The Journal of Systems and Software 108 (2015) 193–210

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Exploring community structure of software Call Graph and its applications in class cohesion measurement

Yu Qu [a], Xiaohong Guan [a,*], Qinghua Zheng [a], Ting Liu [a], Lidan Wang [a], Yuqiao Hou [a], Zijiang Yang [b]

[a] Ministry of Education Key Lab for Intelligent Network and Network Security, Xi'an Jiaotong University, Xi'an, Shaanxi 710049, China
[b] Department of Computer Science, Western Michigan University, Kalamazoo, MI 48167, USA

ARTICLE INFO

Article history:
Received 20 November 2014
Revised 16 April 2015
Accepted 7 June 2015
Available online 16 June 2015

Keywords:
Class cohesion metrics
Complex network
Community structure

ABSTRACT

Many complex networked systems exhibit natural divisions of network nodes. Each division, or community, is a densely connected subgroup. Such community structure not only helps comprehension but also finds wide applications in complex systems. Software networks, e.g., *Class Dependency Networks*, are such networks with community structures, but their characteristics at the function or method call granularity have not been investigated, which are useful for evaluating and improving software intra-class structure. Moreover, existing proposed applications of software community structure have not been directly compared or combined with existing software engineering practices. Comparison with baseline practices is needed to convince practitioners to adopt the proposed approaches. In this paper, we show that networks formed by software methods and their calls exhibit relatively significant community structures. Based on our findings we propose two new class cohesion metrics to measure the cohesiveness of object-oriented programs. Our experiment on 10 large open-source Java programs validate the existence of community structures and the derived metrics give additional and useful measurement of class cohesion. As an application we show that the new metrics are able to predict software faults more effectively than existing metrics.

1. Introduction

Many natural and man-made complex networked systems, including metabolic networks, computer networks and social networks, exhibit divisions or clusters of network nodes (Flake et al., 2000; Fortunato, 2010; Girvan and Newman, 2002; Mucha et al., 2010; Palla et al., 2005). Each division, or **community** (Girvan and Newman, 2002), is a densely connected and highly correlated subgroup. Such community structure not only helps comprehension but also finds wide applications in complex systems. For example, researchers in *Biology* and *Bioinformatics* have applied community detection algorithms to identifying functional groups of proteins in *Protein–Protein Interaction* networks (Dunn et al., 2005; Jonsson et al., 2006). For online auction sites such as ebay.com, community structure is used to improve the effectiveness of the recommendation systems (Jin et al., 2007; Reichardt and Bornholdt, 2007). A survey on the applications of community detection algorithms can be found in Fortunato (2010).

There are also research efforts to investigate community structures in software, a very complex system (Concas et al., 2013; Pan et al., 2011; Šubelj and Bajec, 2011; 2012; Šubelj et al., 2014). Most of them reported a significant community structure of a certain type of software network such as *Class Dependency Networks* (Šubelj and Bajec, 2011). Some pioneering applications of software community structure are proposed (for more details, please refer to Section 2). However, there are still some unsolved problems.

Firstly, most of the measurements are performed on the network of classes. Little results are reported on the granularity of software method or function call, i.e., method/function *Call Graphs* (Graham et al., 1982). Such investigation is necessary from both theoretical and practical perspectives. In addition, measurements of the network of classes cannot be used in intra-class structure, which limits their applications in software quality evaluation and improvement.

Secondly, these pioneering applications have not been directly compared or combined with existing software engineering metrics and practices. Comparison with baseline practices is needed to convince people to adopt the proposed approaches. Only when the proposed approaches outperform or complement the existing method,

* Corresponding author. Tel.: +86 29 82663934.
E-mail addresses: yqu@sei.xjtu.edu.cn (Y. Qu), xhguan@sei.xjtu.edu.cn (X. Guan), qhzheng@mail.xjtu.edu.cn (Q. Zheng), tliu@sei.xjtu.edu.cn (T. Liu), lidanwang@stu.xjtu.edu.cn (L. Wang), yqhou@sei.xjtu.edu.cn (Y. Hou), zijiang.yang@wmich.edu (Z. Yang).

http://dx.doi.org/10.1016/j.jss.2015.06.015
0164-1212/

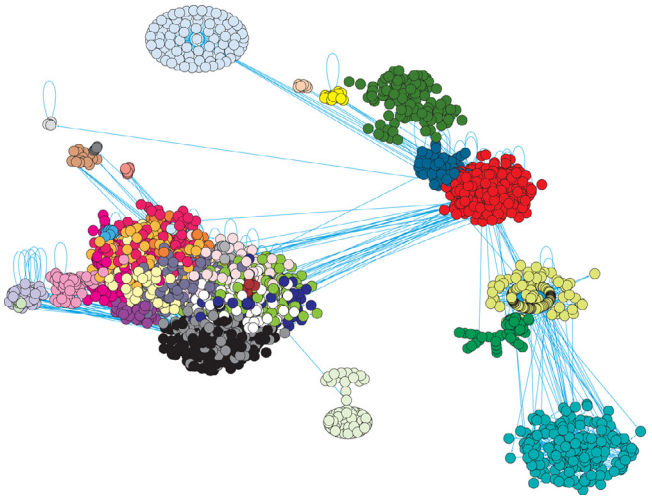

Fig. 1. Community structure of jEdit with 5979 nodes and 34 communities, detected by *Louvain* algorithm (Blondel et al., 2008).

there can be a possibility that the approaches are adopted by software engineering practitioners.

Do software networks at other granularities also present significant community structures? If so how can we make use of it in software engineering practices? To answer these open questions and solve the existing problems, we construct static Call Graphs, in which nodes represent methods in an OO (Object-Oriented) program and edges represent methods invocation relations. We then apply existing community detection algorithms to such graphs. Fig. 1 depicts the community structure of jEdit, an open-source text editor. There are 5979 nodes that are divided into 34 communities shown in different colors. The community structure is detected by *Louvain* algorithm (Blondel et al., 2008) that is implemented in a network analysis and visualization tool called Pajek.[1] In Section 3, we show that such result presents typical community characteristics similar to those previously observed in other complex systems.

It is well known that high-quality software should exhibit "high cohesion and low coupling" nature. Software with such nature is believed to be easy to understand, modify, and maintain (Briand et al., 2001; Pressman, 2010). Object-oriented design strives to incorporate data and related functionality into modules, which usually reduces coupling between modules. However, employing object-oriented mechanism itself does not necessarily guarantee minimal coupling and maximal cohesion. Therefore, a quantitative measurement is valuable in both *a posteriori* analysis of a finished product to control software quality, and *a priori* analysis to guide coding in order to avoid undesirable results in the first place.

The existence of community structures, as confirmed by our experiments on 10 large open-source Java programs using four widely-used community detection algorithms, sheds light on the cohesiveness measurements of OO programs. Intuitively, community structures are able to indicate cohesion as nodes within a community are highly cohesive, and nodes in different communities are loosely coupled. In this paper, we propose two new class cohesion metrics—*MCC* (***M**ethod **C**ommunity **C**ohesion*) and *MCEC* (***M**ethod **C**ommunity **E**ntropy **C**ohesion*) based on community structures. The basic idea of *MCC* is to quantify how many methods of a certain class reside in the same community. As for *MCEC*, it uses the standard notion of *Information Entropy* (Shannon, 2001) to quantify the distribution of all the methods of a class among communities. Comparing with existing metrics, these two metrics provide a new and more systematic point of view for class cohesion measurement.

Fig. 2 gives the overview of our approach. Once a Call Graph is constructed, we apply widely-used community detection algorithms. Fig. 2 shows the static Call Graph of JHotDraw, a Java GUI framework for technical and structured graphics. There are 5125 nodes divided into 35 communities, as reported by Louvain algorithm (Blondel et al., 2008). Based on the community structures, the metrics of *MCC* and *MCEC* are computed.

We validate the proposed metrics using the following processes. Firstly, we show that *MCC* and *MCEC* theoretically satisfy expected properties of class cohesion metrics (Briand et al., 1998). Secondly, we empirically compare *MCC* and *MCEC* with five widely-used class cohesion metrics and our experiments indicate that the new metrics are more reasonable than existing ones. Thirdly, *Principle Component Analysis* (PCA; Pearson, 1901) is conducted to show that *MCC* and *MCEC* provide additional and useful information of class cohesion that is not reflected by existing metrics. Finally, experiments are carried out to show that *MCC* and *MCEC* usually perform equally or better than existing class cohesion metrics when they are used in software fault prediction.

In summary we make the following contributions in this paper:

1. We show through experiments on 10 large open-source Java programs that the static Call Graphs constructed from OO programs usually exhibit relatively significant community structures

[1] http://vlado.fmf.uni-lj.si/pub/networks/Pajek/

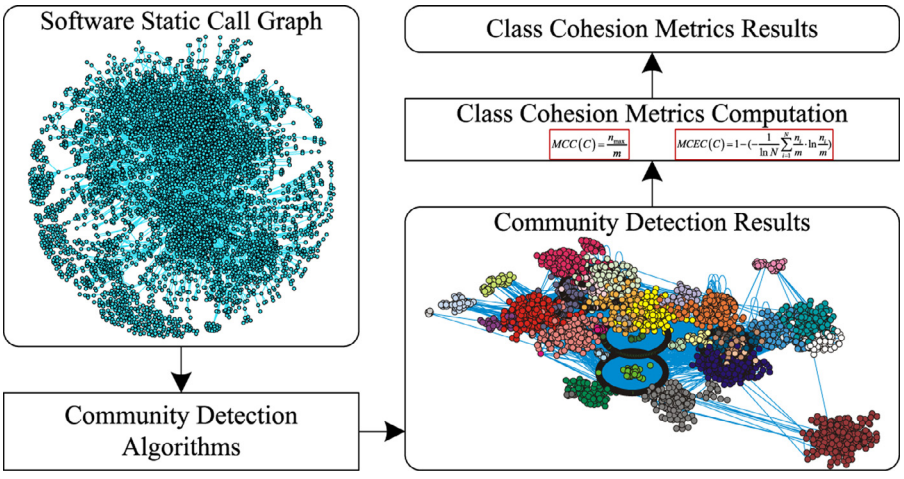

Fig. 2. The proposed approach of this paper, the static Call Graph and community structure of a Java GUI framework for technical and structured graphics—JHotDraw, is depicted in this figure.

as other networked complex systems (e.g., social networks). Such results are helpful in intra-class structure and quality evaluation.
2. Based on community structures of Call Graphs, we propose two new class cohesion metrics. We conduct study to confirm the proposed metrics satisfy the theoretical requirements of cohesion metrics. The comparison with five existing metrics shows that the class cohesion metrics based on community structures can provide new insight of OO programs.
3. We conduct empirical study and illustrate the effectiveness of the new metrics through software fault prediction experiments on four open-source programs with 1500 classes, among which there are 702 faulty ones. Results show that the new metrics usually perform equally or better than existing ones.

The rest of this paper is organized as follows. Section 2 reviews related work. In Section 3, community structures of 10 large open-source programs are investigated using four community detection algorithms. Two class cohesion metrics based on community structure are proposed in Section 4. Section 5 conducts empirical evaluations of the class cohesion metrics, followed by discussions on community detection algorithms and potential applications of the proposed metrics in Section 6. Finally Section 7 concludes the paper with future work.

2. Related work

2.1. Community structure of software

The significant progress of *Complex Network* theory (Barabási and Albert, 1999; Chakrabarti and Faloutsos, 2006; Watts and Strogatz, 1998), which was originally developed in Physics and Data Science, leads to wide adoption in different domains (Fortunato, 2010). In recent years, the theory has been successfully applied in the domain of software engineering, including software evolution process modeling and understanding (Li et al., 2013; Pan et al., 2011; Turnu et al., 2011), software evolution prediction (Bhattacharya et al., 2012), software structure interpretation and evaluation (Baxter et al., 2006; Louridas et al., 2008; Myers, 2003; Potanin et al., 2005), software execution process and behavior modeling (Cai and Yin, 2009; Qu et al., 2015), etc. These researches have revealed that networks that are constructed from software systems usually have *Scale-free* degree distributions (Baxter et al., 2006; Louridas et al., 2008; Myers, 2003; Potanin et al., 2005), and exhibit *Small-world* properties (Myers, 2003; Qu et al., 2015; Valverde and Solé, 2007), like other typical complex networks. Researchers have also found that the evolution processes of these networks are in accordance with the predictions made by Complex Network theory (Li et al., 2013; Pan et al., 2011; Turnu et al., 2011). These results have shown that software systems usually exhibit typical characteristics of complex network systems, thus have laid a good foundation for us to further use community detection algorithms that originally developed in Complex Network theory to analyze software systems.

Researchers have investigated community structures in software (Concas et al., 2013; Pan et al., 2011; Šubelj and Bajec, 2011; 2012; Šubelj et al., 2014). Typically a program is firstly converted to a certain type of software network such as Class Dependency Network (Šubelj and Bajec, 2011), where the nodes denote classes, and edges denote relationships between classes. Such relationships include aggregation, inheritance, interface implementation, parameter types, etc. There are also existing discussions on applications of software's community structure. For instance, Šubelj and Bajec (2012) proposed using community detection results in highly modular package identification. Pan et al. (2011) proposed exploiting community results to identify refactoring point in software evolution process by observing evolving trends of modularity (Newman and Girvan, 2004), a metric which was originally proposed to measure the quality or significance of a community structure. Very recently, Šubelj et al. (2014) showed that besides community structures, Class Dependency Networks also consist of groups of structurally equivalent nodes denoted modules, core/periphery structures and others. Similar investigation on Call Graphs is also need in future research. Actually, such results do not contradict the proposed approaches in this paper. The heterogeneous distribution of community structures can result in discriminative nature of the proposed metrics. Refactoring and further analysis can be done on modules with low metric scores.

