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

Mitchell Joblin
Wolfgang Mauerer
Siemens AG
Erlangen, Germany

Sven Apel
University of Passau
Passau, Germany

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

ABSTRACT

The successful design and implementation of any complex system requires a coordinated effort. Unfortunately, a comprehensive understanding of 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 developer coordination, based on commit information and source-code structure, mined from a version control system. We build on well-established theory from complex network analysis and machine learning to identify developer communities. In contrast to previous work, our method is fine-grained, and identifies communities using order-statistics and a sophisticated community-evaluation technique based on graph conductance. We analyze 10 open-source projects with complex and active histories, written in several programming languages, to demonstrate the scalability and generality our approach. Our results demonstrate that developers of open-source projects form statistically significant community structures, and that we are able to identify and properly visualize them automatically.

1. INTRODUCTION

Software development 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 [18]. Software developers are required to coordinate their efforts to handle software 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 software developers [10, 23, 14, 18].

Recent empirical research has illustrated the significance of developer organization on software quality [10, 39, 36, 37]. It even suggests that evaluation methods that are entirely based on organizational metrics are more indicative of fault proneness than traditional source-code-centric metrics [37].

The organizational structure among software developers has also been shown to influence a software system's architecture. Conway's Law states that "organizations that design systems are constrained to produce systems which are copies of the communication structures of these organizations," [13]. Conway's law has been established both in theory and through empirical evidence [12, 37, 9]. Furthermore, it has been shown that achieving congruence between the developer organization and the system architecture allows developers to work more productively and to adhere more easily to the initial design goals [11].

Modern software development is becoming progressively more ambitious, involving larger numbers of geographically distributed developers leading to increased technical coordination complexity [30, 7, 17, 24]. Commercial style development frequently imposes mandated organizational structures, however, it is unknown whether the mandated structures are obeyed. Furthermore, organizational charts generally fail to capture the dynamic property of developer collaboration. Despite the importance of developer organization, comprehensive, accurate and up-to-date knowledge of developer organization is rarely available to support software-engineering decisions.

Recently, version control systems (VCS) have been utilized to construct developer networks that capture the organizational structure [31, 32, 25, 26, 35]. However, the primary focus has been, so far, on characterizing the coarse-grained properties that govern all developer networks, such as the small-world property.[1] The state-of-the-art method to construct developer networks assumes that all developers contributing to a common file are collaborating. This coarse-grained assumption results in an over-connecting developer network, which obscures subtle but important fine-grained network properties, such as community structure.

We propose an approach to construct developer networks from a VCS with a primary focus on identifying the fine-grained organizational structure and prominent structural features that differentiate one software project from another. Overall, our approach consists of two parts:

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 struc-

[1]The small-world-network 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 [31].

ture 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 for identifying and verifying developer communities (a) by applying state-of-the-art, community-detection algorithms for overlapping, directed, weighted networks, which have not been used before on developer networks, and (b) by applying sound statistical methods and community-quality measurements to verify that identified communities arise from an organized effort and not as an artifact of the method.

We have applied our method to empirically study 10 open-source projects, listed in Table 1. 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 constructing developer networks based on source-code structure and commit information, obtained from a VCS, that is applicable to a wide variety of different software projects.
- We perform case studies on 10 popular open-source projects and demonstrate that state-of-the-art methods to construct developer networks are unsuitable to identify fine-grained organizational features, and that our methods are 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 and quality of developer-network community structure using state-of-the-art machine-learning algorithms and network-analysis techniques suitable for directed, weighted networks with overlapping communities.
- Using expert knowledge, we validate that the automatically constructed developer networks for Linux resemble real-world collaboration.

All experimental data are available at a supplementary web site: `www.infosun.fim.uni-passau.de/spl/prosoda/`.

2. BACKGROUND

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

2.1 Software Repositories

Software engineers coordinate their effort through the use of elaborate data-logging systems or software repositories that capture source-code changes, bug reporting, task management, and developer communication. VCSs store the entire source-code history in the form of commit messages containing all lines of code added, changed, and deleted, an identifier of the person responsible for the commit, the time of when the commit was made, and a note documenting in natural language what the change encompasses. The abundance of data stored in the VCS hold useful insights for software engineering [15], however, VCSs have not been originally intended to be used for this purpose. Therefore, data-mining techniques are required to distill the VCS data into software-engineering insights.

