such a fine-grained level. It was found [8], for example, 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 [9] analysed class dependency networks and found several highly frequent network motifs that appear to be a consequence of network heterogeneity and size, rather than as a result of the functionality of software. Paymal *et al.* [10] also 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 Bunch tool [11] looks for community structure in the Module Dependency Graph which includes high-level system components such as Java classes or C source 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.

II. Communities in Graphs

A *community* in a graph with vertex-set V is a subset V_i of V where the difference between the number of edges connecting elements of V_i and the *expected* number of edges connecting elements of V_i based on the degree distribution of V_i is positive. A positive value indicates that the *tightness of connectivity* of V_i is *better than expected*. Where this is the case, then we say that V_i is a *community*. Clearly, the higher the value, the *stronger* the community. A community where the sum of the degrees of the vertices was low would have a lower expected number of internal edges than one where the sum of the degree of the vertices was high. This implies that it is possible for a weakly connected set of vertices of low degree to be a *stronger community* than a more strongly connected set of vertices of high degree.

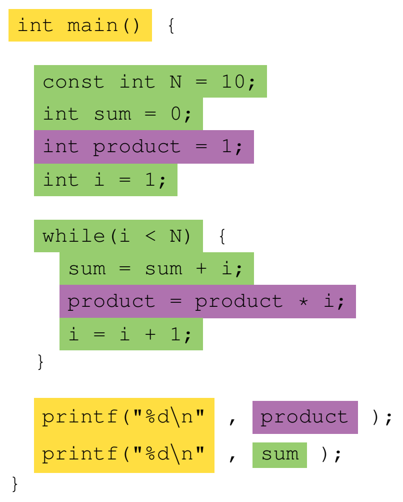
We can partition the vertices of a graph into subsets $\{V_i\}$ (the *elements* of the partition). In general, some elements in a partition may be communities and others may not. As well as computing the community structure of each element of the partition, we can measure the community structure of the whole partition based on the strength of community structure of each element of the partition. This quantity is known as *modularity* [7]. The modularity of a partition is a value between -1 and 1 that measures the density of links inside communities as compared to links between communities. Once again, if a graph has positive modularity it is said to possess *community structure*.

A. Community Detection

Given a graph, finding the partition with the highest modularity is NP-hard. The *Louvain method* [7] is, however, a fast algorithm that can find high modularity partitions in very large graphs. 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 is the algorithm we have used in the work presented in this paper.

III. Dependence Communities

Before we can look for communities of statements in software, we need a suitable graph in which to look for such communities. We want edges to represent dependencies between statements. Program slicing [12] is a technique which computes a set of program statements, known as a *backward slice*, that may affect a point of interest known as the *slicing criterion*. The backward slice of a given statement s contains all the other statements upon which s depends. A natural choice was, therefore, to connect vertex v_1 to v_2 if v_1 is in the backward slice of v_2. We call such a graph



```
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 );
}
```

Figure 2: The *Sum Product* Program with 3 communities highlighted in different colours

a BSG. [1]. Communities in the BSG will, thus, be sets of statements with *strong* interdependencies. We call such sets *dependence communites*.

As an initial experiment, we used the slicing tool *CodeSurfer* [14] to produce the BSG of the well-known *Sum Product* program in Figure 2. We then applied the Louvain method to find communities in this BSG[2]. We found the results far from arbitrary: the algorithm partitioned the graph into communities each of which approximated to different semantic concerns of the program (see Figure 2). There are three communities detected in this program: the *sum* community, the *product* community, and the *support* community. This initial promising result was the evidence which led us to investigate these communities more extensively in an initial empirical study.