Comparing with existing software networks, our approach constructs software network from a different perspective. We extract Call Graph from a program, where nodes represent methods and edges represent method invocations. Our study confirms the existence of relatively significant community structure in Call Graphs. Existing works have proposed pioneering and promising applications in this interdisciplinary research direction, but direct comparison with existing baseline metrics and practices is still needed to convince both software engineering academia and industry to adopt the proposed approaches. To the best of our knowledge, our work is the first one that utilizes community structures to measure class cohesion, a traditional and widely-used metric in software engineering. Moreover, experiments have been conducted to show that the proposed metrics perform equally or better than existing metrics in software fault prediction.

2.2. Class cohesion metrics

Classes are the basic ingredients of an OO program. Many class cohesion metrics have been proposed to quantify the relatedness of a class's attributes and methods. As proposed by Chen et al. (2002), when measuring a class's cohesiveness, the relationships between attributes and attributes, attributes and methods, methods and methods should be considered simultaneously. We believe that the community structure of Call Graph reflects methods' relations from a systematic and effective perspective. In this paper we will not give a detailed survey or taxonomy on class cohesion metrics. For a relatively comprehensive survey on these metrics, please refer to Al Dallal (2012); 2013). Generally speaking, most of the metrics leverage structural information of the class under test (Al Dallal and Briand, 2012; Briand et al., 1998; Chen et al., 2002; Chidamber and Kemerer, 1991; 1994; Sellers, 1996). They measure relationships among the methods of a class considering whether these methods share same attributes or whether there are similarities between each pair of methods. There are some metrics (C3; Marcus et al., 2008 and MWE; Liu et al., 2009) that consider semantic information of the class under test. They extract semantically meaningful topics or concepts implemented in classes. There are also some metrics (TCC; Bieman and Kang, 1995, DC_D and DC_I; Badri and Badri, 2004) reflecting method call relationships of the class under test, but these metrics only consider microscopic method call relationships. On the other hand, our metrics investigate method call relationships from a more systematic point of view. Actually, the metrics proposed in this paper take the whole system's method call relationships into consideration, thus can reflect class cohesion information from a new and systematic perspective.

3. Community structure of software Call Graphs

3.1. Call Graphs

For an OO program P, its Call Graph CG_P is a directed graph: $CG_P = (V, E)$, where each node $v \in V$ represents a method in P, and the edge set E represents the method invocation relationships. Let m_i denotes the method that v_i refers to. Then $v_i \rightarrow v_j \in E$ if and only if m_i has at least one method invocation that calls m_j.

Table 1
Subject software systems.

Programs	Version	SLOC	# Class	# Method	Website
Ant	1.9.3	106,292	1280	10,509	http://ant.apache.org/
Apache POI	3.10.1	245,326	2949	24,700	http://poi.apache.org/
Eclipse Link	2.5.1	450,766	4339	51,631	http://www.eclipse.org/eclipselink/
jEdit	5.1.0	117,365	1291	7844	http://www.jedit.org/
JGroups	3.4.3	71,613	813	7596	http://www.jgroups.org/
JHotDraw	7.6	80,515	1068	7699	http://www.jhotdraw.org/
Log4j	2.0 (RC)	56,112	986	5220	http://logging.apache.org/log4j/
Lucene	4.7.1	441,685	5613	27,720	http://lucene.apache.org/
Tomcat	8.0.5	207,676	2359	19,253	http://tomcat.apache.org/
Xalan	2.7.2	175,006	1279	10,479	http://xalan.apache.org/

To empirically study community structures of software Call Graphs, a data set including 10 widely-used open-source Java programs is collected, as shown in Table 1: Ant is a Java library and command-line tool for automating software build processes; Apache POI is a Java API to process Microsoft Office files; Eclipse Link is a comprehensive persistence service project that provides interactions with various databases and data services; jEdit is a text editor and JHotDraw is a GUI framework for graphics, both of which are mentioned in Section 1; JGroups is a reliable multicast and messaging system; Log4j is a Java-based logging library; Lucene is a searching and information retrieval library; Tomcat is a web server and servlet container; Xalan is a library for processing XML documents. Table 1 summarizes basic information of these programs. Column Version gives the version number of each program ("RC" for Log4j represents "Release Candidate", a version very close to final release). Columns SLOC, # Class and # Method list the static lines of code, the number of classes, and the number of methods, respectively. The last column shows the websites of these programs.

It is a nontrivial task to construct a relatively complete and representative Call Graph. Three important and intertwined issues should be addressed:

3.1.1. Incomplete Call Graph

It is very difficult to construct a **complete** Call Graph solely based on static analysis. This is mainly caused by the problem that modern software systems intensively use frameworks like Spring[2] to decouple interactions between components, and frameworks like Swing[3] to handle GUI (Graphical User Interface) massages, which can also decouple different components' interactions. This problem can be mitigated by using dynamic monitoring techniques (Qu et al., 2015). However, dynamic monitoring itself is not the silver bullet to solve this problem. The main shortcoming of dynamic monitoring is the lack of a complete test case set that can drive the program to execute all of the method calls. In this paper we focus on static approach that ignores implicit method invocations enabled by modern frameworks.

3.1.2. Special classes

When constructing Call Graph, two special kinds of classes should be noticed. Fig. 3 gives two example Java code snippets of Ant 1.9.3. As shown in Fig. 3(a), the first one is an *empty class*, which has no methods or attributes. The second special kind is the *anonymous inner class*, as shown in Fig. 3(b). The anonymous inner class is usually used by programmers to save programming efforts and also provides a convenient way to define callbacks.

In the analysis of Call Graph and the corresponding classes, these two special kinds should be excluded. For empty classes, it is straightforward to exclude them. For anonymous inner classes, it should be noticed that they also have methods that forming the Call Graph's structure. In our analysis, methods in anonymous inner classes are virtually moved to their resided class. Take the anonymous inner class in Fig. 3(b) for example, the `accept` method is moved to its resided class `Diagnostics`. Then the total method number (which is useful in the following analysis) of the resided class is increased correspondingly. If a class does not belong to these special kinds, it is called an *effective class* in our approach. In the following of this paper, the analysis concentrates on effective classes.

```java
package org.apache.tools.ant.taskdefs;
//Comments...
public class Typedef extends Definer {
}
```

(a)

```java
package org.apache.tools.ant;
//Comments...
public final class Diagnostics {
    //...
    private static File[] listJarFiles(File libDir) {
        // anonymous inner class
        FilenameFilter filter = new FilenameFilter() {
            public boolean accept(File dir, String name) {
                return name.endsWith(".jar");
            }
        };// anonymous inner class
        File[] files = libDir.listFiles(filter);
        return files;
    }
    //...
}
```

(b)

Fig. 3. Example code snippets of special classes of Ant 1.9.3.

Table 2 shows statistics of subject programs' classes and methods. Column # Class reviews the total number of classes given in Table 1. Column # Anony Class lists the number of anonymous inner classes. Column # Class no M & A shows the number of empty classes. The 5th column in Table 2 shows the number of effective classes. The last two columns review the number of methods and the number of methods in anonymous inner classes, respectively.

3.1.3. Disconnected components of Call Graph

It is a common situation that the constructed Call Graph is not a connected graph. Table 3 gives statistics of the Call Graphs and the corresponding ***L**argest (weakly)**C**onnected**C**omponent**s* (LCCs; Dill et al., 2002). Columns N_{CG} and E_{CG} give the number of nodes and edges of each Call Graph, respectively. Columns N_{LCC} and E_{LCC} show the corresponding results of each LCC. $C_{LCC-Related}$ and $C_{LCC-Resided}$ are two sets whose elements are classes. For a class C, if and only if at least

[2] http://spring.io/
[3] http://docs.oracle.com/javase/tutorial/uiswing/

Table 2
Statistics of 10 software systems' classes and methods.

Programs	# Class	# Anony Class	# Class no M & A	# Effect Class	# Method	# Anony Method
Ant	1280	99	21	**1160**	**10,509**	140
Apache POI	2949	298	26	**2625**	**24,700**	376
Eclipse Link	4339	190	105	**4044**	**51,631**	497
jEdit	1291	198	3	**1090**	**7844**	229
JGroups	813	98	8	**707**	**7596**	120
JHotDraw	1068	313	8	**747**	**7699**	490
Log4j	986	42	14	**930**	**5220**	45
Lucene	5613	1204	56	**4353**	**27,720**	1974
Tomcat	2359	193	16	**2150**	**19,253**	247
Xalan	1279	132	13	**1134**	**10,479**	144

Table 3
Statistics of Call Graphs' LCCs.

Programs	N_{CG}	E_{CG}	N_{LCC}	E_{LCC}	$\lvert C_{LCC-Related}\rvert$	$\lvert C_{LCC-Resided}\rvert$	$\lvert C_{LCC-Cohesion}\rvert$ $(t = 0.6)$
Ant	8016	17,296	7393	16,833	1034	332	696
Apache POI	20,181	43,558	19,299	42,786	2315	1069	1911
Eclipse Link	41,871	110,095	40,062	108,832	3401	1360	2726
jEdit	6719	15,443	5979	14,653	921	502	767
JGroups	5761	12,237	5194	11,880	628	190	451
JHotDraw	5879	12,205	5125	11658	643	194	453
Log4j	4088	8320	3744	8054	777	392	590
Lucene	22,165	61,604	20,849	60,566	3866	1963	3083
Tomcat	14,073	28,952	12,868	28,024	1780	762	1291
Xalan	7864	15,578	7078	14,988	846	381	643

one of its method appears in LCC, then $C \in C_{LCC-Related}$; if and only if all of its methods appear in LCC, then $C \in C_{LCC-Resided}$. The sixth and seventh columns in Table 3 show the cardinalities of these class two sets. The last column in Table 3 shows the cardinality of another class set—$C_{LCC-Cohesion}$, which is explained later.

The following observations can be made based on Table 3: (1) The static Call Graph is indeed incomplete as there are fewer methods in Call Graph comparing with the total method quantity shown in Table 2. (2) The LCC usually contains the majority (usually more than 90%) of the nodes in Call Graph. Based on this observation, we will concentrate on LCC in each Call Graph as the basis for further analysis. Such choice is also a common practice in network analysis (Leskovec et al., 2008) and previous related research (Šubelj and Bajec, 2011). (3) Based on $C_{LCC-Related}$ and $C_{LCC-Resided}$, it can be observed that most of the classes have at least one method in Call Graph's LCC (comparing with the number of effective classes in Table 2), but only quite a few classes have all their methods appear in LCC. The distribution of a certain class's methods in LCC is of great importance for further analysis. This issue is discussed in more detail in the following.

The proposed class cohesion metrics are computed based on the distributions of a class's methods among detected communities. For a certain class, if there is only a small proportion of its methods reside in LCC, then the cohesion computation result will be bias as it cannot reflect its major method call relations. It is reasonable to only analyze classes in $C_{LCC-Resided}$. However, considering the small quantity of classes in $C_{LCC-Resided}$, such option will hinder the applicability of the proposed approach. Thus, a trade-off should be made between accuracy and applicability. For a class C, suppose its total number of methods is M, and there are m methods located in LCC. We define a threshold t, and a new class set $C_{LCC-Cohesion}$, then $C \in C_{LCC-Cohesion}$ if and only if $\frac{m}{M} \geq t$. The cardinality of $C_{LCC-Cohesion}$ depends on t, and the following relations can be easily derived:

$$C_{LCC-Cohesion} = \begin{cases} C_{LCC-Related}, & \text{when } t = 0, \\ C_{LCC-Resided}, & \text{when } t = 1. \end{cases}$$

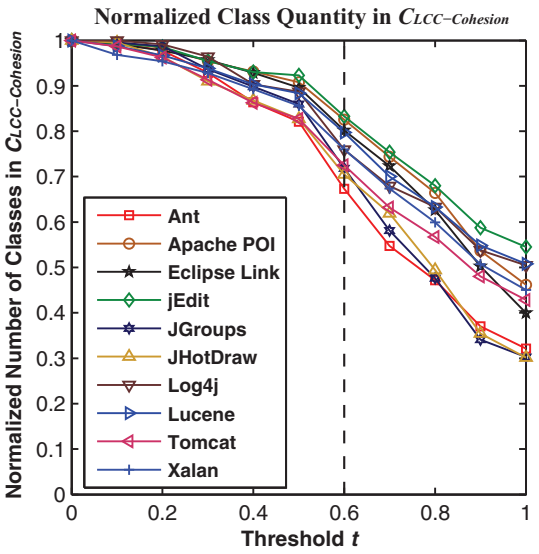

Fig. 4. Normalized method quantity in $C_{LCC-Cohesion}$ versus different values of t.

Fig. 4 shows the normalized class quantity in $C_{LCC-Cohesion}$ versus different values of t. The number of classes in $C_{LCC-Cohesion}$ is normalized by the number of classes in $C_{LCC-Related}$ for each program. It can be observed that these programs exhibit similar tendency when the value of t varies. In this paper, we decide to use $t = 0.6$ as the threshold, i.e., the cohesion metrics of classes in $C_{LCC-Cohesion}$ when $t = 0.6$ are computed. That is, a class's cohesiveness is computed only if there are more than 60% of its methods appear in LCC. It has to be clarified that we only choose part of classes in LCC to compute their cohesiveness, but the community detection is conducted on the whole LCC. Although some classes are excluded from the cohesiveness measurement process, their methods still located in LCC and also influence the detected community structure.

Table 4
Community detection results of 10 software systems.