Git is a VCS that is very popular and lends itself particularly well to data mining [6]. It provides a sensible base to build mining tools on, because nearly every other VCS can be converted into a Git repository. The distributed nature of Git avoids the network overhead of centralized VCSs leading to fast mining operations, and Git reliably captures additional data that other VCSs neglect, such as distinguishing author and committer [6]. For example, in open-source projects only core-developers have commit rights, the other developers must submit their code contributions to a core-developer in the form of a patch. Git is able to reliably maintain distinct author-and-committer information for each submitted line of code. Git also supports a "tagging" convention or more generally a "certificate-of-origin" that allows anyone involved in a commit to include a note in the commit message as part of a manual reporting process. The result is the creation of a trail of responsibility that captures everyone who authored, tested, reviewed, and finally committed the code. We leverage Git in our work to discover insights to software-engineering collaboration that would be lost by other VCSs.

2.2 Network Analysis

The data captured by software repositories allows one to identify collaborative relationships that arise during the software-development process. These relationships can then be organized in a network structure, where nodes represent software artifacts (*e.g.*, files or software components), software developers, or both. An edge is used to represent a relationship between two nodes. For example, an edge placed between two developers could represent collaboration on a software project. The networks 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 un-directed or directed. In the latter case, the two edges defined by ordered pairs (u, v) and (v, u) are interpreted to have different meaning. Weights can be assigned to edges represented by a function $w : E \to \mathbb{R}$. The weight can be interpreted in our context as the magnitude of relationship strength between two vertices.

A network can serve to effectively visualize complex relationships between a set of objects and hold a great deal of information in the local and global network topology. Additionally, networks often contain a substantial amount of noise that can obscure the important information. Community detection is one way to address this problem by identifying strongly related groups of nodes or communities. A community is characterized by a group of nodes that are densely connected to nodes within the group and loosely connected to all other nodes in the network [38]. A community naturally forms as a result of commonalities that exist between members of the community. For example, the commonalities could result from the shared responsibility to the handle development of a particular system component, and each community would represent a division of labor and indicate an organized effort. Unfortunately, a network constructed by randomly assigning edges between a set of nodes will also exhibit a small magnitude of community structure. For this reason, identifying communities is not trivial and requires appropriate community metrics for identifying and differentiating communities that result from a random process from communities that result from an organized process [28]. We present a novel solution to this problem in Section 3.

3. OUR APPROACH

We now present our approach to construct developer networks from Git with a focus on a fine-grained view of developer collaboration. All source code of the implementation of our approach is available under the GPLv2 and MIT licenses on the web: `github.com/wolfgangmauerer/prosoda/`.

3.1 Network Construction

We propose two novel methods to construct developer networks from a Git repository to gain a comprehensive view of the collaborative activities of a software project. The two methods produce networks with distinctly different information, the first method captures technical collaboration at the source-code level, and the second method captures coordination between source-code authors and the core-developers that administrate the project.

3.1.1 Function-based method

To construct a developer collaboration network, some assumptions on what constitutes collaboration must be made. We use a definition of collaboration in which a dependency forms the basis of collaboration [33]. For example, when code written by one developer depends on code written by another developer, the two must collaborate either implicitly or explicitly to achieve a successful implementation. Previous research assumed that any developers who made a contribution to the same file were collaborating [31, 32, 26]. However, files can be large with many functions that are independent. We propose a more fine-grained view of collaboration by assuming collaboration to exist only when two developer contributed code to the same function block.[2] The rationale is that code within a function block depends on itself to accomplish a relatively small task, which is the key of functional and procedural abstraction, and forces the developers of that function to coordinate their effort. A broader definition of collaboration would obscure the subtle features of a community, whereas the narrower definition enables our method to uncover the developer communities, as we will discuss in Section 4.

Extracting structural information from source code normally would imply a parsing step. We carefully designed our approach to be scalable to a wide range of projects covering many programming languages, thus deeming traditional parsing methods impractical. Instead, 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, but this is not problematic for our use case, as we will discuss in Section 5.

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. We assign a weight to each edge in the network, to account for the fact that developers have varying degrees of involvement within a project, resulting in varying degrees of collaboration between developers. The edge weight is a function of the frequency of collaboration and the number of line of code changed, added or deleted excluding whitespace changes.

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 [40]. We capture the notion of incremental contributions by using directed edges in the developer 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 contribution of developer A. Therefore, the dependency is unidirectional, because developer A does not need to be aware of the contribution of developer B. Capturing this subtle feature is imperative to identifying meaningful communities. Without the use of directed edges, strong unidirectional relationships would falsely be perceived as reciprocal collaborative relationships, which obscure the community structure by creating an overly dense network. In Section 4, we demonstrate the impact of graph density on community structure, and we show that a file-based approach leads to dense developer networks.