IV. Initial Empirical Study

The programs studied are a collection of 44 open-source programs that cover a range of application domains including games, small and large utilities and operating system components. In the smallest program we analysed 71 lines of code, while in the largest we analysed 76,369. The total lines

[1]We also considered looking for communities in the System Dependence Graph (SDG) [13]. Here vertices are only connected if there is a *direct* data or control dependence. Transitive dependencies are not connected. Intuitively, transitive dependencies are as important as direct ones (their effect is just as important) and so we rejected this approach. To back up our intuition, we applied the Louvain method directly to the SDG in a number of cases and found there was a strong correlation between the communities and the separate procedures in the programs. The reason for this is high ratio of intra procedural edges to interprocedural edges in the SDG.

[2]Although the BSG is a directed graph, the standard Louvain method is applied to undirected graphs. Future work will compare this approach with variants of the Louvain method which take direction of edges into account.

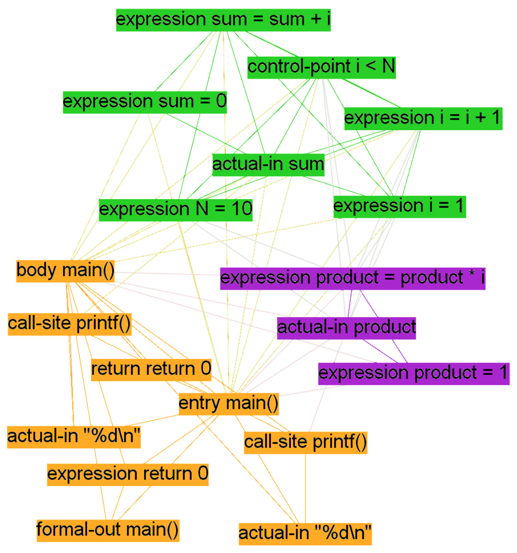


Figure 3: BSG for the program in Figure 2, with 3 communities highlighted in different colours.

of code analysed for the set of programs was 464,621 and the average lines of code analysed per program was 10,559.57. For each program, we first computed the BSG with the help of *CodeSurfer* and then applied *Louvain method* to each BSG. 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.

An important result of the study is that all 44 programs have a positive modularity (although some are almost zero) which is evidence that dependence in software may, indeed, exhibit community structure at the statement level.

A. *Semantic Separation*

As implied earlier, a dependence community is a set of statements in a program that have *higher than expected* dependence between them. It seems plausible that such collections of statements will be part of the same functional behaviour of the program. An important part of our empirical study was to investigate this hypothesis. To do this, we manually inspected several programs and their dependence communities to see if, like the *Sum Product* example, the communities closely approximated the separate semantic concerns of the program.

B. *GNU wc*

In the GNU *wc* program, that counts lines, characters and words in a text file, we found two dependence communities (see Figure 4). The two dependence 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 statements that iterate through the characters in a file, statements that increment counters, and statements that deal with checking if a string is a word. The *input/output community* contains statements which deal with the opening of the file, printing of error messages and printing of the results of the *counting community*.

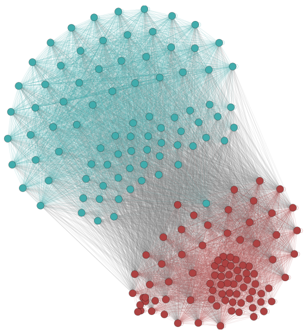


Figure 4: Dependence Communities detected in the *wc* program.

C. *GNU bc*

The program *bc* is an *arbitrary precision numeric processing language* [15] 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, two main dependence communities were detected: the *parser* and the *calculator*. These two dependence communities combined make up 96% of the program; the *parser* community is 51% of the program and the *calculator* community is 45%.

D. *GNU Chess*

GNU Chess [16] is another program with clearly defined dependence communities which correspond to syntactic modules of the program. The program is composed of three loosely-coupled modules: the *front-end*, *adapter* and *engine*; the *adapter* sits in between the *front-end* and the *engine*. The three main communities detected in the BSG correspond to these three components of the software.

The dependence communities in these three programs show clear semantic sepraration: in all cases the Louvain algortihm has partitoned the BSG into separate functionalities of the program.