Programs	# Com_{fg}	Q_{fg}	# Com_{im}	Q_{im}	# Com_{lp}	Q_{lp}	# Com_{ml}	Q_{ml}
Ant	109	0.679	600	0.609	287	0.528	42	0.706
Apache POI	200	0.789	1160	0.725	842	0.751	59	**0.855**
Eclipse Link	425	0.716	2109	0.662	1110	0.714	60	0.813
jEdit	106	0.698	475	0.632	154	0.622	45	0.753
JGroups	106	0.649	442	0.594	183	**0.339**	57	0.676
JHotDraw	54	0.772	378	0.701	236	0.729	34	0.798
Log4j	51	0.754	302	0.707	225	0.718	38	0.785
Lucene	263	0.664	1276	0.649	663	0.471	81	0.740
Tomcat	148	0.747	952	0.673	498	0.646	67	0.794
Xalan	63	0.804	541	0.704	362	0.753	38	0.825

3.2. Community structure detection

Four widely-used community detection algorithms are implemented in this paper. Particularly, **f**ast **g**reedy (***fg***) is based on greedy optimization of modularity (Clauset et al., 2004); **i**nfo**m**ap (***im***) detects community structure of a network based on the *Infomap* method proposed by Rosvall and Bergstrom (2008); **l**abel **p**ropagation (***lp***) is a fast partitioning algorithm proposed by Raghavan et al. (2007); **m**ulti**l**evel (***ml***) is a layered and bottom-up community detection algorithm given by Blondel et al. (2008). It has to be noticed that the *ml* algorithm is usually referred to as, as mentioned in previous sections, Louvain algorithm. Louvain is a city in Belgium, and the algorithm is named after its authors' location. Nevertheless, in the following part of this paper, we still name it as *ml* to keep a consistent naming convention. It should also be noticed that most of the community detection algorithms work on undirected graphs. In the analysis on software Call Graphs and other networks in this paper, the directed graph is converted to its simple undirected version by removing the edge direction.

Implementation of these algorithms is based on the open-source network analysis package *igraph*.[4] The open-source Python software NetworkX,[5] a package for the computation of the structure, dynamics, and functions of complex networks, has been re-developed to conduct some of the network data analysis tasks.

In order to quantify the quality of a detected community structure, the notion of *modularity* Q (Newman and Girvan, 2004) has been proposed. Given a network with k communities, we can generate a $k \times k$ matrix where element e_{ij} represents fraction of edges that connect nodes in communities i and j. Note that $\sum_i e_{ii}$ denotes the fraction of edges that connect nodes in the same community, and the sum of column i, $a_i = \sum_j e_{ij}$, represents the fraction of edges that connect to nodes in community i. Modularity Q is defined as:

$$Q = \sum_i \left(e_{ii} - {a_i}^2\right)$$

According to the definition of Q, if the edges in a network are randomly distributed among communities, the value of Q approaches 0. On the other hand, Q is close to 1 with an ideal community structure. It has been reported that the typical value of Q in the domain of Physics is between 0.3 and 0.7, and "higher values are rare" (Newman and Girvan, 2004).

Table 4 gives the community detection results using the aforementioned four algorithms. For each algorithm we report the number of communities and the Q values. Most of the Q values are between 0.6 and 0.9, and the maximum and minimum values are 0.855 and 0.339, respectively. On the other hand, community detection algorithms based on Class Dependency Networks (Šubelj and Bajec, 2011) and similar software networks (Concas et al., 2013; Pan et al., 2011) have reported relatively lower Q values.

It can be noticed that the number of detected communities in Table 4 present significant differences among algorithms. In Section 4, we show that these algorithms tend to obtain similar class cohesion measurement results although they have different community detection results.

Although the notion of modularity was originally proposed to measure the significance of a detected community structure, it has been realized that this metric is insufficient. Researchers have shown that high values of modularity do not necessarily indicate that a network has a significant community structure (Fortunato, 2010; Karrer et al., 2008), although it is true that networks with strong community structure have high modularity. For instance, Guimera et al. (2004) showed that ordinary random graphs may also exhibit high modularity. Good et al. (2010) showed that maximizing modularity is ineffective for partition problems in many networks. Thus, a more careful and thorough study should be performed to investigate whether software Call Graphs have significant community structure.

Based on the preceding understandings, researchers have proposed other approaches to measuring the significance of communities. A group of approaches have been proposed with the basic understanding that the significance of community can be quantified by the *robustness* or *stability* of community structure against random *perturbations* (Hu et al., 2010; Karrer et al., 2008; Lancichinetti et al., 2010). Intuitively, if a network has significant community structure, such structure should be robust to perturbation. In this paper, we use the perturbation method proposed by Hu et al. (2010) to measure the significance of Call Graphs' community structures. Briefly, the approach works as follows:

(1) *Perturbing the network*. To conduct a perturbation on a network, edges are randomly removed and then added with a probability p. When an edge is removed, a new edge is randomly added between another node pair. The larger the value of p is, the more significant perturbation is performed on the original network. When $p = 1$, a random graph that is uncorrelated with the original network is generated.
(2) *Measuring the similarity between original network and the perturbed one*. Once a perturbed network is obtained, the original network and the perturbed network's community structures are detected using a certain algorithm. Then the *Normalized Mutual Information* (NMI) (Danon et al., 2005) is used to quantify the similarity between these two community structures. NMI of two identical community structures is 1, and is 0 if the two structures are independent. A similarity score is computed:

$$S(p) = I(A, A(p)) - I(A_r, A_r(p))$$

where $S(p)$ is the similarity score with p, A and $A(p)$ are the community structures before and after perturbation. $I(A, A(p))$

[4] http://igraph.org/python/
[5] http://networkx.github.io/

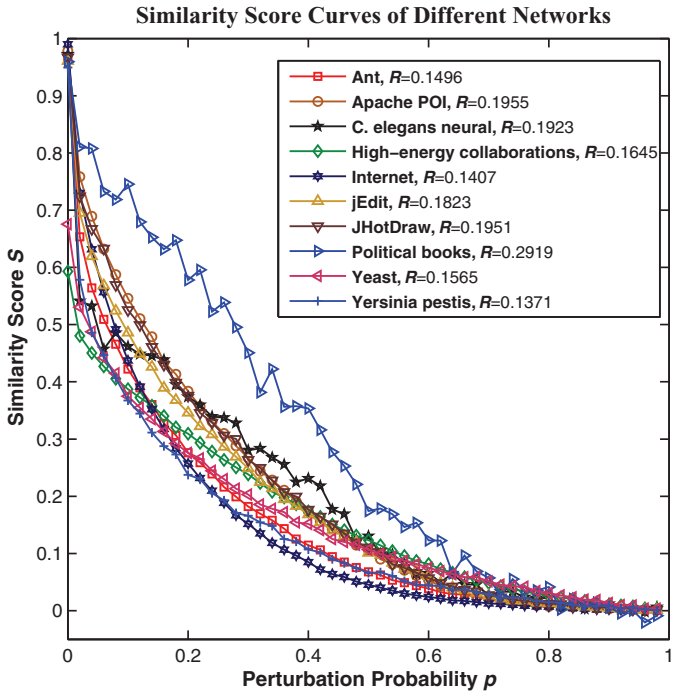

Fig. 5. Similarity score curves showing the network perturbation experiment results of 10 networks.

is the NMI between A and $A(p)$. A_r and $A_r(p)$ are two community structures that have same number of communities with the same number of nodes in each community as A and $A(p)$ respectively, the only difference is that the nodes in each community in A_r and $A_r(p)$ are randomly selected from the entire set of nodes. A_r and $A_r(p)$ are introduced to eliminate the influence of the random background and effects of network size. Thus, networks with different sizes can be directly compared.

(3) *Index R from integrating the similarities*. Using the aforementioned steps, a series of $S(p)$ is obtained by gradually increasing the probability p from 0 to 1. The simulation should be performed several times to obtain the expectation of $S(p)$. Then an index R is computed by integrating all the expected values of $S(p)$:

$$R = \int_0^1 E[S(p)]dp$$

where $E[S(p)]$ is the expectation of $S(p)$. If a network has a significant community structure, then its R is relatively high.

Using this perturbation approach, the Call Graphs of subject programs have been analyzed. Other networks that have been widely used in previous Complex Network research have also been analyzed. We used 0.02 as the step-size to increase p gradually, and for each network, the simulation has been performed for 10 times to obtain the expectation of $S(p)$. The *ml* algorithm was used to detect community structure in this experiment.

Fig. 5 shows the curves of $E[S(p)]$ of 10 networks, including software Call Graphs, neural networks, social networks, etc. Fig. 5 also reports the corresponding R values of these networks, actually, R value is equal to the area under a certain curve in Fig. 5. Table 5 shows all the R values of other investigated networks.[6] For each network, we report its number of nodes and number of edges, then the R value followed by the network's type. Table 6 lists R values of the subject programs.

Table 5
Perturbation results—R values of other types of networks.

Networks	N	E	R	Type
Yeast	1870	2277	0.1565	Protein
Aquifex aeolicus	1057	2503	0.1451	Metabolic
Helicobacter pylori	949	2291	0.1441	
Yersinia pestis	1453	3403	0.1371	
C. elegans neural	297	2151	0.1923	Neural
Internet	22,963	48,436	0.1407	Technical
Co-authorships in network science	1589	2742	0.1867	Social
High-energy theory collaborations	8361	15,751	0.1645	
Political books	105	441	0.2919	
Zachary's karate club	34	78	0.2252	

Table 6
Perturbation results—R values of 10 software systems.

Programs	R	Programs	R
Ant	0.1496	JHotDraw	0.1951
Apache POI	0.1955	Log4j	0.1841
Eclipse Link	0.1951	Lucene	0.1772
jEdit	0.1823	Tomcat	0.1755
JGroups	0.1477	Xalan	0.1926

In previous study, social networks, internet and the C. elegans neural network have been believed to have significant community structures (Fortunato, 2010; Hu et al., 2010). Protein networks and metabolic networks usually have weak or fuzzy community structures. Results in Table 5 are consistent with previous research although different community detection algorithms have been used (Hu et al., 2010). Based on Tables 5 and 6, it can also be noticed that R values of software Call Graphs are usually close or equal to social networks' and C. elegans neural network's, and are noticeably larger than those of metabolic and protein networks. Based on these result, we can make a conclusion that software Call Graphs usually have relatively significant community characteristics that are similar to other complex networks which have exhibited significant community structures.

4. Class cohesion measurement based on community detection

4.1. New class cohesion metrics

In this section we propose two class cohesion metrics based on software community structures.

Definition 1. *Method Community Cohesion* (*MCC*): Given a class C with m methods located in LCC, after applying a certain community detection algorithm, these m methods distribute in N communities. For the ith community, there are n_i methods belonging to C ($1 \le i \le N$). Let $n_{\max} = \max\{n_i\}$. We define

$$\mathrm{MCC}(C) = \begin{cases} 1, & \text{if } m = 1, \\ 0, & \text{if } n_{\max} = 1 \text{ and } m \ge 2, \\ \frac{n_{\max}}{m}, & \text{otherwise.} \end{cases} \quad (1)$$

The definition of *MCC* describes the largest portion of its methods that reside in the same community, which represents more cohesive relation than the rest of the methods in C. The second line in equation (1) is proposed to make sure that the lower bound of *MCC* is consistent for different classes and is not influenced by the number of methods in a class. For instance, suppose class C_1 has three methods that are distributed in three communities and class C_2 has four methods that are distributed in four communities, then the cohesiveness of these two classes should all reach the lower bound of *MCC*,

[6] The network data of Yeast, Aquifex aeolicus, Helicobacter pylori and Yersinia pestis are provided at the webpage: http://www3.nd.edu/~networks/resources.htm by Albert-László Barabási and the rest of networks are provided at the webpage: http://www-personal.umich.edu/~mejn/netdata/ by Mark Newman. For the explanations of these networks, please refer to these webpages.

rather than 1/3 and 1/4 respectively, according to the second line in equation (1). The value of *MCC* is in the interval [0, 1].

Definition 2. *Method Community Entropy Cohesion* (*MCEC*): With the same symbols in Definition 1, We define

$$\mathrm{MCEC}(C) = \begin{cases} 1, \text{if } N = 1, \\ 1 - (-\frac{1}{\ln N} \sum_{i=1}^{N} \frac{n_i}{m} \cdot \ln \frac{n_i}{m}), \text{if } N \geq 2. \end{cases} \quad (2)$$

The methods reside in a single community if $N = 1$, therefore the value of *MCEC* is 1. $-\frac{1}{\ln N} \sum_{i=1}^{N} \frac{n_i}{m} \cdot \ln \frac{n_i}{m}$ in equation (2) represents the normalized Information Entropy (Shannon, 2001) of methods distribution of *C*. The value of Entropy equation is 1 if all the methods of *C* evenly distribute in every community; it approaches 0 if almost all the methods reside in a single community. That is, larger Entropy values represent more even distribution. We use 1 minus the Entropy value in equation (2). With an interval [0, 1], *MCEC* achieves its upper bound when all the methods reside in a single community.

As previously mentioned, throughout the paper, *MCC* and *MCEC* are computed based on the community structure of LCC of Call Graph. They are computed for classes in $C_{\mathrm{LCC-Cohesion}}$ when $t = 0.6$, which means that a class's cohesiveness is computed if and only if there are more than 60% of its methods appear in LCC.

4.2. Theoretical properties of cohesion metrics

Briand et al. (1998) proposed that well-defined class cohesion metrics should have the following mathematical properties:

- *Nonnegativity and normalization*: The cohesion measure of a class *C* should belong to a specific interval [0, Max]. Normalization allows for direct comparison between cohesion measures of different classes.
- *Null value and maximum value*: The cohesion measure of a class *C* equals 0 if the class has no cohesive relations, and the cohesion measure is equal to Max if all possible interactions are presented.
- *Monotonicity*: The cohesion measure does not decrease if class *C*'s interactions increase.
- *Cohesive modules*: If two independent classes C_1 and C_2 are merged to a new class C_3, then the cohesion measure of C_3 should not be larger than the larger value of cohesion measures of C_1 and C_2, which means that $\mathrm{cohesion}(C_3) \leq \max\{\mathrm{cohesion}(C_1), \mathrm{cohesion}(C_2)\}$.

These properties have been widely used in theoretical investigations on the proposed class cohesion metrics (Al Dallal, 2010; Al Dallal and Briand, 2012; Briand et al., 1998; Zhou et al., 2004).

It can be easily concluded that *MCC* satisfies the *Nonnegativity and normalization* and the *Null value and maximum value* properties. *MCC* satisfies the *Monotonicity* property as well because when $n_{\max}$ increases, *MCC* increases correspondingly. In the following we prove that *MCC* satisfies the *Cohesive modules* property.