3.1.2 Committer–author-based method

Our second method is inspired by earlier work on using tags on commit messages to build developer networks [6]. The tagging convention was originally devised to record a trail of responsibility for code in the final release in the event of litigation [34]. Tag information can be exploited to identify relationships between code authors and all other people involved in the software-development process, including reviewers and testers. A tag-based network contains important information on the software-development process, workflow, and developers with related interests and knowledge [6]. Developers actively tag any piece of code that they are responsible for and have collaborated on. As a result of the self-reflective process, tag-based networks undoubtedly capture real-world collaboration. Unfortunately, only a small number of projects currently use the tag convention. To overcome this limitation, we need another way to obtain this information.

The tag-based network captures the relationship between developers who author the code and the other side of the software-development process that handles preparation of the code to be integrated. On this basis, our solution to substitute the tag information 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 again important since relationships of this type are not necessarily reciprocal. We assign weights to the edges as a function of commit frequency. For example, if author A and committer B are associated with two commits, then their edge weight is two. In Section 4, we show that the

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

committer–author network of Linux is able to capture the same information as the corresponding tag-based network, which is luckily available for Linux. The tag-based network represents the factual real-world collaborative structures, and we use it to validate the structure of the automatically constructed committer–author-based network.

3.2 Network Analysis

Representing developer collaboration as a graph enables the application of network-analysis methods and provides a mathematically rigourous approach to deduce insights about developer coordination.

3.2.1 Community Detection

The basis of our approach of community detection is built on the idea that groups of developers have stronger intra-group interactions than inter-group interactions. The communities represent the important functional groups of developers with closely related goals, area of expertise, and interests. We expect a small number of developers to cross community boundaries and represent the important ”weak ties” [22]. Without these cross-community relationships, technical knowledge would not be able to propagate through the network. We consider the developers that are split between multiple communities as an integral part of the overall coordination and seek to uncover these individuals.

Recent advances in community detection [2, 28, 16] have made it possible to decompose an arbitrary network into communities. However, many community-detection algorithms are unable to handle weighted and directed graphs, and many more are unable to discover overlapping communities. In the case of developer networks, we expect important developers to lie at the boundary between two or more communities, and identifying these developers is of primary concern. If overlapping communities are not permitted, a developer will be incorrectly forced to exist in one community, preventing one from identifying the ”weak ties” that exist between communities [22]. Therefore, we stress the importance of choosing an appropriate community-detection algorithm that supports weighted, directed edges and overlapping communities.

To perform community-detection, we employ the order statistics local optimization method (OSLOM) [28]. OSLOM is one of the few community-detection methods that is able to handle weight and directed networks and is able to form overlapping communities. This approach has not been applied before to developer collaboration networks.

We experimented with several other community-detection algorithms and experienced generally poor performance from basic techniques, such as random-walk or eigenvector based methods [27]. A statistical-mechanics approach using spin-glasses had comparable performance to OSLOM but it does not produces overlapping communities [16].

3.2.2 Community Verification

All community-detection algorithms are essentially unsupervised classifiers and will invariably find communities in any given collaboration graph. The problem is thus to determine whether these are meaningful—an important step that is usually neglected in previous approaches [28]. We determine the validity of a given community by constructing new communities with the same structural properties as the original one, but with randomized preferences for the community members to associate with other members—a standard method in network statistics. If it is possible to detect substantial differences between the original and randomized communities, we can conclude that the community does not originate from a random, meaningless process.

Comparing communities is done via *community quality measures*, of which several have been proposed in the literature. We avoid the commonly used modularity metric in favor of conductance for two reasons [29]. Firstly, modularity is known to suffer from a ”resolution limit”, meaning it is unable to reliably measure small communities [20]. Secondly, modularity is often the optimization criterion used by community-detection algorithms. By using conductance, we avoid the community structure bias introduced by the optimality criterion imposed by the community-detection algorithms. Conductance also allows us to measure the quality of an individual community, where modularity measures the quality of the entire decomposition and does not have a meaningful interpretation for a single community. Furthermore, modularity is known to increase with the number of communities and nodes, making it inappropriate to compare projects of different size [19]. 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 community structures [19]. Formally, conductance $\phi \in [0, 1]$ of a community C that is a sub-graph of a larger collaboration graph G is defined as

$$\phi_G(C) := \frac{|\operatorname{cut}(C, G \setminus C)|}{\min\{\deg(C), \deg(G \setminus C)\}}\,, \tag{1}$$

where cut is the cut-set of a given graph cut, and deg is the total degree of a given graph [1]. Intuitively, ϕ describes the amount of edges crossing the community boundary compared to the amount of edges associated with at least one community member: a completely isolated community has zero conductance and a community with every edge crossing the community boundary has a conductance of one.

