2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

From Developer Networks to Verified Communities: A Fine-Grained Approach

Mitchell Joblin
Siemens AG
Erlangen, Germany

Wolfgang Mauerer
Siemens AG
OTH Regensburg
Munich/Regensburg, Germany

Sven Apel, Janet Siegmund
University of Passau
Passau, Germany

Dirk Riehle
Friedrich-Alexander-University
Erlangen-Nürnberg
Erlangen, Germany

***Abstract*—Effective software engineering demands a coordinated effort. Unfortunately, a comprehensive view on developer coordination is rarely available to support software-engineering decisions, despite the significant implications on software quality, software architecture, and developer productivity. We present a fine-grained, verifiable, and fully automated approach to capture a view on developer coordination, based on commit information and source-code structure, mined from version-control systems. We apply methodology from network analysis and machine learning to identify developer communities automatically. Compared to previous work, our approach is fine-grained, and identifies statistically significant communities using order-statistics and a community-verification technique based on graph conductance. To demonstrate the scalability and generality of our approach, we analyze ten open-source projects with complex and active histories, written in various programming languages. By surveying 53 open-source developers from the ten projects, we validate the authenticity of inferred community structure with respect to reality. Our results indicate that developers of open-source projects form statistically significant community structures and this particular view on collaboration largely coincides with developers' perceptions of real-world collaboration.**

I. Introduction

Software engineering is fundamentally the coordinated effort of individuals to construct a software system. Typically, the complexity of a software system is managed by a divide-and-conquer strategy, in which the system whole is decomposed into simpler sub-components [12]. Software developers are required to coordinate their efforts to manage sub-component dependencies and to reconcile the software's sub-components into a functional whole. Failure in the software to meet expectations is often the consequence of insufficient coordination between developers [7], [16], [9], [12].

Recent empirical research has demonstrated the significant influence of developer organization on software quality [7], [31], [27], [28]. It even suggests that evaluation methods that are entirely based on organizational properties are more indicative of fault proneness than traditional source-code-centric metrics [28].

Recently, version-control systems (VCS) have been used to construct developer networks that capture the organizational structure [23], [24], [17], [18], [26]. Thus far, the primary focus has been on characterizing the global network properties that govern all developer networks, such as the small-world property.[1] The predominant method to construct developer networks assumes that all developers contributing to a common file are collaborating. We will show that this coarse-grained view results in over-connecting the developer network, which obscures important latent network properties, such as community structure. Additionally, we will challenge the current working assumption that developer networks are an accurate representation of reality.

We propose an approach to construct developer networks from a VCS with a primary focus on identifying fine-grained organizational features:

1) **Developer-Network Construction:** We propose two distinct fine-grained methods that improve on the state-of-the-art (a) by analyzing the source-code structure at the function level, instead of at the file level, and (b) by analyzing the committer–author relationship, to identify closely collaborating developers.
2) **Developer-Network Analysis:** We propose a statistically sound approach of identifying and verifying developer communities (a) by applying sophisticated community-detection algorithms for detecting overlapping communities in directed and weighted graphs, which have not been used before on developer networks, and (b) by applying sound statistical methods, with carefully chosen null models and community-quality metrics, to verify that the developer communities arise from an organized effort and not as an artifact of the method.

We have applied our method to empirically study ten open-source projects, listed in Table I. We chose the projects to demonstrate our methods' applicability to a wide range of projects, from a variety of domains, written in various programming languages, and ranging in size from tens of developers to thousands.

In summary, we make the following contributions:

- We define a general approach for *automatically* constructing developer networks based on source-code structure and commit information, obtained from a VCS, that is applicable to a wide variety of software projects.
- We study ten popular open-source projects and demonstrate that the state-of-the art method of constructing

[1] The small-world property is a well understood characteristic of networks, where the distance between nodes grows with the logarithm of the number of nodes, and it is responsible for the small-world phenomenon [23].

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.73

developer networks is unsuitable to identify fine-grained organizational features, while our approach is suitable.
- We demonstrate that committer–author information can be used to *automatically* construct developer networks with similar information as developer networks constructed using the manual certificate-of-origin reporting system for documenting the responsibility of code changes.
- We present an approach to statistically evaluate the existence of developer-network communities using state-of-the-art machine-learning algorithms and network-analysis techniques suitable for directed, weighted networks with overlapping communities.
- We validate our approach by questioning 53 open-source developers from ten different projects, and show that most developers agree that the networks accurately capture reality and the identified communities have real-world meaning.

All experimental data are available at a supplementary site: http://siemens.github.io/codeface/icse/ .

II. Background

We begin with the introduction of VCSs as a data source for empirical software-engineering research. Then, we formalize developer collaboration in terms of network structure.

A. Version-Control Systems

Software engineers use version-control systems to coordinate their incremental contributions to a common software system. A VCS stores the entire source-code change history in the form of atomic change sets, called commits, which contain information about the changed lines of code and the person responsible for the change set. Through the application of data mining to VCSs, it is possible to glean insights about the coordination structure from the change history [10].

Git is a popular VCS that is especially appropriate for data mining, and it supports migration from many other VCSs [5]. Git also captures additional data that other VCSs neglect [5]. For example, in open-source projects, the author of a patch often differs from the person who commits the patch to the main development branch. For each commit, Git captures distinct author-and-committer information. Git also supports a "sign-off" (i.e., "Developer's Certificate-of-Origin") convention that helps track responsibility for a patch. A sign-off tag is a self-reported reference to anyone who authored, tested, reviewed, or committed a patch. In Section IV, we show how this information is useful for studying developer collaboration.

B. Network Analysis

The data stored in a VCS enable researchers to identify collaborative relationships between developers that arise from the software-development process. The developer relationships can be described by a network, in which nodes represent developers and edges represent collaborations between developers. A network can be formalized as a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges, denoted by $V(G)$ and $E(G)$, respectively. Some edge $e \in E$ with origin $u \in V$ and destination $v \in V$ is denoted $e = \{u, v\}$. Graph edges may be undirected or directed. In the latter case, the two edges are defined by ordered pairs, so that (u, v) and (v, u) are distinct. Weights can be assigned to edges represented by a function $w : E \to \mathbb{R}$. In our context, the edge weight represents the strength of collaboration between two developers.