Proof. Following the notations in Definition 1, suppose there are m_1 methods in C_1 located in LCC, and m_2 methods in C_2 located in LCC. Moreover, $n_{1\max}$ and $n_{2\max}$ are the corresponding values of $n_{\max}$ in Definition 1 for C_1 and C_2 respectively. Thus, $\mathrm{MCC}(C_1) = \frac{n_{1\max}}{m_1}$ and $\mathrm{MCC}(C_2) = \frac{n_{2\max}}{m_2}$.

Firstly, it should be proved that if $C_1 \in C_{\mathrm{LCC-Cohesion}}$ and $C_2 \in C_{\mathrm{LCC-Cohesion}}$ $(t = 0.6)$, then their merged class C_3 also satisfies that $C_3 \in C_{\mathrm{LCC-Cohesion}}$. Suppose the total method quantities in C_1 and C_2 are M_1 and M_2. Then

$$C_1 \in C_{\mathrm{LCC-Cohesion}} \rightarrow \frac{m_1}{M_1} \geq 0.6,$$

and

$$C_2 \in C_{\mathrm{LCC-Cohesion}} \rightarrow \frac{m_2}{M_2} \geq 0.6.$$

Based on these two conditions, it can be easy derived that:

$$\frac{m_1 + m_2}{M_1 + M_2} \geq 0.6,$$

which means that $C_3 \in C_{\mathrm{LCC-Cohesion}}$.

After merging C_1 and C_2 into C_3, suppose $n_{3\max}$ is the corresponding value of $n_{\max}$ in Definition 1 for C_3. There are three cases of method distribution of C_3:

Case 1 . $n_{3\max} = n_{1\max} + n_{2\max}$. In such case, the largest proportion of C_3's methods is the union of the largest proportions of C_1 and C_2's methods. Then $\mathrm{MCC}(C_3) = \frac{n_{3\max}}{m_1+m_2}$. To prove that *MCC* satisfies the *Cohesive modules* property, it should be proved that

$$\frac{n_{3\max}}{m_1 + m_2} \leq \max\left\{\frac{n_{1\max}}{m_1}, \frac{n_{2\max}}{m_2}\right\} \rightarrow$$

$$\frac{n_{1\max} + n_{2\max}}{m_1 + m_2} \leq \max\left\{\frac{n_{1\max}}{m_1}, \frac{n_{2\max}}{m_2}\right\}$$

It can be proved using *proof by contradiction*: Supposing

$$\frac{n_{1\max} + n_{2\max}}{m_1 + m_2} > \max\left\{\frac{n_{1\max}}{m_1}, \frac{n_{2\max}}{m_2}\right\},$$

which means that

$$\frac{n_{1\max} + n_{2\max}}{m_1 + m_2} > \frac{n_{1\max}}{m_1},$$

and

$$\frac{n_{1\max} + n_{2\max}}{m_1 + m_2} > \frac{n_{2\max}}{m_2}.$$

$$\begin{aligned} &\frac{n_{1\max} + n_{2\max}}{m_1 + m_2} > \frac{n_{1\max}}{m_1} \\ &\rightarrow n_{1\max}m_1 + n_{2\max}m_1 > n_{1\max}m_1 + n_{1\max}m_2 \\ &\rightarrow n_{2\max}m_1 > n_{1\max}m_2, \end{aligned}$$

and

$$\begin{aligned} &\frac{n_{1\max} + n_{2\max}}{m_1 + m_2} > \frac{n_{2\max}}{m_2} \\ &\rightarrow n_{1\max}m_2 + n_{2\max}m_2 > n_{2\max}m_1 + n_{2\max}m_2 \\ &\rightarrow n_{1\max}m_2 > n_{2\max}m_1. \end{aligned}$$

Then a contradiction is derived. Thus, the original assumption is not true. So

$$\frac{n_{1\max} + n_{2\max}}{m_1 + m_2} \leq \max\left\{\frac{n_{1\max}}{m_1}, \frac{n_{2\max}}{m_2}\right\},$$

in other words,

$$\mathrm{MCC}(C_3) \leq \max\{\mathrm{MCC}(C_1), \mathrm{MCC}(C_2)\}.$$

Case 2 . $n_{3\max} = n_{1\max} + n_{22}$, where $n_{22} \leq n_{2\max}$. Such case means that the largest proportion of C_3's methods is the union of the largest proportion of C_1's methods and a proportion, which is not necessarily be the largest one, of C_2's methods. In such situation,

$$\mathrm{MCC}(C_3) = \frac{n_{1\max} + n_{22}}{m_1 + m_2} \leq \frac{n_{1\max} + n_{2\max}}{m_1 + m_2} \leq \max\left\{\frac{n_{1\max}}{m_1}, \frac{n_{2\max}}{m_2}\right\},$$

which means that

$$\mathrm{MCC}(C_3) \leq \max\{\mathrm{MCC}(C_1), \mathrm{MCC}(C_2)\}.$$

Case 3 . $n_{3\max} = n_{12} + n_{22}$, where $n_{12} \leq n_{1\max}$ and $n_{22} \leq n_{2\max}$. Such case means that the largest proportion of C_3's methods is the union of two proportions of C_1 and C_2's methods, which are not the largest ones in C_1 and C_2. Proof in such case is similar to that in *Case 2*. □

Table 7
Results of the class-merging simulation experiments on jEdit.

Community detection algorithms	*fg*	*im*	*lp*	*ml*
Percentage of class pairs violating the *Cohesive modules* property	6%	6%	7%	5%

Table 8
Correlation analysis on each pair of *MCC*s and *MCEC*s based on four community detection algorithms.

Programs	Spearman Correlation Coefficients of *MCC*s (all the *p*-values < 0.0001)						Spearman Correlation Coefficients of *MCEC*s (all the *p*-values < 0.0001)					
	fg-im	*fg-lp*	*fg-ml*	*im-lp*	*im-ml*	*lp-ml*	*fg-im*	*fg-lp*	*fg-ml*	*im-lp*	*im-ml*	*lp-ml*
Ant	0.582	0.425	0.631	0.547	0.608	**0.300**	0.522	0.398	0.586	0.528	0.547	**0.273**
Apache POI	0.492	0.570	0.617	0.770	0.594	0.625	0.399	0.473	0.569	0.720	0.510	0.552
Eclipse Link	0.560	0.534	0.475	0.643	0.407	0.395	0.529	0.554	0.462	0.654	0.342	0.375
jEdit	0.671	0.474	0.776	0.453	0.683	0.473	0.648	0.474	0.766	0.445	0.667	0.479
JGroups	0.737	0.369	0.807	0.479	0.689	0.344	0.716	0.365	0.756	0.466	0.642	0.346
JHotDraw	0.556	0.517	0.745	0.659	0.643	0.516	0.553	0.477	0.720	0.626	0.606	0.487
Log4j	0.725	0.635	0.783	**0.814**	0.778	0.682	0.692	0.607	0.763	0.792	0.747	0.644
Lucene	0.661	0.473	0.678	0.569	0.622	0.370	0.653	0.468	0.667	0.559	0.598	0.367
Tomcat	0.592	0.431	0.742	0.661	0.612	0.456	0.555	0.423	0.726	0.659	0.564	0.453
Xalan	0.633	0.687	0.796	0.810	0.715	0.718	0.626	0.689	0.770	**0.808**	0.691	0.715

It can be easily concluded that *MCEC* satisfies the *Null value and maximum value*. Null value happens when every single method distributes in a unique community, and maximum value can be achieved if all the methods are in the same community. It can also be easily concluded that *MCEC* satisfies the *Nonnegativity and normalization* and the *Monotonicity* properties.

However, we are not able to theoretically prove or disprove the *Cohesive modules* property of *MCEC*. Instead we conduct empirical study to examine the property.

For each community detection result of jEdit, we randomly choose 100 pairs of classes and then merge the pairs. *MCEC* is then calculated for the new 100 classes. Table 7 gives the percentage of class pairs that do not satisfy the *Cohesive modules* property. Based on these counterexamples, it is concluded that *MCEC* violates the *Cohesive modules* property.

In summary, *MCC* satisfies all the four properties. *MCEC* satisfies the *Nonnegativity and normalization*, the *Null value and maximum value* and the *Monotonicity* properties. Although *MCEC* does not satisfy all the expected properties proposed in (Briand et al., 1998), our empirically study show that it is more informative, and performs better in fault predictions than existing class cohesion metrics. There are also some widely-used class cohesion metrics that do not satisfy all the four properties. For instance, Zhou et al. (2004) investigated that LCOM2 (Chidamber and Kemerer, 1994) does not satisfy the *Nonnegativity and normalization* and the *Monotonicity* properties.

4.3. Correlations between different community detection algorithms

Since there are multiple community detection algorithms, we have conducted empirical study to evaluate their effects on *MCC* and *MCEC*. Table 8 gives the *Spearman Correlation Coefficient*s of each pair of *MCC*s and *MCEC*s. Spearman Correlation Coefficient is a widely-used nonparametric measure of statistical dependence between two variables (Spearman, 1904). All the *p*-values[7] are less than 0.0001, meaning that all the *MCC*s and *MCEC*s are statistically correlated with each other. Most of the Spearman Correlation Coefficients are greater than 0.5, representing that all *MCC*s and *MCEC*s have a significant positive correlation with each other. In summary, different community detection algorithms tend to give similar results when they are applied to class cohesion measurement and evaluation.

[7] In statistical significance testing, *p-value* is the probability that the "null hypothesis" is actually correct. In the computing process of *Spearman Correlation Coefficient*, the null hypothesis is that the two variables are statistically uncorrelated.

These results are interesting and need further investigation. Here we give a partial and possible explanation for these results. Table 9 shows the NMIs between each community structure pairs with the four algorithms. Results of other networks are also shown in Table 9. NMIs of Call Graphs are usually greater than 0.6 and present a significant increment comparing with NMIs of biological networks and Internet in Table 9, which means that different algorithms tend to obtain similar results to a certain extent, comparing with results obtained on biological networks and Internet. Such similarity might be one of the reasons for the results in Table 8. It can also be noticed that social networks usually present very high NMI results. Such differences also needs further research.

5. Empirical evaluation of class cohesion metrics

In our empirical study we first compare our proposed class cohesion metrics with several existing ones, followed by two case studies. The purpose of the first case study is to determine whether *MCC* and *MCEC* provide additional information comparing with other well-known metrics. The second case study is to explore whether *MCC* and *MCEC* can lead to better results in class fault prediction. These two evaluation processes have been widely used in previous studies (Al Dallal and Briand, 2012; Gyimothy et al., 2005; Liu et al., 2009; Marcus et al., 2008).

5.1. Comparisons with existing class cohesion metrics

Table 10 lists definitions of five widely-used class cohesion metrics. Coh and LSCC positively correlate with a class's cohesiveness and are in the interval [0, 1]. LCOM1, LCOM2 and LCOM5 negatively correlate with a class's cohesiveness and are regarded as "inverse cohesion metrics" (Al Dallal and Briand, 2012). LCOM1 and LOCM2 do not have an upper bound, while LCOM5 has an upper bound value of 2.

Figs. 6 and 7 depict distributions of Ant and Tomcat's class cohesion metrics, respectively.

Based on the two figures, it can be noticed that most classes have very low Coh and LSCC scores, which indicates extremely weak cohesiveness. As for LCOM5, most classes have middle scores. The results of Coh and LSCC are surprising because Tomcat and Ant are widely-used applications with very good software structure. LCOM5 shows that most classes have mediocre cohesiveness. Since it is a inverse cohesion metric, LCOM5 contradicts Coh and LSCC in that there are relatively more class with strong cohesiveness than those with week cohesiveness.

Table 9
NMI between different community detection algorithms.

Programs	*fg-im*	*fg-lp*	*fg-ml*	*im-lp*	*im-ml*	*lp-ml*
Ant	0.638	0.631	0.627	0.781	0.657	0.657
Apache POI	0.616	0.640	0.671	0.846	0.666	0.697
Eclipse Link	0.555	0.574	0.608	0.713	0.605	0.640
jEdit	0.617	0.631	0.654	0.654	0.640	0.677
JGroups	0.661	0.495	0.659	0.516	0.675	**0.464**
JHotDraw	0.650	0.661	0.717	**0.842**	0.688	0.688
Log4j	0.686	0.664	0.714	0.836	0.716	0.704
Lucene	0.613	0.598	0.639	0.705	0.637	0.613
Tomcat	0.638	0.638	0.716	0.738	0.655	0.637
Xalan	0.626	0.680	0.755	0.829	0.661	0.689
Aquifex aeolicus	0.577	0.429	0.474	0.553	0.625	0.468
Helicobacter pylori	0.568	0.427	0.529	0.431	0.621	0.413
Yersinia pestis	0.565	0.331	0.573	0.347	0.627	0.351
C. elegans neural	0.520	0.356	0.514	0.363	0.651	0.364
Internet	0.503	0.466	0.614	0.566	0.625	0.498
Co-authorships in network science	0.953	0.949	0.992	0.982	0.959	0.956
High-energy theory collaborations	0.819	0.820	0.833	0.943	0.850	0.845
Political books	0.901	0.950	0.971	0.862	0.902	0.939
Zachary's karate club	0.826	0.692	0.712	0.699	0.860	0.587

Table 10
Definitions of existing class cohesion metrics.