V. Conclusions and Future Work

This work 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. This new approach has applications in all areas of software engineering where system decomposition is important, including reverse engineering of legacy software and program comprehension.

We believe that good semantic separation is the key to the usefulness of partitioning techniques in software engineering. (There is little point in breaking a piece of software into smaller components if these components do not in some way reflect different *functionalities*.) Of course no automated approach can perform perfect semantic separation. Our hypothesis is that dependence communities computed using new algorithms applied to program graphs can give *sufficiently good* semantic separation to be highly applicable in a number of areas of software engineering, including re-engineering and re-factoring, maintenance, comprehension and metrics.

The overall aim of future work will be to investigate the applicability of novel algorithms in community detection to these graphs and assess the impact on software engineering by measuring the resulting semantic separation and comparing it with other approaches. Importantly, we need to understand why some approaches to semantic separation are better than others in order that the techniques can be improved. This breaks down into the following tasks:

(1) Measuring semantic separation. This will be done by employing techniques similar to watermark injection [17]. *Intertwining* a number of pieces of code with separate functionality and measuring how well the pieces are separated again by the community detection algorithm.

(2) We will take advantage of the fact that we have already developed the infrastructure for dependence community detection of large programs and perform a large empirical study on a large number of real programs which measures the semantic separation in our approach and compares it with other clustering techniques.

(3) Investigating methods of improving the semantic separation in dependence communities. So far we have looked for communities in only one form of program dependence graph. Other similar program based graphs and community detection algorithms may be more appropriate.

(4) A theoretical analysis of semantic separation. Which semantic aspects of program are approximated to by dependence communities and why? This understanding may lead to better approaches to semantic separation of software in future.

References

[1] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications (Structural Analysis in the Social Sciences)*. Cambridge University Press, 1994.

[2] M. A. Porter, J.-P. Onnela, and P. J. Mucha, "Communities in Networks," *Notices of the American Mathematical Society*, vol. 56, no. 9, Feb. 2009.

[3] F. Liljeros, C. R. Edling, L. A. Amaral, H. E. Stanley, and Y. Aberg, "The web of human sexual contacts." *Nature*, vol. 411, no. 6840, pp. 907–8, Jun. 2001.

[4] R. Albert, "Scale-free networks in cell biology." *Journal of cell science*, vol. 118, no. Pt 21, pp. 4947–57, Nov. 2005.

[5] A. Potanin, J. Noble, and M. Frean, "Scale-free geometry in Object Oriented programs," *Communications of the ACM*, vol. 48, no. 5, pp. 1–8, 2002.

[6] 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*, vol. 99, no. 12, pp. 7821–6, Jun. 2002.

[7] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, Oct. 2008.

[8] L. Šubelj and M. Bajec, "Community structure of complex software systems: Analysis and applications," *Physica A: Statistical Mechanics and its Applications*, vol. 390, no. 16, pp. 2968–2975, Aug. 2011.

[9] S. Valverde and R. Solé, "Network motifs in computational graphs: A case study in software architecture," *Physical Review E*, vol. 72, no. 2, Aug. 2005.

[10] P. Paymal, R. Patil, S. Bhowmick, and H. Siy, "Empirical Study of Software Evolution Using Community Detection," University of Nebraska, Omaha, Tech. Rep., 2011.

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

[12] M. D. Weiser, "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method," PhD, University of Michigan, Ann Arbor, 1979.

[13] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, Jan. 1990.

[14] GrammaTech Inc., "CodeSurfer," 2011. [Online]. Available: www.grammatech.com

[15] GNU, "bc," 2011. [Online]. Available: http://www.gnu.org/software/bc/

[16] GNU, "Gnu chess 6.0.0," 2011. [Online]. Available: http://www.gnu.org/software/chess/manual/gnuchess.html

[17] C. Collberg, C. Thomborson, and G. M. Townsend, "Dynamic graph-based software fingerprinting," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 6, p. 35, 2007.