By formalizing developer relationships as a graph, we can use network-analysis and visualization techniques to distill the vast amount of data into practical insights. However, networks often contain a substantial amount of noise that can conceal the latent graph structures. *Community-detection algorithms* address this problem by identifying topologically related groups of nodes. A *community* is formally characterized by a group of nodes that are densely connected to nodes within the group and sparsely connected to all other nodes in the network [29]. Communities are expected to naturally form as a result of commonalities that exist between members of a community (e.g., a shared responsibility to handle the development of a particular system component; each community would represent a division of labor and indicate an organized effort). Unfortunately, two known problems exist: (1) Networks that arise from a random process can exhibit community structure and (2) there is no guarantee that the identified communities will coincide the more abstract notions of a community (developer communities in the real world), which have characteristics that transcend the strictly topological domain. For this reason, identifying meaningful communities is not trivial and requires appropriate metrics for identifying and differentiating communities that result from a random process from communities that result from an organized process [20]. In Section III-B, we address this problem for community detection in developer networks.

III. Our Approach

We now present our approach to construct developer networks and to infer and verify developer communities using statistical techniques. All source code that implements our approach is available under the GPLv2 and MIT licenses on the supplementary website.

A. Network Construction

We propose two methods for constructing developer networks, each of which captures different views on developer collaboration.

1) Function-based method: To construct a developer network, we use a heuristic for identifying when two developers are engaged in a coordinated effort. Coordination theory has established that the demand for coordination arises from inter-dependencies between the tasks carried out by a set of individuals [25]. Therefore, the validity of the heuristic is based on how accurately it can identify inter-dependent developer tasks. Previous research relied on file-based heuristic where developers were said to be coordinated when they made a contribution to a common file [23], [24], [18]. Advantages of using a file-based heuristic include ease of computation, programming-language independence, and suitability for heterogeneous documents (e.g., source-code and configuration

files). The file-based approach has certainly proved useful for studying global network properties (e.g., vertex degree distribution, average clustering coefficient, average shortest path length) [24], however, we identified specific limitations of the file-based approach that hinder community detection and justify a more fine-grained method (see Section IV).

The activity of contributing code to a common file does not always demand a coordinated effort because files often contain a multitude of different functionalities. In our *function-based* approach, we use a more fine-grained heuristic based on code structure, where developers are considered to be coordinated when they contribute code to a common function block.[2] The rationale is that code within a function block is inter-dependent as a result of accomplishing a relatively small task, which is the key principle of functional and procedural abstraction, and which indicates that the developers of that function are engaged in a coordinated effort. A finer-grained heuristic will invariably result in identifying a subset of the developer relationships implied by a coarser-grained heuristic. By using the function-based approach, we consciously sacrifice some edges between developers in the corresponding developer network to gain the ability to detect developer communities. In Section IV-F, we empirically address this trade-off by testing whether the sacrificed edges are authentic with respect to capturing real-world collaboration. In Section IV-D, we discuss how the file-based and function-based heuristics perform with respect to identifying developer communities.

Software development is achieved through incremental contributions, where one builds on previous work to introduce or improve features or functionality through commits, which are typically only a few lines of code [32]. We capture this notion of incremental contributions by using the commits' timestamp for identifying the appropriate directions of the edges in the network. For example, developer A creates a new function without the need to collaborate closely with any other developer. At a later point, when that functionality is modified, developer B must understand and adhere to the constraints imposed by the remaining contribution of developer A. Thus, the dependency is unidirectional (developer A does not need to be aware of the contribution of developer B). By using directed edges, we enhance the graph by modeling an additional dimension of developer coordination, which is utilized by the community detection algorithm to more accurately identify communities.

To support numerous programming languages with our approach, we use the source-code indexing tool Exuberant Ctags to obtain the necessary structural information. Exuberant Ctags supports over 40 programming languages and is able to process thousands of files in seconds. It is necessarily based on heuristics for recognizing function blocks, but this is not problematic for our use case, as we discuss in Section V.

Using the author information acquired from Git, together with structural information provided by Exuberant Ctags, we construct a weighted and directed developer network. Vertices of the network represent developers who authored the code, and edges are included between two developers only when both had made a contribution to a common function block.

[2]for example, the same function implemented in C or the same method or constructor implemented in Java

We assign a weight to each edge in the network to model the varying degrees of collaboration between two developers from contributing to a common software artifact. For the function-based method, we formalize the edge weight between developers $d1$ and $d2$ collaborating on function f as

$$\omega_{d1,d2}(f) = \sum_{i=1}^{n} \sum_{j=1}^{i} |\,\mathrm{sloc_{d1}}(i,f)| + |\,\mathrm{sloc_{d2}}(j,f)|, \qquad (1)$$

where $\mathrm{sloc_{d1}(i,f)}$ is the set of source lines of code added or modified (neglecting white space additions) by developer $d1$ to function f in commit i. The commits are sorted in time-increasing order, so that a collaboration is only assigned between developer $d1$ and developers who made a *previous* commit. Equation 1 defines the collaboration between developers as a function of both temporal location and amount of contributed code made through successive changes. The nested summation captures the consecutive nature of one commit building upon the development work of all previous commits. The inner summation captures the collaboration weight between a single commit and all prior commits to that function. The outer summation then sums over all commits relevant to f. Equation 1 considers directionality of edges, therefore $\omega_{d1,d2}(f) \neq \omega_{d2,d1}(f)$ in general. Finally, the total weight between $d1$ and $d2$ is

$$w_{d1,d2} = \sum_{f \in F} w_{d1,d2}(f), \qquad (2)$$

where F is the set of all functions.

2) Committer–author-based method: Our second method is inspired by earlier work that used sign-off tags on commit messages to build developer networks [5]. In this method, tags are used to identify relationships between all people that contributed to a common commit, including authors, reviewers, and testers. A tag-based network contains important information about the software-development process, workflow, and developers with related interests and knowledge [5]. Sign-off tags are self-reported acknowledgments of participation on a commit, therefore the tag-based networks undoubtedly capture real-world collaboration. Unfortunately, only a small number of projects currently use the tag convention.

Our solution for projects that lack the tagging convention is to use the distinct author-and-committer information captured by Git to construct the network. For every commit, we place a unidirectional edge pointing from the committer to the author. The direction is important, since relationships of this type are not necessarily reciprocal. A weight for each edge is the sum of the number of commits with a common author-and-committer pair.