Class cohesion metrics	Definitions and explanations
Coh (Briand et al., 1998)	$\text{Coh} = \frac{a}{kl}$, where l is the number of attributes, k is the number of methods, and a is the summation of the number of distinct attributes that are accessed by each method in a class.
LCOM1 (Lack of Cohesion in Methods) (Chidamber and Kemerer, 1991)	LCOM1=Number of pairs of methods that do not share attributes.
LCOM2 (Chidamber and Kemerer, 1994)	$\text{LCOM2} = \begin{cases} P-Q, & \text{if } P-Q \geq 0, \\ 0, & \text{otherwise.} \end{cases}$ P=Number of pairs of methods that do not share attributes, Q=Number of pairs of methods that share attributes.
LCOM5 (Sellers, 1996)	$\text{LCOM5} = \frac{(kl-a)}{(kl-l)}$, where a, k and l have the same definitions as in the definition of Coh.
LSCC (LLD, Similarity-based Class Cohesion) (Al Dallal and Briand, 2012)	$\text{LSCC} = \begin{cases} 0, & \text{if } l=0 \text{ and } k>1, \\ 1, & \text{if } (l>0 \text{ and } k=0) \text{ or } k=1, \\ \frac{\sum_{i=1}^{l} x_i(x_i-1)}{lk(k-1)}, & \text{otherwise.} \end{cases}$ Where k and l have the same definitions as in the definition of Coh, and x_i is the number of methods that reference attribute i.

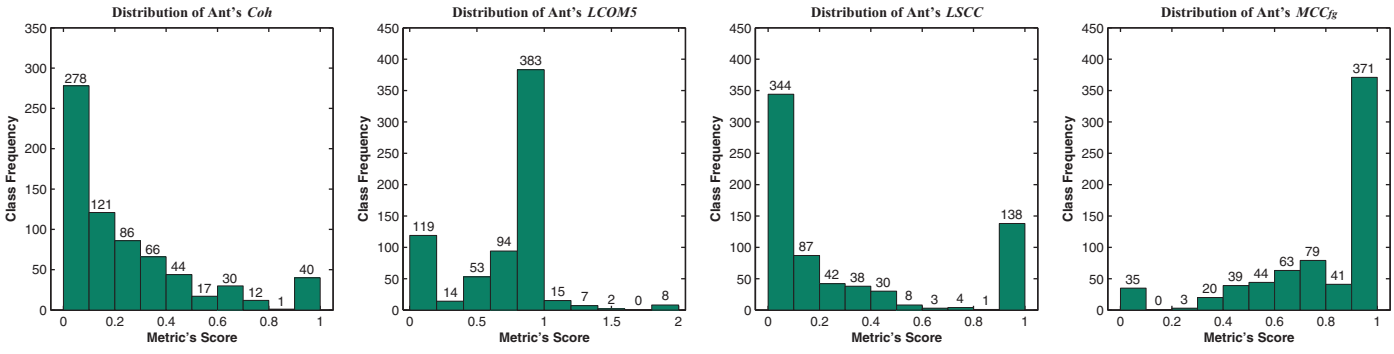

Fig. 6. Distributions of four class cohesion metrics of Ant 1.9.3.

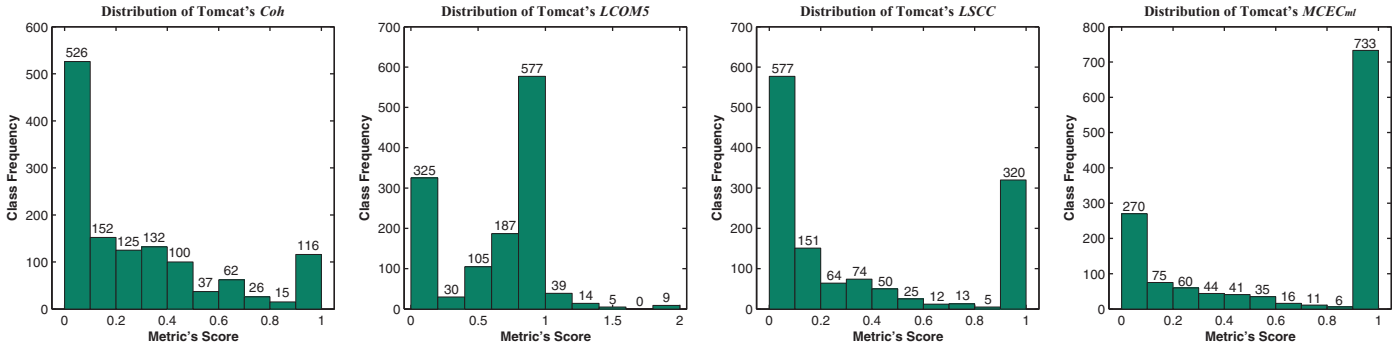

Fig. 7. Distributions of four class cohesion metrics of Tomcat 8.0.5.

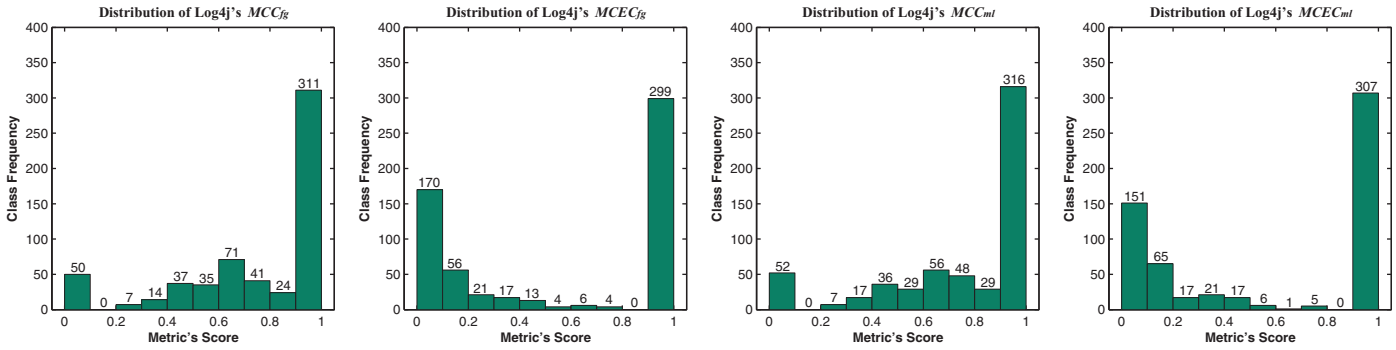

Fig. 8. Distributions of *MCC*s and *MCEC*s of Log4j 2.0.

Fig. 8 shows the distribution of *MCC* and *MCEC* scores of Log4j. The distribution is consistent with the last two sub-figures in Figs. 6 and 7. The results are as expected: most classes in the widely-used applications have strong cohesiveness. There are a few classes with weak cohesiveness, which provides guidance for developers to improve the software structure.

5.2. Principle component analysis

Following the processes of prior studies, we conducted the PCA (Pearson, 1901) on Eclipse Link 2.5.1 and Lucene 4.7.1 to determine whether the newly proposed metrics provide additional information on class cohesiveness. PCA is a statistical procedure that has been used by a number of previous researches to identify orthogonal dimensions captured by different cohesion metrics (Al Dallal and Briand, 2012; Liu et al., 2009; Marcus et al., 2008). Generally speaking, PCA can convert a set of observations of possibly correlated variables into a set of uncorrelated variables (i.e., dimensions). These uncorrelated variables are called principle components (PCs). We conducted PCA with the same settings as previous studies (Al Dallal and Briand, 2012; Liu et al., 2009; Marcus et al., 2008).

There are in total 5809 (2726+3083) classes that have been analyzed. Tables 11 and 12 give the results of PCA. Table 11 shows that for Eclipse Link, three PCs are obtained. The first three rows in Table 11 show the eigenvalue (i.e., measures of the variances of the PCs), the PCs' percentages, and the cumulative percentage. The cumulative percentage indicates that these three PCs have captured 76.001% of the data set variance. After the first three rows, each metric's coefficients for each PC are shown in the corresponding row, and important coefficients are marked in bold. It can be noticed that, for Eclipse Link, *MCC* and *MCEC* are the only two major factors (i.e., the original variables that comprise the corresponding PC) in PC2, and they capture more data variance (the larger the data variance is, the more variability of the data set is captured by the corresponding PC) than LSCC and Coh, which are major factors in PC3. Situations are similar for Lucene with a more significant and positive experiment result. These results indicate that *MCC* and *MCEC* capture an additional measurement dimension of their own. In Section 5.3, fault prediction experiments have been performed to show that the additional dimension is helpful to improve the performance of fault prediction. Thus, this new measurement dimension is also important and helpful.

Table 11
Results of PCA on Eclipse Link 2.5.1.

	PC1	PC2	PC3
Eigenvalue	2.553	2.083	1.444
Percent	31.911%	26.039%	18.051%
Cumulative percentage	31.911%	57.950%	76.001%
Coh	-0.259	0.032	**0.895**
LCOM1	**0.935**	0.138	0.181
LCOM2	**0.930**	0.138	0.182
LCOM5	0.167	0.286	0.119
LOC	**0.769**	0.119	0.020
LSCC	-0.305	0.547	**0.669**
MCC_{im}	-0.144	**0.887**	-0.279
$MCEC_{im}$	-0.118	**0.929**	-0.191

Table 12
Results of PCA on Lucene 4.7.1.

	PC1	PC2	PC3
Eigenvalue	2.592	1.944	1.476
Percent	32.402%	24.305%	18.444%
Cumulative percentage	32.402%	56.707%	75.151%
Coh	-0.365	-0.080	**0.780**
LCOM1	**0.921**	-0.047	0.272
LCOM2	**0.916**	-0.045	0.257
LCOM5	0.095	0.281	0.357
LOC	**0.780**	-0.031	0.082
LSCC	-0.367	0.334	**0.745**
MCC_{ml}	0.127	**0.932**	-0.142
$MCEC_{ml}$	0.061	**0.934**	-0.136

Table 13
Basic statistics of programs with fault data.

Programs	Version	SLOC	# Effect Class	# Methods
Ant	1.7.0	93,520	1052	9271
Apache POI	3.0	53,097	508	6141
Lucene	2.4.0	35,984	459	3838
Xalan	2.6.0	155,067	1027	9686

5.3. Fault prediction

The case study on PCA confirms that *MCC* and *MCEC* can provide additional information of class cohesion. In the second case study, the fault prediction capabilities of *MCC* and *MCEC* are investigated. In the literature, fault prediction has been widely used to evaluate class cohesion metrics (Al Dallal and Briand, 2012; Gyimothy et al., 2005; Liu et al., 2009; Marcus et al., 2008).

The fault data in this case study are obtained from the PROMISE data repository[8] (Boetticher et al., 2007). Table 13 gives information about the Ant, Apache POI, Lucene and Xalan versions that are used in this case study. The third to fifth columns list the lines of code, the number of effective classes and the number of methods, respectively.

Table 14 shows statistics of the faulty programs' LCCs and their fault data. The second and third columns give the number of nodes and edges in the corresponding LCC. The fourth column lists the number of classes in $C_{\text{LCC}-\text{Cohesion}}$ ($t = 0.6$). The fifth column shows the number of **C**lasses that have records in PROMISE data repository to indicate whether they are **F**aulty (CF). The sixth column in Table 14 shows the number classes that appear in both the fourth and fifth columns ($C_{\text{LCC}-\text{Cohesion}} \bigcap \text{CF}$). The last column gives the number of **F**aulty **C**lasses (FC) in $C_{\text{LCC}-\text{Cohesion}} \bigcap \text{CF}$. Statistics in Table 14 mean that there are 1500 (468+295+208+529) classes evaluated in fault predictions, among them, there are 702 (119+208+146+229) classes contained at least one fault.

The evaluation process is similar to the one used in Liu et al. (2009). We apply the *univariate* and *multivariate logistic regression* (Hosmer and Lemeshow, 2004) to predict the fault proneness of a class using one or several combinations of its cohesion metrics. Firstly, the univariate logistic regression is used to investigate fault prediction ability of a single metric. Then the multivariate logistic regression is used to investigate all the possible combinations of metrics to investigate whether *MCC* and *MCEC* metrics can improve the fault detection results when they are combined with other metrics. There are 14 unique metrics (five previous class cohesion metrics, LOC, four versions of *MCC* and four versions of *MCEC*) in this case study. Thus, there are 91 combinations in the multivariate logistic regression. Ten top-performers are given in the following part of this section.

In logistic regression, the Nagelkerke R^2 (Nagelkerke, 1991) is often used to measure the *goodness of fit*. R^2 is in the interval [0, 1]. The bigger the value of R^2, the larger the portion of the variance in the dependent variable that is explained by the regression model.

Furthermore, to evaluate the prediction performance of logistic regression, the standard *Precision, Recall* and *F-measure* evaluation criteria (Olson and Delen, 2008) in information retrieval are used. Precision is defined as the number of classes correctly classified as faulty divided by the total number of classes classified as faulty. Recall is defined as the number of classes correctly classified as faulty divided by the actual number of classes that are faulty. Finally *F*-measure, defined as

$$F\text{-measure} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

takes both Precision and Recall into consideration. The shortcoming of *F*-measure criteria is that it requires a probability threshold to predict classes as faulty (Al Dallal and Briand, 2012), which is usually hard to decide. Thus, the *Receiver Operating Characteristic* (*ROC*) curves (Hanley and McNeil, 1982) is also used. In the context of fault prediction, the ROC curve is a graphical plot that is created by plotting the *True Positive Rate* (the ratio of classes correctly classified as faulty, equals to *Recall*) versus the *False Positive Rate* (the ratio of classes incorrectly classified as faulty) at different thresholds. The **A**rea **U**nder the ROC **C**urve (AUC) shows the fault prediction ability, and 100% AUC shows an ideal prediction result. For instance, Fig. 9 shows the ROC Curves of LSCC and *MCC* on Apache POI 3.0, and Coh and *MCC* on Xalan 2.6.0.

Results of the univariate logistic regression are given in Table 15. For each program, the second column gives the R^2 values and the regression results are sorted by these values. The third to fifth columns show the values of Precision, Recall and *F*-measure when the fault prediction threshold is 0.5. The sixth column gives AUC results, followed by its ranking in the seventh column. Then the parameters of logistic regression (C_0 and C_1) are listed. The *p*-values are given in the last column. Based on Table 15, it can be observed that there is no fixed order of these metrics for different programs, in other words, we cannot tell which metric performs better than others. Thus, when these metrics are used in software fault prediction, their statistical significance should all be examined firstly and then the proper metrics should be selected for specific programs. We can also make a conclusion that the four versions of *MCC* and the four versions of *MCEC* perform **equally** comparing with existing metrics.