Since tag-based networks represent factual real-world collaborative structures, we use them (if available) to validate the structures of the automatically constructed committer–author-based networks. In Section IV-E, we show that the committer–

author network of Linux is able to capture the same information as the corresponding tag-based network.

B. Network Analysis

We now discuss how we use network-analysis and statistical methods to infer *statistically significant* communities. Additionally, in Section IV-F, we validate the community's *real-world significance* by surveying the developers that participate in the detected communities.

1) Community Detection: Community-detection algorithms allow us to decompose an arbitrary network into communities [2], [20], [11]. However, many community-detection algorithms are unable to handle weighted and directed graphs, and many more are unable to identify overlapping communities. In the case of developer networks, we expect important developers to lie at the boundary between two or more communities. If overlapping communities are not permitted, a developer will be incorrectly forced to exist in one community.

For community detection, we use the order statistics local optimization method (OSLOM), which has not been done before on developer networks. OSLOM is one of the few methods that is able to handle weighted and directed networks and to form overlapping communities [20].[3]

2) Community Verification: The validity and interpretation of the identified communities is often unclear because community-detection techniques inherently rely on principles of unsupervised learning. An important step that is often neglected is to determine whether the identified communities are meaningful [20]. We asses the validity of the observed communities by computing the probability of observing the community in an equivalent class of null-model graphs that lack a community structure. We generate the null model using a standard approach called the configuration model for random graphs, where nodes are joined uniformly at random under the constraint that the degree distribution is identical to the observed graph [15]. If it is possible to detect a statistically significant difference between the null model and observed graph communities, we can conclude that it is improbable that the topological structure of the observed developer network arose from a uniformly random process and is more likely explained by an organized process, such as a coordinated development effort.

Communities are evaluated according to *community-quality metrics*, of which several have been proposed in the literature [1]. We avoid the commonly used modularity metric in favor of *conductance* for four reasons [21]. First, modularity is known to suffer from a "resolution limit", meaning it is unable to reliably measure small communities [14]. Second, modularity is often the optimization criterion used by community-detection algorithms. By using conductance, we avoid topological-structure bias introduced by the optimality criterion imposed by the community-detection algorithms. Third, conductance allows us to characterize an individual community, whereas modularity is a global metric that considers all identified communities and does not have a meaningful interpretation for a single community [19]. Fourth, modularity is known to increase with the number of communities and nodes, making it inappropriate to compare projects of different size [13]. Although all known community-quality metrics suffer from some type of bias, conductance has been shown to exhibit reliable behavior for a wide range of cases [13].

Formally, conductance $\phi \in [0, 1]$ of a community C, in which $V(C) \subseteq V(G)$, is defined as:

$$\phi_G(C) := \frac{|\operatorname{cut}(C, G \setminus C)|}{\min\{\deg(C), \deg(G \setminus C)\}}, \tag{3}$$

where cut is the cut-set of a graph cut, and deg is the total degree of a graph [1]. Intuitively, ϕ is the probability that a random edge leaves the vertex set that composes the community. An isolated community, with no edges leaving the community-vertex set, has zero conductance. Conversely, a community with every edge leaving the community-vertex set has a conductance of one. It is important to recognize that ϕ is a function of both intra-cluster and inter-cluster edges.

To discriminate between identifying statistically significant communities and purely random topological features of the network, we employ a stochastic simulation. Given a developer network G with $N \equiv |V(G)|$ vertices (developers) and E edges (connections between developers), we apply a community-detection algorithm to identify a set of communities $\mathcal{C} = \{C_1, C_2, \ldots, C_i\}$ where $V(C_i) \subseteq V(G)\ \forall i$. Mean conductance over all communities is given by $q_G(\mathcal{C}) = \sum_{C \in \mathcal{C}} \phi_G(C)/|\mathcal{C}|$.

Using these input data, we generate a null model that represents an equivalent developer network but with disorganized collaboration. To generate the null model, we randomize the original network according to a configuration model using a graph-rewiring technique, with which the pairs of edges are selected uniformly at random and the end points swapped, such that an edge pair (u, v) and (s, t) is rewired to (u, t) and (s, v) [15]. The rewiring procedure maintains the amount of participation (i.e., number of edges) for each developer, but destroys the preferential attachment to a particular group of developers. The rewiring procedure is executed m times[4] to generate a set $\mathcal{R} = \{R_1, R_2, \ldots, R_m\}$ of rewired graphs with $V(R_i) = V(G)\ \forall i$.

The degree distribution, which represents the amount of participation by each developer, is given by $P_C(k) = |\{c \in C | \deg(c) = k\}|/|N|$. The rewiring procedure is intentionally designed to maintain the original degree distribution, that is,

$$P_{R_i}(k) = P_G(k) \quad \forall R_i \in \mathcal{R}. \tag{4}$$

[3]We experimented with several other community-detection algorithms and experienced generally poor performance from basic techniques, such as random-walk or eigenvector based methods [19]. A statistical-mechanics approach using spin-glasses had comparable performance to OSLOM, but it does not produce overlapping communities [11].

[4]We ensured that our choice $m = 3000$ was sufficiently large by checking the convergence of all derived results.

For each rewired graph R_i, we calculate the mean conductance $q_{R_i}(\mathcal{C})$ and define the probability distribution as

$$P_{\mathcal{R}}(Q) = \frac{|\{i \in [1, |\mathcal{R}|] \mid q_{R_i}(\mathcal{C}) = Q\}|}{|\mathcal{R}|}. \quad (5)$$

With standard hypothesis testing, we can then evaluate whether the collaboration structure is statistically significant. We check whether it is possible that the observed graph could be described by the generated null model with the equivalent degree distribution as the observed graph; if this is not the case, we conclude that our observation is not described by a uniformly random process.

More precisely, the null hypothesis H_0 that the observed mean conductance $q_G(\mathcal{C})$ is described by the conductance distribution of the rewired (null model) graphs with a nonvanishing probability is given by,

$$H_0 : P_{\mathcal{R}}(Q = q_G(\mathcal{C})) > \epsilon, \quad (6)$$

with the alternative hypothesis given by $H_1 : P_{\mathcal{R}}(\cdot) \leq \epsilon$.