Results of the multivariate logistic regression are given in Table 16. As previously mentioned, 10 combinations of metrics with largest R^2 values are listed for each program. Columns are similar to those in Table 16, followed by another parameter of logistic regression (C_2) shown in the last column. These results are quite promising. For each program, most of the top 10 combinations contain *MCC* and *MCEC* metrics. Concretely, there are 27 combinations out of 40 contain *MCC* and *MCEC* metrics. Moreover, three out of four top one combinations contain one of *MCC* or *MCEC* metrics.

It has to be noticed that the R^2 values in Tables 15 and 16 are relatively low comparing with common application scenarios of logistic regression. Previous researches have also reported similar results (Gyimothy et al., 2005; Liu et al., 2009; Marcus et al., 2008). Such results are understandable, as class cohesion metrics only give measurement of software's quality from one aspect. Software faults are the consequences of many complicated and inter-played factors, e.g., the coding style, the software architecture and the programmer's perception on the programming language. Thus, it is not beyond our expectations that cohesion metrics obtain relatively low R^2 values in fault prediction experiments. On the other hand, the statistical significances in these experiments have shown that class cohesion metrics

[8] http://openscience.us/repo/

Table 14
Statistics of faulty programs' LCCs.

Programs	N_{LCC}	E_{LCC}	$\lvert C_{\text{LCC}-\text{Cohesion}} \rvert$	$\lvert \text{CF} \rvert$	$\lvert C_{\text{LCC}-\text{Cohesion}} \bigcap \text{CF} \rvert$	$\lvert \text{FC} \subseteq C_{\text{LCC}-\text{Cohesion}} \bigcap \text{CF} \rvert$
Ant	6299	14,234	590	745	468	119
Apache POI	4369	7536	310	442	295	208
Lucene	2667	5124	277	340	208	146
Xalan	6573	13,819	622	885	529	229

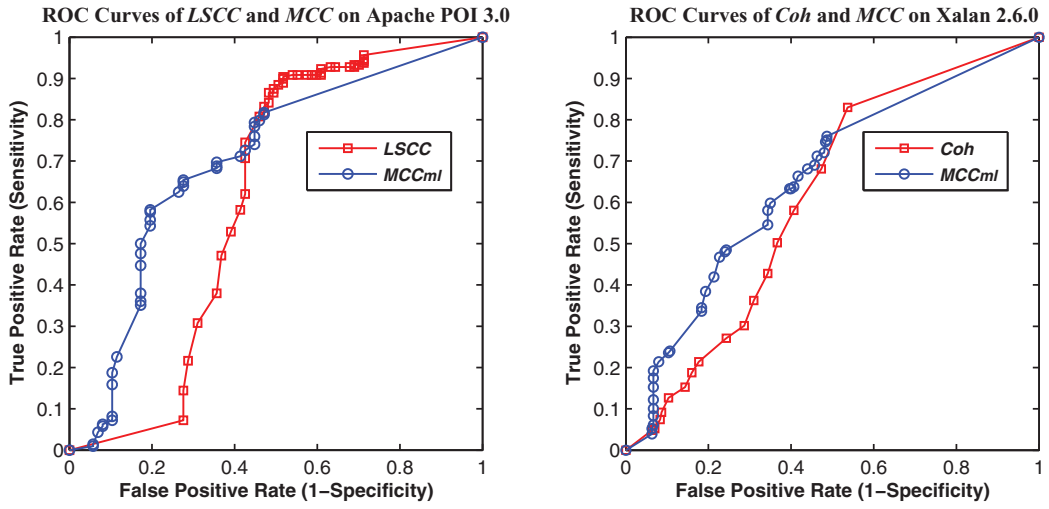

Fig. 9. ROC Curves of *LSCC* and *MCC* on Apache POI 3.0, Coh and *MCC* on Xalan 2.6.0.

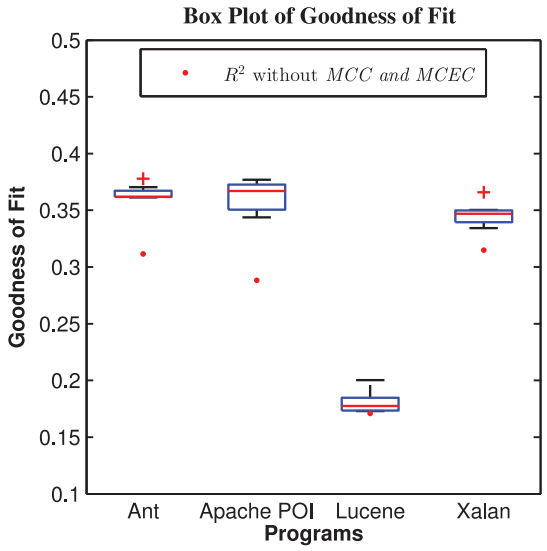

Fig. 10. Box plot shows the improvement introduced by the proposed metrics.

do influence software quality and classes with low cohesion metrics are indeed more prone to be faulty. Therefore, class cohesion metrics are still helpful indicators for software quality. For instance, some *MCC* and *MCEC* metrics in Table 15 already have obtained AUC greater than 0.7, which indicates an acceptable predictor for software faults. In fault prediction practices, class cohesion metrics are usually used together with many other software metrics to obtain more practical and actionable results (Gyimothy et al., 2005; Scanniello et al., 2013).

Experiments using all the 14 metrics in the multivariate logistic regression were also conducted. Table 17 shows the results of these experiments. For each program, the R^2 values and AUC are shown. The metrics whose *p*-values are less than 0.05 in the regression model are shown in the last column. It can be observed that the proposed metrics appear for three out of four programs.

Based on these results, we can draw a conclusion that *MCC* and *MCEC* metrics perform **better** than existing class cohesion metrics in the multivariate logistic regression.

Fig. 10 shows the comparisons between fault prediction results with and without *MCC* and *MCEC*. For each program, the previous metrics whose *p*-values in the univariate logistic regression (Table 15) less than 0.05 are selected, then these metrics are used in the multivariate logistic regression. Then each *MCC* and *MCEC* metric is included in the multivariate logistic regression **separately**. The red dots in Fig. 10 show the R^2 values without *MCC* and *MCEC*, and the box plots show the distributions of R^2 values after *MCC* and *MCEC* metrics are included in the model separately. It can be observed that after the inclusion of each single *MCC* and *MCEC* metric, the R^2 value always increases. Again, this result proves that every single *MCC* and *MCEC* metric can improve the performance of fault prediction. This experiment also answers the question in Section 5.2 that the proposed metrics do provide additional and useful measurement dimension of class cohesion.

Based on these experimental results, we can observe that *MCC* and *MCEC* usually perform equally comparing with existing metrics when they are used alone in the univariate logistic regression, and they usually provide additional and useful information for predicting faults when they are combined with other metrics in the multivariate logistic regression, in which they usually perform better than existing metrics.

6. Discussion

6.1. The (in)stability of community detection algorithms

Community detection algorithms may not obtain exactly the same results in different runs. To study this effect we run each community detection algorithm 100 times on jEdit. Results of these experiments are shown in Fig. 11. The number of detected communities are shown in the first subfigure, followed by the *Q* values in the second subfigure. It can be noticed that the *fg* and *ml* algorithms are very stable as both obtain exactly same results in all the experiments. On the other hand, the *lp* algorithm exhibits relative drastic changes among different runs, which makes the cohesion measurement unstable. In the computing process of the proposed metrics, one can use average values of multiple runs to alleviate this problem. For instance, a programmer can run the community detection process for 100 times and then calculate 100 values of a certain class's *MCC*, then she can average these 100 values to obtain an average value of *MCC* of this class.

6.2. Applications in refactoring

We believe it is possible to apply the proposed metrics to refactoring practices. Here we give a simple example on how to use the

Table 15
Fault prediction results using the univariate logistic regression on four programs (sorted by R^2).

Ant 1.7.0

Metrics	R^2	Precision	Recall	*F*-measure	AUC	AUC rank	C_0	C_1	*p*-value
LOC	0.321	0.529	0.605	0.565	0.814	1	-1.219	0.005	< 0.0001
LCOM1	0.304	0.561	0.538	0.549	0.784	2	-0.759	0.008	< 0.0001
LCOM2	0.268	0.541	0.496	0.518	0.767	3	-0.644	0.008	< 0.0001
LSCC	0.098	0.302	0.815	0.441	0.585	9	0.397	-1.802	< 0.0001
LCOM5	0.076	0.316	0.824	0.457	0.586	8	-1.230	1.593	0.00049
$MCEC_{lp}$	0.075	0.302	0.765	0.433	0.550	12	0.475	-1.352	0.00034
MCC_{im}	0.063	0.364	0.639	0.463	0.633	6	0.872	-1.530	0.00096
MCC_{lp}	0.061	0.362	0.647	0.464	0.642	4	0.948	-1.541	0.0012
MCC_{fg}	0.052	0.351	0.563	0.432	0.633	5	1.254	-1.646	0.0026
$MCEC_{im}$	0.045	0.298	0.765	0.429	0.546	13	0.329	-1.061	0.0052
MCC_{ml}	0.038	0.322	0.546	0.405	0.610	7	1.018	-1.344	0.0103
Coh	0.034	0.299	0.748	0.427	0.502	14	0.306	-1.389	0.0167
$MCEC_{ml}$	0.033	0.318	0.630	0.423	0.552	11	0.409	-0.775	0.0160
$MCEC_{fg}$	0.027	0.341	0.639	0.444	0.578	10	0.373	-0.705	0.0275

Apache POI 3.0

Metrics	R^2	Precision	Recall	*F*-measure	AUC	AUC rank	C_0	C_1	*p*-value
LSCC	0.198	0.807	0.865	0.835	0.608	10	0.674	-2.647	< 0.0001
LOC	0.158	0.837	0.519	0.641	0.764	3	-0.760	0.005	0.00014
$MCEC_{im}$	0.138	0.766	0.788	0.777	0.576	11	0.617	-2.245	0.00011
$MCEC_{fg}$	0.134	0.804	0.712	0.755	0.631	9	0.837	-1.719	< 0.0001
LCOM1	0.133	0.918	0.433	0.588	0.813	1	-0.379	0.006	0.0057
$MCEC_{ml}$	0.130	0.800	0.750	0.774	0.641	8	0.800	-1.668	< 0.0001
MCC_{fg}	0.113	0.798	0.625	0.701	0.697	5	1.927	-2.568	0.00023
MCC_{im}	0.103	0.831	0.731	0.777	0.677	6	1.322	-2.291	0.00042
MCC_{ml}	0.102	0.850	0.654	0.739	0.701	4	1.690	-2.296	0.00042
MCC_{lp}	0.086	0.797	0.716	0.754	0.660	7	1.214	-2.102	0.0011
LCOM2	0.079	0.905	0.365	0.521	0.770	2	-0.231	0.004	0.0265
$MCEC_{lp}$	0.078	0.748	0.798	0.772	0.529	14	0.441	-1.549	0.0022
LCOM5	0.015	0.766	0.740	0.753	0.534	13	-0.421	0.564	0.1609
Coh	0.00030	0.719	0.639	0.677	0.556	12	0.029	-0.112	0.8437

Lucene 2.4.0

Metrics	R^2	Precision	Recall	*F*-measure	AUC	AUC rank	C_0	C_1	*p*-value
LCOM1	0.134	0.853	0.397	0.542	0.706	2	-0.393	0.013	0.0071
LOC	0.133	0.871	0.555	0.678	0.716	1	-0.658	0.005	0.0030
LCOM2	0.133	0.852	0.356	0.502	0.655	3	-0.304	0.023	0.0156
$MCEC_{ml}$	0.062	0.789	0.664	0.721	0.591	5	0.550	-1.040	0.0162
$MCEC_{lp}$	0.025	0.740	0.760	0.750	0.520	11	0.240	-0.676	0.1307
LCOM5	0.018	0.730	0.630	0.676	0.564	8	-0.380	0.644	0.1942
MCC_{ml}	0.013	0.813	0.507	0.624	0.615	4	0.496	-0.664	0.2780
LSCC	0.012	0.745	0.699	0.721	0.576	6	0.185	-0.571	0.2869
$MCEC_{fg}$	0.012	0.758	0.623	0.684	0.547	9	0.248	-0.452	0.2906
$MCEC_{im}$	0.009	0.731	0.801	0.765	0.481	14	0.124	-0.440	0.3693
Coh	0.002	0.745	0.562	0.641	0.516	12	0.084	-0.252	0.6902
MCC_{lp}	0.001	0.732	0.486	0.584	0.538	10	0.117	-0.199	0.7089
MCC_{im}	0.001	0.689	0.486	0.570	0.510	13	-0.082	0.157	0.7730
MCC_{fg}	0.00022	0.652	0.500	0.566	0.571	7	-0.067	0.088	0.8872

Xalan 2.6.0

Metrics	R^2	Precision	Recall	F-measure	AUC	AUC rank	C_0	C_1	p-value
LOC	0.264	0.717	0.541	0.617	0.791	1	-0.906	0.004	< 0.0001
$MCEC_{im}$	0.134	0.528	0.852	0.652	0.584	10	0.539	-1.724	< 0.0001
LCOM1	0.102	0.795	0.288	0.423	0.705	2	-0.223	0.002	0.00019
LCOM2	0.085	0.826	0.249	0.383	0.637	4	-0.177	0.002	0.00064
MCC_{im}	0.081	0.543	0.668	0.599	0.629	5	0.745	-1.443	< 0.0001
$MCEC_{ml}$	0.077	0.526	0.668	0.588	0.619	7	0.587	-1.188	< 0.0001
MCC_{ml}	0.070	0.566	0.598	0.582	0.655	3	1.246	-1.708	< 0.0001
$MCEC_{fg}$	0.064	0.519	0.594	0.554	0.602	9	0.618	-1.092	< 0.0001
LSCC	0.063	0.488	0.721	0.582	0.548	12	0.368	-1.200	< 0.0001
MCC_{fg}	0.039	0.516	0.493	0.504	0.624	6	1.001	-1.288	0.00036
$MCEC_{lp}$	0.038	0.487	0.738	0.587	0.523	13	0.337	-0.844	0.00031
MCC_{lp}	0.013	0.506	0.555	0.529	0.571	11	0.373	-0.602	0.0345
Coh	0.013	0.488	0.428	0.456	0.614	8	-0.171	0.650	0.0374
LCOM5	0.002	0.462	0.638	0.536	0.481	14	-0.108	0.173	0.4647

metrics to guide the refactoring of the *Move Method* (Fowler et al., 1999). There is one threshold in this process, denoted as k, which is a constant value between 0 and 1.