We use a one-sample t test to evaluate the hypothesis with the standard significance level of 0.05. Since the t test is robust against the deviation from a normal distribution with large sample sizes (i.e., larger than 30), we do not need to check our data for a normal distribution. We present the results of the statistical test for all subject projects next.

IV. Evaluation & Results

We now present our hypotheses and findings on the network properties of developer networks we constructed for ten open-source projects. To address our hypotheses, we compare developer networks constructed using the prevalent file-based method and the more fine-grained methods we propose. In Section IV-F, we present the results of a developer survey to address the validity of our approach with respect to capturing real-world collaboration.

A. Hypotheses

In order to derive value and utility from developer networks, previous work has largely assumed that the networks are an accurate representation of developer collaboration [26]. We will challenge this fundamental assumption about developer networks by investigating the local topological features that should be present if the network is indeed an authentic representation of developer collaboration. Though other views are possible, we then validate that this particular view on collaboration aligns with developers' perceptions (see Section IV-F).

Software development is an organized process and, if a developer network faithfully captures real-world developer collaboration, it should also exhibit an organized structure.

H1—*Developer networks exhibit identifiable communities that significantly exceed the magnitude of organization that is expected from an equivalent unorganized process.*

By an equivalent unorganized process, we mean a situation that is equivalent to the original process except that developers' contributions to the software system are randomly distributed across various system components, showing no particular organized responsibility toward a particular aspect of the system.

The standard method of constructing developer networks relies on file-level information to identify collaborating developers. We show that this approach is insufficient for identifying the latent community structure as a result of over-connecting developers in the network. Dense networks are known to hinder community-detection algorithms [6]; furthermore, prior work has shown this problem arises for file-based developer networks [18].

H2—*Developer networks constructed using the standard file-based approach fail to identify statistically meaningful communities, whereas a more fine-grained function-based approach is able to identify statistically meaningful communities.*

The manual process of tagging a commit is an intentional acknowledgment of one's participation in a commit. Each developer only tags a commit once they have made a contribution to the code. Therefore, a developer network constructed on the basis of commit tags can be regarded as a faithful representation of real-world collaboration. To evaluate the validity of the committer–author-based method, we quantify congruence between the ground truth tag-based network and our automatically-constructed committer–author network.

H3—*Tag-based developer networks constructed from the manual process of tagging commits are highly congruent with automatically determined committer–author-based networks.*

B. Subject Projects

We selected ten open-source projects, listed in Table I, to evaluate the methods we proposed; the projects vary by the following dimensions: (a) size (lines of source code from 50 KLOC to over 16 MLOC, number of developers from 15 to 500), (b) age (days since first commit), (c) technology (programming language, libraries used), (d) application domain (operating system, development, productivity, etc.), (e) development process employed, and (f) VCS used (Git, Subversion).

For each project, we analyze the VCS for a 3-month window starting in the second quarter of 2014. While window size certainly influences the resulting network, the impact of enlarging the window beyond 3 months is marginal [26].

C. Existence of Statistically Significant Communities

To test hypothesis H1, we used our function-based method to construct developer networks for all subject projects. We expected statistically significant communities to exist as a result of an organized software-development process in, at least, some of the subject projects, and we now evaluate whether our method is able to identify the communities using the community-verification procedure described in Section III-B2.

As an example, Figure 1 clearly shows the separation between the observed developer-network conductance of QEMU and the conductance distribution for the unorganized (rewired) network. The small p value of the t test indicates that the observed communities are statistically significant. Table I

TABLE I
OVERVIEW OF SUBJECT PROJECTS FOR A 90-DAY DEVELOPMENT WINDOW AND COMPARISON OF COMMUNITY CONDUCTANCE FOR THE ORIGINAL (OBSERVED) AND RANDOMIZED (REWIRED) NETWORKS.

Project	Devs	MLOC	Lang	Domain	Observed conductance	Rewired conductance
Linux	580	16	C	OS	0.05	0.88
Chromium	500	6.5	C, C++	User	0.20	0.74
Firefox	400	9.3	C++, JS	User	0.11	0.79
GCC	70	6.2	C, C++	Devel	0.01	0.48
QEMU	50	0.78	C	OS	0.39	0.56
PHP	50	2.2	PHP, C	Devel	0.15	0.80
Joomla	30	1.3	PHP, JS	Devel	0.57	0.84
Perl	30	4.5	Perl, C	Devel	0.49	0.66
Apache http	15	2.2	C	Server	0.27	0.80
jQuery	15	0.05	JS	Library	0.49	0.75

summarizes the results for all the subject projects: The function-based method identifies strong communities in several of the subject projects and a statistically significant difference between the original and rewired networks.

In conclusion, we reject the null hypothesis that the observed developer networks exhibit communities that could arise from an unorganized process. Thus, we *accept H1*.

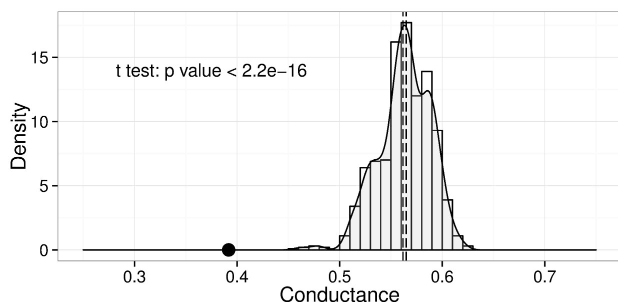


Fig. 1: Community significance test on the observed mean community conductance (black dot) against the distribution of mean community conductance for 3000 rewired graphs for QEMU development from 14.2.2014 to 14.5.2014. Vertical lines represent the 95% confidence intervals.

D. Comparison of File-Based and Function-based Methods

To test hypothesis H2, we performed a comparison between the file-based method and the function-based method for constructing developer networks (cf. Section III-A1). The comparison draws attention to limitations of the file-based approach that manifest as the inability to identify statistically significant communities. In particular, we evaluated the mean community conductance and mean community density for two revisions of each project. Graph density is a measure of graph connectedness, where a complete graph has density 1, and a graph with no edges has density 0. Figure 2 shows a scatter plot of mean community conductance versus mean community density, in which each point represents a three month project revision. We see an approximate but distinct separation between the file-based and function-based networks. Communities identified in the function-based network are both

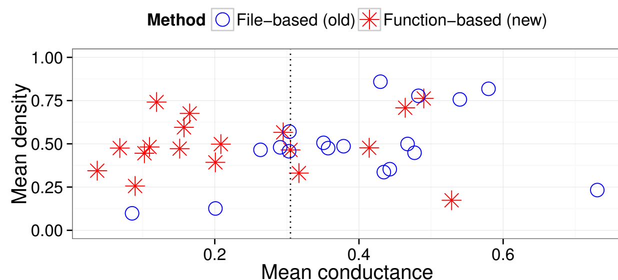


Fig. 2: Scatter plot of projects analyzed using both file-based and function-based methods for two different revisions. A clustering by crosses (left) and circles (right) is visible; the function-based approach is able to resolve more significant communities without compromising density.

internally dense and exhibit low conductance (i.e., strong community structure). In contrast, the file-based communities are dense, which we would expect because of the overall high density of the network, but exhibit high conductance (i.e., weak community structure). From this result, we can conclude that the edges that are neglected by the function-based method are the ones which cross community boundaries. For this reason, we see the function-based and file-based communities exhibit similar levels of internal density, but the conductance in the function-based communities is lower. In Section IV-F, we address the validity of the file-based edges that are crossing a community boundary and ignored by the function-based approach. In summary, this result demonstrates that the finer granularity of the function-based method enables the discovery of statistically significant communities, but is not excessively fine such that it destroys the connectedness of the graph.

The probability density plot shown in Figure 3 further illustrates the significant difference between the function-based and file-based network communities. There is a clear separation between the distributions where the function-based method identifies significantly stronger communities compared to the file-based method. We performed a paired t test to evaluate whether the difference in the distributions is statistically significant. Before performing the t test, we checked

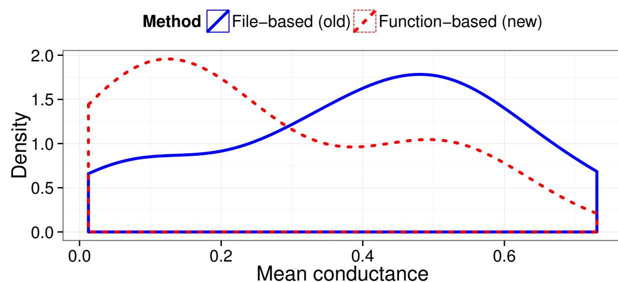


Fig. 3: Density plot of mean community conductance computed for each project comparing the file-based and function-based methods.

the distributions for normality using a Shapiro-Wilk test. It produced p values of 0.99 and 0.075 for the file-based and function-based distributions, respectively. The following t test generated a p value of 1.29×10^{-5}. Thus, we can confidently reject the null hypothesis that the difference between the two measurements has a zero mean value.

We visualize the developer networks of QEMU, created using the file-based and function-based methods in Figure 4. It illustrates the inability of the file-based approach to identify statistically meaningful communities with the example of the QEMU developer network. Each bounding box represents a single community of developers. The border color of each box uniquely identifies each community, and pie charts are used to represent each developer's relative participation in a community. A box's background color is used to represent the significance of each community, calculated according to the conductance distribution (cf. Section III-B2). Green represents a significant (strong) community and yellow represents an insignificant (weak) community. Intra-community edges are shown in black, and inter-community edges are shown in red. The edge thickness represents the strength of a relationship. We use PageRank centrality to identify important developers, denoted by the size of each node.[5]

Figure 4 illustrates that *all* communities identified by the file-based method (left side) for QEMU are insignificant. The conductance of the communities is on the order of what is expected from a unorganized (rewired) network, represented by the yellow background color; it indicates that the file-based method failed to capture the organized structure of developer collaboration. In contrast, the function-based method is capable of identifying several significant communities in the same project. Notice further that the file-based method has generated an extremely dense network, in which nearly every developer is contributing in every community, visible by the large number of multicolored nodes. In comparison, the developer network constructed using the function-based method is less dense; it has identified developers that make contributions only within one or two communities, which is visible by the large number of single-colored nodes.

In summary, we conclude that the file-based method fails to identify statistically meaningful communities as indicated by high conductance values that are statistically equivalent to the unorganized networks. In contrast, the function-based approach was able to identify the latent community structure that was concealed by the file-based approach. We emphasize that we use conductance here to evaluate whether the topological structure exhibits statistically significant communities, but we have not made any judgment about the communities quality beyond strictly topological features. Thus, we *accept H2*.

[5]To reduce clutter, we filter the inter-community edges by aggregating the edge multiplicity between two communities into a single edge, connecting the two most important developers. The weight of an inter-community edge represents the total collaboration between all nodes in the connected communities.

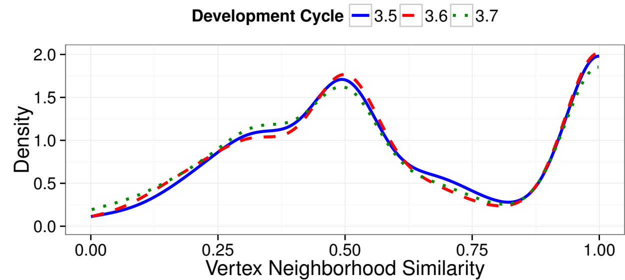


Fig. 5: Comparison between tag-based and committer–author-based networks. Each curve represents a single revision of Linux. The similarity between each developer neighborhood is shown as a density plot with a mean of 70% similarity.

E. Tag-Based and Committer–Author-Based Network Similarity

To test hypothesis H3, we opt for an alternative approach to validating the network structure because we have access to a ground-truth network constructed using commit tags. We constructed developer networks for three revisions of the Linux kernel using the tag-based method (as provided by Linux's VCS) and the committer–author-based method (automatically constructed by us, as described in Section III-A2). We chose Linux as the *sole* test subject, because it enforces a strict tagging convention for every commit, which the other subject projects do not.