Assume there are m_1 methods in class C_1 located in LCC. Let $MCC(C_1) = \frac{n_{1\max}}{m_1}$. That is, there are $n_{1\max}$ methods in C_1 that are located in a community Com_1, and $m_1 - n_{1\max}$ methods in C_1 that are located in other communities. If the following conditions hold:

1. $MCC(C_1) > k$.
2. There exist one method $Method_1$ in C_1, another community Com_2 and another class C_2, satisfies that: (1) $Method_1 \in Com_2$; (2)

Table 16
Fault prediction results using the multivariate logistic regression on four programs, top 10 combinations out of 91 are shown (sorted by R^2).

Ant 1.7.0									
Metrics	R^2	Precision	Recall	*F*-measure	AUC	AUC rank	C_0	C_1	C_2
LOC+$MCEC_{lp}$	0.340	0.529	0.613	0.568	0.809	6	-0.846	0.004	-0.900
LCOM1+LOC	0.338	0.570	0.613	0.591	0.813	2	-1.106	0.004	0.003
LCOM1+$MCEC_{lp}$	0.334	0.560	0.588	0.574	0.780	18	-0.365	0.008	-1.069
LOC+MCC_{lp}	0.332	0.556	0.622	0.587	0.801	13	-0.655	0.005	-0.846
LCOM2+LOC	0.331	0.557	0.613	0.584	0.813	3	-1.139	0.003	0.004
LCOM1+LCOM2	0.328	0.576	0.571	0.574	0.783	15	-0.855	0.031	-0.025
LOC+MCC_{im}	0.325	0.548	0.622	0.583	0.806	9	-0.863	0.005	-0.542
LCOM5+LOC	0.325	0.532	0.622	0.574	0.802	12	-1.524	0.476	0.005
LSCC+LOC	0.324	0.522	0.605	0.560	0.804	11	-1.050	-0.438	0.004
LOC+MCC_{fg}	0.324	0.541	0.613	0.575	0.808	8	-0.787	0.005	-0.515
Apache POI 3.0									
Metrics	R^2	Precision	Recall	*F*-measure	AUC	AUC rank	C_0	C_1	C_2
LSCC+Coh	0.301	0.801	0.870	0.834	0.742	27	0.022	-7.153	5.748
LOC+$MCEC_{fg}$	0.270	0.833	0.793	0.813	0.761	15	0.061	0.005	-1.800
LSCC+LOC	0.264	0.777	0.736	0.756	0.750	22	-0.006	-2.211	0.003
LOC+$MCEC_{ml}$	0.263	0.840	0.808	0.824	0.780	5	0.017	0.005	-1.703
LOC+MCC_{fg}	0.258	0.855	0.736	0.791	0.756	17	1.156	0.005	-2.649
LSCC+$MCEC_{ml}$	0.251	0.852	0.774	0.811	0.698	48	1.181	-2.333	-1.254
LSCC+LCOM1	0.249	0.810	0.817	0.813	0.732	31	0.322	-2.233	0.003
LCOM1+$MCEC_{im}$	0.245	0.818	0.865	0.841	0.767	12	0.250	0.005	-2.312
LCOM1+$MCEC_{fg}$	0.243	0.821	0.769	0.794	0.773	8	0.434	0.005	-1.693
LSCC+$MCEC_{fg}$	0.241	0.841	0.764	0.801	0.687	54	1.125	-2.250	-1.169
Lucene 2.4.0									
Metrics	R^2	Precision	Recall	*F*-measure	AUC	AUC rank	C_0	C_1	C_2
LCOM2+$MCEC_{ml}$	0.176	0.808	0.719	0.761	0.720	7	0.208	0.022	-0.945
LCOM2+LOC	0.171	0.875	0.479	0.619	0.731	1	-0.633	0.016	0.003
LCOM1+$MCEC_{ml}$	0.167	0.815	0.692	0.748	0.729	3	0.075	0.012	-0.822
LOC+$MCEC_{ml}$	0.157	0.861	0.637	0.732	0.729	2	-0.208	0.004	-0.712
LCOM1+LCOM2	0.150	0.841	0.363	0.507	0.709	16	-0.397	0.007	0.015
LOC+MCC_{im}	0.149	0.833	0.548	0.661	0.711	14	-1.126	0.005	0.756
LCOM2+$MCEC_{lp}$	0.147	0.847	0.418	0.560	0.669	29	-0.090	0.022	-0.571
LCOM1+MCC_{im}	0.147	0.845	0.411	0.553	0.660	30	-0.791	0.014	0.687
LCOM1+LOC	0.146	0.867	0.493	0.629	0.725	4	-0.581	0.007	0.003
LCOM1+$MCEC_{lp}$	0.144	0.865	0.438	0.582	0.710	15	-0.205	0.013	-0.477
Xalan 2.6.0									
Metrics	R^2	Precision	Recall	*F*-measure	AUC	AUC rank	C_0	C_1	C_2
LOC+$MCEC_{im}$	0.345	0.681	0.690	0.685	0.804	1	-0.398	0.004	-1.658
LOC+MCC_{im}	0.314	0.665	0.642	0.653	0.776	10	-0.222	0.004	-1.359
LOC+MCC_{ml}	0.308	0.674	0.642	0.658	0.776	9	0.242	0.004	-1.581
Coh+LOC	0.305	0.663	0.694	0.678	0.784	4	-1.384	1.404	0.004
LOC+$MCEC_{ml}$	0.304	0.708	0.646	0.676	0.779	6	-0.377	0.003	-1.025
LOC+$MCEC_{fg}$	0.303	0.678	0.633	0.655	0.774	11	-0.329	0.004	-1.020
LOC+MCC_{fg}	0.295	0.653	0.624	0.638	0.770	13	0.113	0.004	-1.340
LCOM5+LOC	0.280	0.696	0.550	0.615	0.780	5	-0.567	-0.722	0.004
LOC+$MCEC_{lp}$	0.276	0.727	0.594	0.654	0.774	12	-0.653	0.004	-0.575
LOC+MCC_{lp}	0.269	0.734	0.555	0.632	0.777	8	-0.632	0.004	-0.439

there are $n_{2\max}$ methods in C_2 distribute in Com_2; (3) $MCC(C_2) = \frac{n_{2\max}}{m_2} > k$.

Then $Method_1$ should be moved from C_1 to C_2.

This process is similar to a "Thought Experiment" in Physics that is lack of experimental and data support. But we hope this discussion will inspire the research community to develop more practical algorithms using the metrics proposed in this paper in refactoring practices.

Table 17
Fault prediction results using the multivariate logistic regression with all the 14 metrics.

Programs	R^2	AUC	Metrics (p-value < 0.05)
Ant	0.407	0.800	$MCEC_{lp}$
Apache POI	0.508	0.834	Coh, LCOM1, LCOM2, LSCC, $MCEC_{lp}$
Lucene	0.303	0.770	LCOM2
Xalan	0.408	0.826	Coh, LOC, $MCEC_{im}$, MCC_{ml}

6.3. Toward a unified measurement and theoretical framework

The metrics and its computation methodology discussed in this paper might be combined with other metrics or networks in previous research. Two potential directions exist:

The first one is to combine with previous class cohesion metrics. As discussed in Section 2, many class cohesion metrics (LCOM1, LSCC, etc.) measure relationships among methods of a class considering whether these methods share same attributes. Such attribute sharing quantification results can be treated as edge weights in Call Graphs. The shortcoming of this approach is that such weights can only be added if two nodes belong to the same class. How can we add weights between two nodes that representing methods from different classes? This question is left open for future research.

The second one is to combine Call Graph with networks at other granularities. For instance, Class Dependency Network is such network reflecting Class relations. It might be possible to use the similar methodology to quantify a software package's cohesiveness by

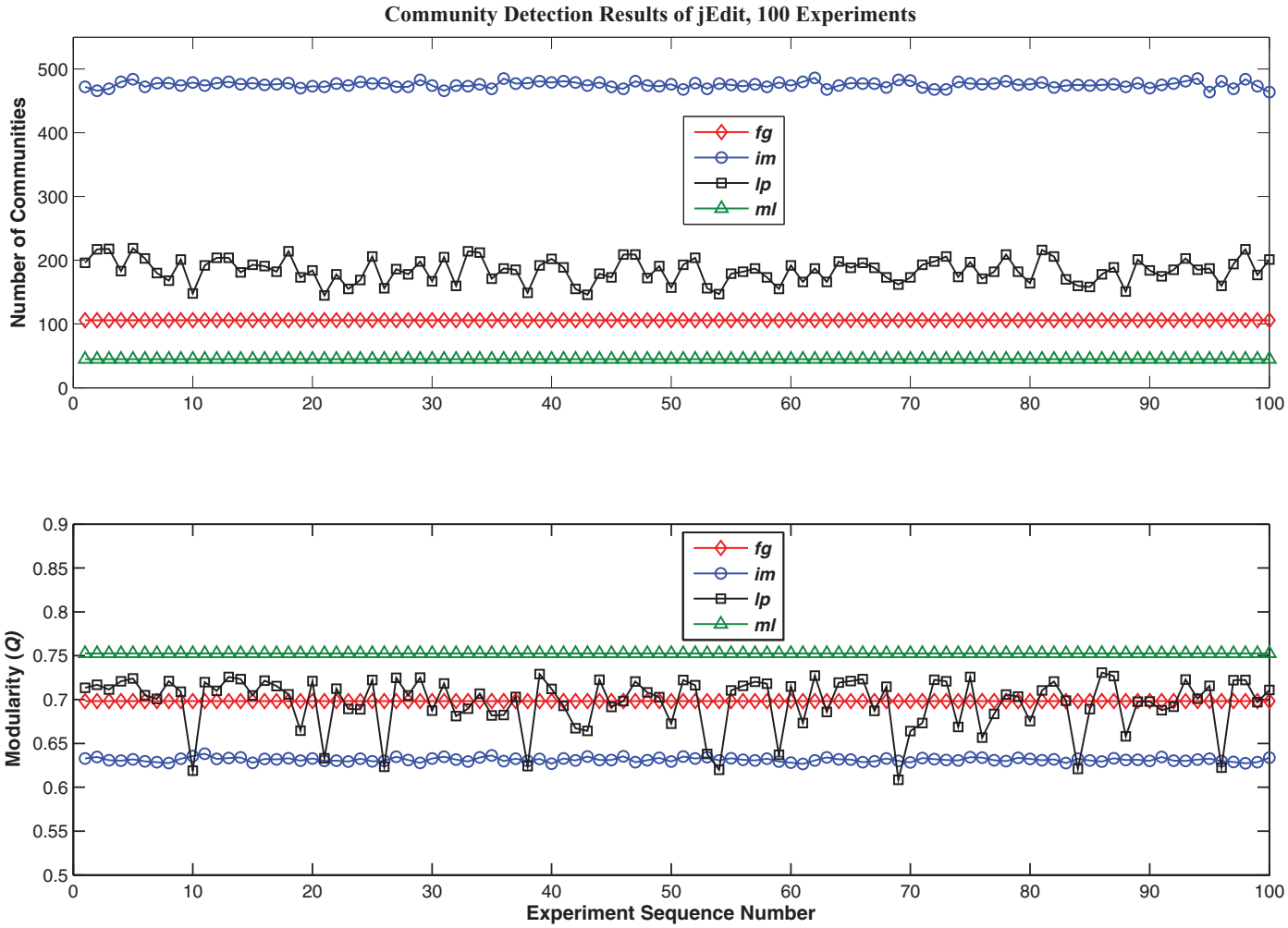

Fig. 11. Community detection results on jEdit, which can show the stability of different algorithms.

detecting community structures in Class Dependency Network, as classes belong to the same package are supposed to reside in the same community. It is more challenging to combine these networks reflecting software entity relations at different levels. The newly developed theory of *Multilayer Networks* (Boccaletti et al., 2014) might be a promising theoretical basis for this direction.

7. Conclusions

In this paper, by using four community detection algorithms in the analysis of 10 widely-used open-source Java software systems, we have shown that software static Call Graphs usually present relatively significant community structures. Two class cohesion metrics have been proposed. The two metrics are based on the distributions of a class's methods among communities, thus can reflect the class's cohesiveness. We show that the proposed metrics can provide additional and useful information of class cohesion that is not reflected by existing class cohesion metrics. In fault prediction experiments on four open-source programs containing 1500 classes, when the proposed metrics are used alone, they usually perform equally comparing with existing metrics. When combinations of metrics are evaluated, the proposed metrics usually provide better results than existing metrics. In the future, we plan to investigate how to use the community structure and the proposed metrics to guide software refactoring practices.

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (91118005, 91218301, 91418205, 61221063, 61203174, 61428206 and U1301254), Doctoral Fund of Ministry of Education of China (20110201120010), 863 High Tech Development Plan of China (2012AA011003), 111 International Collaboration Program of China, and the Fundamental Research Funds for the Central Universities. We would also like to thank the anonymous reviewers for their insightful comments and valuable suggestions for improving this paper.