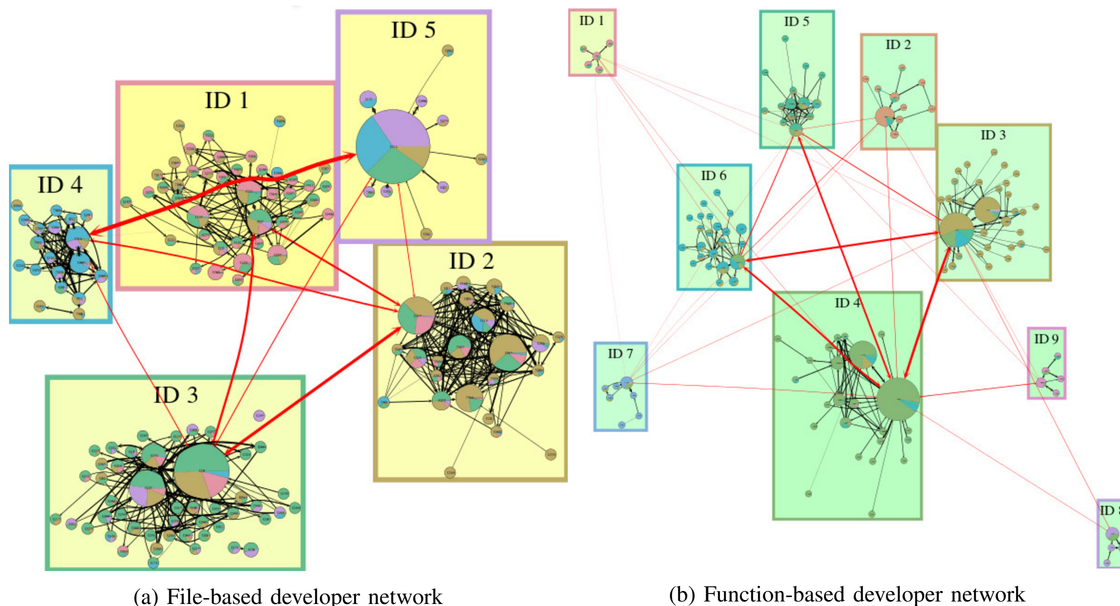
For each revision, we compute the similarity between the tag-based and committer–author-based networks using a graph matching strategy based on the Jaccard index [13]. Figure 5 shows the results as a density plot. We see a bimodal distribution with peaks at 100% similarity (perfect match) and 50% similarity. The average taken over the three revisions is 70% match between the two networks. The probability of having a 70% match between two labeled random graphs, with equivalent size of the committer–author-based network, is less than a half of one percent; hence, we conclude that the committer–author-based network is an authentic representation of developer collaboration, and we *accept H3*.

F. Network Validation

Goals: We now address whether the function-based approach accurately captures real-world developer collaboration by means of a survey. There is no need to include committer-author networks in the survey because they are constructed from direct references to developer collaboration. The template of the questionnaires we used in the survey is available at the supplementary website. Specifically, we address two research questions (RQ):

RQ1—*Do the network edges, weights, and directions, accurately represent real-world collaborative relationships as they are understood by developers in the project?*

RQ2—*Do the identified communities represent developer communities that have real-world significance?*



(a) File-based developer network

(b) Function-based developer network

Fig. 4: Developer networks constructed from QEMU v1.1. All communities in the file-based network are insignificant (yellow background color). The function-based method identified several significant communities (green background color).

Participants: We selected participants that made a contribution to one of the projects shown in Table I within the previous year, whose contact data we extracted from commit messages of publicly available VCSs. For each project, we constructed the developer networks and decomposed them into developer communities of, at least, five developers (fewer developers typically contain an insignificant number of collaborations). From the population of 6704 developers, we randomly selected 521 developers of different levels of involvement (e.g., single contributor to project lead), responsibilities (e.g., developers from various subsystems), and roles (e.g., tester, reviewer, bug fixer).

Experimental Material: We conducted the survey online. It included, among others, demographic questions and the following survey questions (SQ):

SQ1—*Whom did you collaborate closely with during the development of version X?*

SQ2—*Does the following network accurately represent collaborative relationships between developers?*

SQ3—*Do the developers shown in the above network represent a developer community?*

For SQ1, we provided auto-completion (obtained from analyzing the VCS) to help with recall and correct spelling of developers' names. For SQ2 and SQ3, we displayed a resizable visualization of the community network (not the entire developer network), labeled with the developer names. Both questions had to be answered on a five-point Likert scale [22], ranging from strongly disagree to strongly agree. Additionally, participants could enter a free-format response. A pilot of the survey with ten testers did not reveal any significant issues.

Analysis: In total, 53 developers of the 521 that we contacted completed the survey. We show the responses to the Likert-scale questions in Figure 6.

Regarding the network-accuracy question (left in the figure): Almost half of the participants agree or strongly agree, and a quarter disagrees or strongly disagrees that the network accurately captured developer collaborative relationships. We see a similar distribution for the community-authenticity question (right in Figure 6).

Furthermore, we received a number of written responses for each question and categorized them manually. Regarding network accuracy, 13 written responses were given: 8 referenced missing developers or collaboration, 3 referenced incorrect collaboration, and 4 made various comments, such as, "Interesting Survey!". Regarding community authenticity, 14 responses were given: 8 stated that the network is accurate and provided a real-world meaning to the community, 2 responses stated that the network is accurate, 3 responses stated the identified communities are partially accurate, and 2 responses stated the network was inaccurate.

In Section III-A1, we identified that the function-based networks contain less edges than the file-based networks. We now address whether the missing edges distort the view on collaboration by neglecting authentic relationships. Unfortunately, we are unable to directly compare the file-based and function-based communities, because the file-based networks are extremely dense and hinder community-detection algorithms [18]. Instead, we focus our attention to the edges that cross a community boundary, because these edges conceal the community structure and the results of Section IV-D indicate that most missing edges are in fact cross-community edges. We used the responses from SQ1 to test the authenticity of cross-

community edges neglected by the function-based method. To accomplish this test, we first identified the communities using the function-based method. We then used the function-based communities to identify all the edges that crossed community boundaries in the equivalent file-based network (i.e., same project and revision). We then removed all developers from the network who did not answer the survey. Finally, we calculated the percent of file-based cross-community edges that were confirmed by survey responses. For two subject systems (Linux and QEMU), we collected a sufficient amount of data (148 ground-truth edges). For QEMU, we acquired 47 ground-truth edges between 25 developers. Among the 25 developers, we found 82 file-based edges that cross a community boundary and were neglected by the function-based method. On average, 15.3% of the edges crossing a community boundary are authentic with a median of 7.7%. For Linux, we acquired 101 confirmed edges, and none of the 27 file-based edges crossing a community boundary were authentic. From these results we conclude that most of the edges that obscure the community structure in the file-based approach are in fact unconfirmed by developers that answered the survey and appear to be an artifact of the method.

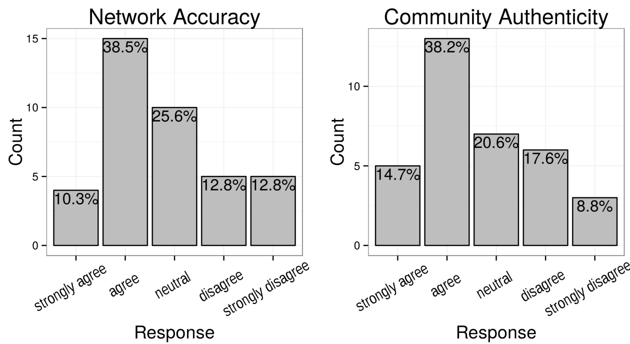


Fig. 6: Developer survey responses to questions stated in Section IV-F (SQ2 left, SQ3 right).

Interpretation & Discussion: Both histograms in Figure 6 illustrate a substantial quantity of agreement responses for both the network-accuracy and community-authenticity questions. Additionally, in the written responses, we see that developers largely agree that the networks are accurate and that the identified communities have real-world meaning. Interestingly, the responses indicate that our approach is precise in the sense that the identified collaborations and communities are authentic, but the approach has imperfect recall, because some edges are missing. Compared to the file-based method, we noted that the function-based method contains less edges. For Linux and QEMU, we were able to investigate the authenticity of missing edges that influence the community structure, and we found that very few file-based edges that are not captured in the function-based method were authentic. Given that our approach is fully automated and based on a single data source, this is a very encouraging result. To be fair, it was not to be expected that our approach achieves perfect recall rates, since not every mode of collaboration is manifested in the VCS. Still, despite this lack of information, many developers agree that the identified communities capture a logical partitioning of developers.

V. Threats To Validity

We applied our approach to ten manually selected subject projects, which threatens external validity. We chose projects of different but substantial sizes, with long and active development histories, and from different application domains to cover a considerable diversity of projects. Despite the diversity of our subject projects, they still represent only a fraction of the total diversity of software projects. Furthermore, it is unclear as to whether our results generalize to commercial projects since all of the subject projects are open-source. Commercial project data is generally well protected, making such studies difficult. For the tag-based networks, only a few projects employ a strict tagging convention, so we cannot generalize the results.

We realize that incomplete or incorrect recollection of a developer's collaborative relationships could compromise the survey responses. To help mitigate the consequences, we displayed the labeled developer network beside the survey questions. The compromise is that developers may only recall the collaborative relationships that are shown in the network and still forget others. However, this only influences the completeness and not the correctness of the responses. Since we already characterize our method as partially incomplete, the consequences of displaying the developer network does not affect our conclusions. A control group formed by including a secondary survey with randomized networks could increase confidence in the results, however, we recognized that the response rate is low and a control group would further reduce the already small experimental group size.

The use of Ctags to identify fine-grained collaborations at the function level is based on heuristics. This leaves room for introducing misclassified collaborations. However, this threat to validity is minor, as only many misclassifications would influence the outcome of the statistical analyses we applied. As Ctags is widely used in practice, this is not to be expected. In a sense, we accept this minor threat in exchange for an approach that is language independent.

VI. Related Work

Developer networks constructed from VCS data was first done by Lopez-Fernandez et al., where developers were linked based on contributions to a common module [23], [24]. Huang et al. improved Fernandez's work by automating module classification using knowledge of file directories [17]. The results indicated that modules may not provide detailed enough information to be useful. Improvements were made by narrowing the collaboration assumption to common file contributions to identify more fine-grained collaboration [18]. Narrowing the definition of collaboration helped to identify more subtle features than with the module-based assumption, but the authors noted that the networks were still too dense to identify community structure. In our approach, we use an even finer-grained definition of collaboration to further

reduce the density of the network, which enabled us to uncover community structures, in the first place. Previous work mostly applied metrics that do not produce rich visualizations, such as degree distributions or centrality plots, and no one has visualized community structure [24], [23], [17], [33], [26]. We are aware of one paper focusing on visualization of developer collaboration using a file-based approach, but community-detection was not possible without edge filtering due to the extreme density of the developer networks [18].

Toral et al. applied social-network analysis to investigate participation inequality in the Linux mailing list that contributes to role separation between core and peripheral contributors [33]. Bird et al. investigated developer organization and community structure in the mailing list of four open-source projects and used modularity as the community-significance measure to confirm the existence of statistically significant communities [4]. Panichella et al. constructed developer networks based on mailing-list and issue-tracker data to identify developer teams and examine the driving forces behind splitting and merging teams during system evolution [30]. Our work differs by constructing networks from source-code contributions, instead of communication networks based on e-mail archives or issue trackers. Additionally, we apply our approach to a diverse set of projects and show that our findings have real-world significance.

Bird et al. examined the influence of code ownership on defect proneness at the component level of two commercial software products [3]. They operationalize ownership based on the percentage of commits to a component made by a single developer, however, in open-source projects component ownership is rarely dominated by a single individual [34]. Our work is complementary by supporting the identification of developer communities, which can be used to study ownership at the community level instead of the developer level.

Cataldo et al. examined the important concept of socio-technical congruence and its impact on development productivity and software quality [8], [7]. Based on knowledge of work dependencies and technical dependencies, they identified coordination requirements. The “fit” between the actual coordination and required coordination was examined with the conjecture that high congruence is a desirable property. To establish the actual developer coordination, they used a-priori knowledge of developer teams, manual investigation of communication logs, and modification requests. Our work contributes to their framework by providing a fully automated method to identify modes of coordination using only data from the VCS.

Previous work utilized the Linux tagging convention to construct a developer network consisting of people involved in reviewing, acknowledging, and testing commits [5]. We extended this work by proposing a method to extract similar information for projects that do not use the manual tagging convention, to automate the approach, and we validated it against the tag-based network for Linux.