References

Al Dallal, J., 2010. Mathematical validation of object-oriented class cohesion metrics. Int. J. Comput. 4 (2), 45–52.
Al Dallal, J., 2012. The impact of accounting for special methods in the measurement of object-oriented class cohesion on refactoring and fault prediction activities. J. Syst. Softw. 85 (5), 1042–1057.
Al Dallal, J., 2013. Qualitative analysis for the impact of accounting for special methods in object-oriented class cohesion measurement. J. Softw. 8 (2), 327–336.
Al Dallal, J., Briand, L.C., 2012. A precise method-method interaction-based cohesion metric for object-oriented classes. ACM Trans. Softw. Eng. Methodol. (TOSEM) 21 (2), 8.
Badri, L., Badri, M., 2004. A proposal of a new class cohesion criterion: an empirical study. J. Object Technol. 3 (4), 145–159.
Barabási, A.-L., Albert, R., 1999. Emergence of scaling in random networks. Science 286 (5439), 509–512.

Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., et al., 2006. Understanding the shape of java software. In: ACM SIGPLAN Notices, vol. 41. ACM, pp. 397–412.
Bhattacharya, P., Iliofotou, M., Neamtiu, I., Faloutsos, M., 2012. Graph-based analysis and prediction for software evolution. In: Proceedings of the 2012 International Conference on Software Engineering. IEEE Press, pp. 419–429.
Bieman, J.M., Kang, B.-K., 1995. Cohesion and reuse in an object-oriented system. In: ACM SIGSOFT Software Engineering Notes, vol. 20. ACM, pp. 259–262.
Blondel, V.D., Guillaume, J.-L., Lambiotte, R., Lefebvre, E., 2008. Fast unfolding of communities in large networks. J. Stat. Mech.: Theor. Exper. 2008 (10), P10008.
Boccaletti, S., Bianconi, G., Criado, R., Del Genio, C., Gómez-Gardeñes, J., Romance, M., 2014. The structure and dynamics of multilayer networks. Phys. Rep. 544 (1), 1–122.
Boetticher, G., Menzies, T., Ostrand, T., 2007. Promise Repository of Empirical Software Engineering Data. Department of Computer Science, West Virginia University.
Briand, L.C., Bunse, C., Daly, J.W., 2001. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. IEEE Trans. Softw. Eng. 27 (6), 513–530.
Briand, L.C., Daly, J.W., Wüst, J., 1998. A unified framework for cohesion measurement in object-oriented systems. Empir. Softw. Eng. 3 (1), 65–117.
Cai, K.-Y., Yin, B.-B., 2009. Software execution processes as an evolving complex network. Inform. Sci. 179 (12), 1903–1928.
Chakrabarti, D., Faloutsos, C., 2006. Graph mining: Laws, generators, and algorithms. ACM Comput. Survey (CSUR) 38 (1), 2.
Chen, Z., Zhou, Y., Xu, B., Zhao, J., Yang, H., 2002. A novel approach to measuring class cohesion based on dependence analysis. In: Proceedings of the International Conference on Software Maintenance, 2002. IEEE, pp. 377–384.
Chidamber, S.R., Kemerer, C.F., 1991. Towards a metrics suite for object oriented design. In: Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications, vol. 26. ACM, Phoenix, Arizona, USA, pp. 197–211.
Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493.
Clauset, A., Newman, M.E., Moore, C., 2004. Finding community structure in very large networks. Phys. Rev. E 70 (6), 066111.
Concas, G., Monni, C., Orru, M., Tonelli, R., 2013. A study of the community structure of a complex software network. In: 2013 4th International Workshop on Emerging Trends in Software Metrics (WETSoM). IEEE, pp. 14–20.
Danon, L., Diaz-Guilera, A., Duch, J., Arenas, A., 2005. Comparing community structure identification. J. Stat. Mech.: Theor. Exper. 2005 (09), P09008.
Dill, S., Kumar, R., McCurley, K.S., Rajagopalan, S., Sivakumar, D., Tomkins, A., 2002. Self-similarity in the web. ACM Trans. Internet Technol. (TOIT) 2 (3), 205–223.
Dunn, R., Dudbridge, F., Sanderson, C.M., 2005. The use of edge-betweenness clustering to investigate biological function in protein interaction networks. BMC Bioinform. 6 (1), 39.
Flake, G.W., Lawrence, S., Giles, C.L., 2000. Efficient identification of web communities. In: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp. 150–160.
Fortunato, S., 2010. Community detection in graphs. Phys. Rep. 486 (3), 75–174.
Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. Refactoring: Improving the Design of Existing Code. Addison Wesley.
Girvan, M., Newman, M.E., 2002. Community structure in social and biological networks. Proc. Natl. Acad. Sci. 99 (12), 7821–7826.
Good, B.H., de Montjoye, Y.-A., Clauset, A., 2010. Performance of modularity maximization in practical contexts. Phys. Rev. E 81 (4), 046106.
Graham, S.L., Kessler, P.B., Mckusick, M.K., 1982. Gprof: A call graph execution profiler. In: ACM Sigplan Notices, vol. 17. ACM, pp. 120–126.
Guimera, R., Sales-Pardo, M., Amaral, L.A.N., 2004. Modularity from fluctuations in random graphs and complex networks. Phys. Rev. E 70 (2), 025101.
Gyimothy, T., Ferenc, R., Siket, I., 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans. Softw. Eng. 31 (10), 897–910.
Hanley, J.A., McNeil, B.J., 1982. The meaning and use of the area under a receiver operating characteristic (roc) curve.. Radiology 143 (1), 29–36.
Hosmer Jr, D.W., Lemeshow, S., 2004. Applied Logistic Regression. John Wiley & Sons.
Hu, Y., Nie, Y., Yang, H., Cheng, J., Fan, Y., Di, Z., 2010. Measuring the significance of community structure in complex networks. Phys. Rev. E 82 (6), 066106.
Jin, R.K.-X., Parkes, D.C., Wolfe, P.J., 2007. Analysis of bidding networks in ebay: Aggregate preference identification through community detection. In: Proceedings of AAAI Workshop on Plan, Activity and Intent Recognition (PAIR). AAAI, pp. 66–73.
Jonsson, P.F., Cavanna, T., Zicha, D., Bates, P.A., 2006. Cluster analysis of networks generated through homology: Automatic identification of important protein communities involved in cancer metastasis. BMC Bioinform. 7 (1), 2.
Karrer, B., Levina, E., Newman, M., 2008. Robustness of community structure in networks. Phys. Rev. E 77 (4), 046119.
Lancichinetti, A., Radicchi, F., Ramasco, J.J., 2010. Statistical significance of communities in networks. Phys. Rev. E 81 (4), 046110.
Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W., 2008. Statistical properties of community structure in large social and information networks. In: Proceedings of the 17th International Conference on World Wide Web. ACM, pp. 695–704.
Li, H., Zhao, H., Cai, W., Xu, J.-Q., Ai, J., 2013. A modular attachment mechanism for software network evolution. Phys. A: Stat. Mech. Appl. 392 (9), 2025–2037.
Liu, Y., Poshyvanyk, D., Ferenc, R., Gyimóthy, T., Chrisochoides, N., 2009. Modeling class cohesion as mixtures of latent topics. In: IEEE International Conference on Software Maintenance, 2009 (ICSM'09). IEEE, pp. 233–242.
Louridas, P., Spinellis, D., Vlachos, V., 2008. Power laws in software. ACM Trans. Softw. Eng. Methodol. (TOSEM) 18 (1), 2.
Marcus, A., Poshyvanyk, D., Ferenc, R., 2008. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Trans. Softw. Eng. 34 (2), 287–300.
Mucha, P.J., Richardson, T., Macon, K., Porter, M.A., Onnela, J.-P., 2010. Community structure in time-dependent, multiscale, and multiplex networks. Science 328 (5980), 876–878.
Myers, C.R., 2003. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. Phys. Rev. E 68 (4), 046116.
Nagelkerke, N.J., 1991. A note on a general definition of the coefficient of determination. Biometrika 78 (3), 691–692.
Newman, M.E., Girvan, M., 2004. Finding and evaluating community structure in networks. Phys. Rev. E 69 (2), 026113.
Olson, D.L., Delen, D., 2008. Advanced Data Mining Techniques. Springer.
Palla, G., Derényi, I., Farkas, I., Vicsek, T., 2005. Uncovering the overlapping community structure of complex networks in nature and society. Nature 435 (7043), 814–818.
Pan, W., Li, B., Ma, Y., Liu, J., 2011. Multi-granularity evolution analysis of software using complex network theory. J. Syst. Sci. Complex. 24 (6), 1068–1082.
Pearson, K., 1901. Liii. on lines and planes of closest fit to systems of points in space. London Edinburgh Dublin Philos. Mag. J. Sci. 2 (11), 559–572.
Potanin, A., Noble, J., Frean, M., Biddle, R., 2005. Scale-free geometry in oo programs. Commun. ACM 48 (5), 99–103.
Pressman, R.S., 2010. Software Engineering: A Practitioner's Approach. McGraw-Hill.
Qu, Y., Guan, X., Zheng, Q., Liu, T., Zhou, J., Li, J., 2015. Calling network: A new method for modeling software runtime behaviors. ACM SIGSOFT Softw. Eng. Note 40 (1), 1–8. doi:10.1145/2693208.2693223.
Raghavan, U.N., Albert, R., Kumara, S., 2007. Near linear time algorithm to detect community structures in large-scale networks. Phys. Rev. E 76 (3), 036106.
Reichardt, J., Bornholdt, S., 2007. Clustering of sparse data via network communities—a prototype study of a large online market. J. Stat. Mech.: Theor. Exper. 2007 (06), P06016.
Rosvall, M., Bergstrom, C.T., 2008. Maps of random walks on complex networks reveal community structure. Proc. Natl. Acad. Sci. 105 (4), 1118–1123.
Scanniello, G., Gravino, C., Marcus, A., Menzies, T., 2013. Class level fault prediction using software clustering. In: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE). IEEE, pp. 640–645.
Sellers, B.H., 1996. Object-Oriented Metrics: Measures of Complexity. Prentice Hall.
Shannon, C.E., 2001. A mathematical theory of communication. ACM SIGMOBILE Mobile Comput. Commun. Rev. 5 (1), 3–55.
Spearman, C., 1904. The proof and measurement of association between two things. Amer. J. Psychol. 15 (1), 72–101.
Šubelj, L., Bajec, M., 2011. Community structure of complex software systems: Analysis and applications. Phys. A: Stat. Mech. Appl. 390 (16), 2968–2975.
Šubelj, L., Bajec, M., 2012. Software systems through complex networks science: Review, analysis and applications. In: Proceedings of the First International Workshop on Software Mining. ACM, pp. 9–16.
Šubelj, L., Žitnik, S., Blagus, N., Bajec, M., 2014. Node mixing and group structure of complex software networks. Adv. Comp. Syst. 17, 1450022.
Turnu, I., Concas, G., Marchesi, M., Pinna, S., Tonelli, R., 2011. A modified yule process to model the evolution of some object-oriented system properties. Inform. Sci. 181 (4), 883–902.
Valverde, S., Solé, R.V., 2007. Hierarchical small worlds in software architecture. Dynam. Cont. Discr. Impul. Syst.: Ser. B 14, 1–11.
Watts, D.J., Strogatz, S.H., 1998. Collective dynamics of 'small-world' networks. Nature 393 (6684), 440–442.
Zhou, Y., Lu, J., Xu, H.L.B., 2004. A comparative study of graph theory-based class cohesion measures. ACM SIGSOFT Softw. Eng. Note 29 (2), 1–6.

Yu Qu received the B.S. degree from the School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, China in 2006. He is currently a Ph.D. candidate student at the Ministry of Education Key Lab for Intelligent Networks and Network Security, Xi'an Jiaotong University. His research interests include trustworthy software and applying complex network and data mining theories to analyzing software systems.

Xiaohong Guan received the B.S. and M.S. degrees from Tsinghua University, Beijing, China in 1982 and 1985 respectively, and his Ph.D. degree from the University of Connecticut in 1993. He was with the Division of Engineering and Applied Science, Harvard University from 1999 to 2000. He is the Cheung Kong Professor of Systems Engineering and the Dean of School of Electronic and Information Engineering, Xi'an Jiaotong University. He is also the Director of the Center for Intelligent and Networked Systems, Tsinghua University, and served as the Head of Department of Automation, 2003–2008. His research interests include cyber-physical systems and network security.

Qinghua Zheng received the B.S. and M.S. degrees in computer science and technology from Xi'an Jiaotong University, Xi'an, China in 1990 and 1993, respectively, and his Ph.D. degree in systems engineering from the same university in 1997. He was a postdoctoral researcher at Harvard University in 2002. Since 1995 he has been with the Department of Computer Science and Technology at Xi'an Jiaotong University, and was appointed director of the Department in 2008 and Cheung Kong Professor in 2009. His research interests include intelligent e-learning and trustworthy software.

Ting Liu received the B.S. and Ph.D. degrees from Xi'an Jiaotong University, Xi'an, China in 2003 and 2010 respectively. He is an associate professor in systems engineering at

Xi'an Jiaotong University. His research interests include cyber-physical systems, network security and trustworthy software.

Lidan Wang received the B.S. degree from the School of Software Engineering, Xidian University, Xi'an, China in 2013. She is currently an M.S. candidate student at the Ministry of Education Key Lab for Intelligent Networks and Network Security, Xi'an Jiaotong University. Her research interests include trustworthy software and software engineering.

Yuqiao Hou received the B.S. degree from the School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, China in 2012. She is currently an M.S. candidate student at the Ministry of Education Key Lab for Intelligent Networks and Network Security, Xi'an Jiaotong University. Her research interests include trustworthy software and software engineering.

Zijiang Yang is an associate professor in computer science at Western Michigan University. He holds a Ph.D. degree from the University of Pennsylvania, an M.S. degree from Rice University and a B.S. degree from the University of Science and Technology of China. Before joining WMU he was an associate research staff member at NEC Labs America. He was also a visiting professor at the University of Michigan from 2009 to 2013. His research interests are in the area of software engineering with the primary focus on the testing, debugging and verification of software systems. He is a senior member of IEEE.