Meneely et al. addressed the question of whether networks constructed from VCSs using the file-based approach captured real-world collaboration [26]. They concluded that the file-based networks were largely representative of developer perception, but that the networks suffered from errors in missing collaboration and also falsely suggesting collaboration. In contrast, our survey revealed that the more fine-grained approach mainly suffers from missing edges. Furthermore, we extended on the original questionnaire format by allowing the participants to observe the developer network directly, instead of only displaying a list of names.

VII. Conclusion

The ability to accurately capture collaborative relationships in large software projects is a valuable asset to project management, developer productivity, and software quality. Despite considerable advances, current methods fail to recognize fine-grained organizational structures and prominent structural features that differentiate one software project from another, or they require developers to manually document their involvement in a collaboration.

We proposed a fine-grained and automatic approach to identify the community structure of a software project based on source-code structure and committer–author information obtained from VCSs. We used a set of statistically sound methods to identify and verify developer communities.

We evaluated our approach (in particular, the function-based method) on ten diverse open-source projects, with complex and active histories, from a variety of domains, written in various programming languages, and of different sizes. We found that the developers of these projects form statistically significant communities, and we were able to identify and visualize them automatically using our approach, which has not been accomplished in previous work.

From a survey of 53 open-source developers, we learned that most developers agree that the network accurately depicts reality and the developer communities have real-world meaning. Furthermore, we found that the predominant source of error was from missing collaborative links; the links that were identified are largely accurate. We were able to show that, while the finer-granularity of our approach inherently sacrifices some edges, only a small percentage of edges concealing the community structure in the file-based networks are authentic. Given the abstract nature of a human-centric concept, such as collaboration and community structure, and our fully automated method of detection, we find our results encouraging and supportive of the validity of our approach.

Since our analysis suite can process other types of socio-technical data beyond VCSs, future work shall investigate approaches that integrate data from heterogeneous sources, including e-mail archives, wikis, and issue trackers.

Acknowledgment

We thank all participants of the online survey. This work has been supported by Siemens and the DFG grants AP 206/4, AP 206/5, and AP 206/6.

References

[1] H. Almeida, D. Guedes, W. Meira, and M. J. Zaki. Is there a best quality metric for graph clusters? In *Proceedings of the 2011 European conference on Machine learning and knowledge discovery in databases - Volume Part I*, ECML PKDD'11, pages 44–59. Springer-Verlag, 2011.

[2] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 44–54. ACM, 2006.

[3] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.

[4] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Latent social structure in open source projects. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 24–35. ACM, 2008.

[5] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 1–10. IEEE Computer Society, 2009.

[6] U. Brandes, M. Gaertler, and D. Wagner. Experiments on graph clustering algorithms. *Algorithms-ESA 2003*, pages 568–579, 2003.

[7] M. Cataldo and J. D. Herbsleb. Coordination Breakdowns and Their Impact on Development Productivity and Software Failures. *Software Engineering, IEEE Transactions on*, 39(3):343–360, 2013.

[8] M. Cataldo, J. D. Herbsleb, and K. M. Carley. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 2–11. ACM, 2008.

[9] C. R. B. de Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson. How a good software practice thwarts collaboration: the multiple roles of apis in software development. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, SIGSOFT '04/FSE-12, pages 221–230. ACM, 2004.

[10] D. Draheim and L. Pekacki. Process-centric analytical processing of version control data. In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 131–136. IEEE, 2003.

[11] E. Eaton and R. Mansbach. A Spin-Glass Model for Semi-Supervised Community Detection. In *AAAI*, pages 900–906, 2012.

[12] S. Eppinger, D. Whitney, R. Smith, and D. Gebala. A model-based method for organizing tasks in product development. *Research in Engineering Design*, 6(1):1–13, 1994.

[13] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[14] S. Fortunato and M. Barthelemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

[15] C. Gkantsidis, M. Mihail, and E. Zegura. The Markov Chain Simulation Method for Generating Connected Power Law Random Graphs. *ALENEX*, 2003.

[16] J. D. Herbsleb and M. Cataldo. Architecting in software ecosystems: interface translucence as an enabler for scalable collaboration. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 65–72. ACM, 2010.

[17] S.-K. Huang and K.-m. Liu. Mining version histories to verify the learning process of Legitimate Peripheral Participants. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[18] A. Jermakovics, A. Sillitti, and G. Succi. Mining and visualizing developer networks from version control systems. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, pages 24–31. ACM, 2011.

[19] R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *J. ACM*, 51(3):497–515, 2004.

[20] A. Lancichinetti, F. Radicchi, J. J. Ramasco, and S. Fortunato. Finding Statistically Significant Communities in Networks. *PLoS ONE*, 6(4):e18961, 2011.

[21] J. Leskovec, K. J. Lang, and M. W. Mahoney. Empirical Comparison of Algorithms for Network Community Detection. *Proceedings of the 19th international conference on World wide web WWW 10*, 30(3):631–640, 2010.

[22] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22:1–55, 1932.

[23] L. López, G. Robles, Jesús, and I. Herraiz. Applying Social Network Analysis Techniques to Community-driven Libre Software Projects. *International Journal of Information Technology and Web Engineering*, 1(3):27–48, 2006.

[24] L. Lopez-Fernandez, G. Robles, and J. M. Gonzalez-Barahona. Applying Social Network Analysis to the Information in CVS Repositories. In *1st International Workshop on Mining Software Repositories (MSR)*, pages 101–105. IET, IET, 2004.

[25] T. W. Malone and K. Crowston. What is coordination theory and how can it help design cooperative work systems? In *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, CSCW '90, pages 357–370. ACM, 1990.

[26] A. Meneely and L. Williams. Socio-technical developer networks: should we trust our measurements? In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 281–290. ACM, 2011.

[27] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 13–23. ACM, 2008.

[28] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 521–530. ACM, 2008.

[29] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69(2):26113, 2004.

[30] S. Panichella, G. Canfora, M. Di Penta, and R. Oliveto. How the evolution of emerging collaborations relates to code changes: an empirical study. In *ICPC*, pages 177–188, 2014.

[31] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 2–12. ACM, 2008.

[32] D. Riehle, C. Kolassa, and M. A. Salim. Developer belief vs. reality: The case of the commit size distribution. In *Software Engineering*, pages 59–70, 2012.

[33] S. Toral, M. Martínez-Torres, and F. Barrero. Analysis of virtual communities supporting oss projects using social network analysis. *Information and Software Technology*, 52(3):296–303, 2010.

[34] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008.