To discriminate between identifying meaningful community structures and meaningless random properties 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 $\mathcal{C} = \{C_1, C_2, \ldots, C_i\}$, $C_i \subseteq G$. 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 simulation of an equivalent developer network except with unstructured collaboration. We randomize the original network according to a configuration model using a graph rewiring technique, with which the pairs of edges are randomly selected and the end points swapped such that an edge pair (u, v) and (s, t) is rewired to (u, t) and (s, v) [21]. 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[3] to generate a set $\mathcal{R} = \{R_1, R_2, \ldots, R_m\}$ of randomized 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 needs to main-

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

tain this distribution, that is,

$$P_{R_i}(k) = P_G(k)\ \forall R_i \in \mathcal{R}\,. \tag{2}$$

For each randomized graph R_i, we calculate the overall conductance $q_{R_i}(\mathcal{C})$ and define a probability distribution for the random graph conductance as

$$P_{\mathcal{R}}(Q) = \frac{|\{i \in [1, |\mathcal{R}|] \mid q_{R_i}(\mathcal{C}) = Q\}|}{|\mathcal{R}|}\,. \tag{3}$$

Ascertaining that the collaboration structure is meaningful, that is, it does likely not stem from an unorganized, random process is then a matter of standard hypothesis testing. We want to reject the null hypothesis H_0 that the observed overall conductance $q_G(\mathcal{C})$ can arise from the conductance distribution of the randomized graphs with a non-vanishing probability,

$$H_0 : P_{\mathcal{R}}(Q = q_G(\mathcal{C})) > \epsilon\,, \tag{4}$$

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

After establishing the required normal distribution of $P_{\mathcal{R}}(\cdot)$ using a Shapiro-Wilk test, we perform a standard t-test on H_0 vs. H_1, resulting in a p-value, depending on which we can or cannot reject the null hypothesis.

3.2.3 *Developer Importance*

In networks, the notion of importance is captured by node-centrality measures. Such measures can be calculated via link-analysis methods, such as Google's PageRank algorithm [3].

The well-researched and scalable PageRank algorithm has historically been used in measuring relative importance of web-pages in the WWW [3]. To the best of our knowledge, we are the first who apply PageRank to developer networks.

Applying PageRank analysis to developer networks, we are able to visualize the distribution of developer influence for a given project. Our primary goal with applying PageRank analysis is to determine if the structure of a developer network accurately reflect real-world collaboration. PageRank values are based on the global network structure. Therefore, we conjecture that, if important nodes in the developer networks agree with reality, then the global graph structure should also accurately reflect real-world collaboration.

4. EVALUATION & RESULTS

We now present our findings on the network properties and community structure of developer networks for 10 open-source projects. To confirm or refute our hypotheses, we evaluate developer networks constructed using the state-of-the-art file-based method and the methods we propose.

4.1 Hypotheses

Most of the previous work on mining VCSs for developer networks has not focused on identifying evidence for authentically measuring real-world developer collaboration [35]. To test the authenticity of our methods for constructing developer networks, we examine a well-established characteristic of software development that we expect to see manifestations of, and thus far no method has been able to capture. Software development is an organized process and if a developer network faithfully captures real-world developer collaboration, it should also exhibit an organized structure.

Hypothesis 1—*Developer networks exhibit identifiable community structures that significantly exceed the level of organization expected from an unorganized process.*

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

Furthermore, the state-of-the-art method to construct developer networks relies on file-level information to identify collaborating developers. We will show that this approach is unable to capture subtle features that lead to community structure as a result of over-connecting developers in the network. Dense networks are known to negatively impact the performance of graph-clustering algorithms [8], and this problem is relevant for file-based developer networks [26]. In Section 3, we proposed a fine-grained approach that uses source-code structural information to avoid over-connecting the network.

Hypothesis 2—*Developer networks constructed using the state-of-the-art file-based approach fail to identify statistically meaningful community structures, whereas a more fine-grained approach is able to identify statistically meaningful community structures.*

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. A developer network constructed on the basis of commit tags can be regarded as faithfully representing real-world collaboration. We can then use congruence between the ground truth tag-based network and our automatically-constructed committer–author network to effectively achieve 100% survey response rate. Since all developers would already agree they are in collaboration with anyone they tag a common commit with the commit tags function as the survey for which developers collaborated on any piece of code.

Hypothesis 3—*Tag-based developer networks constructed from the manual process of tagging commits are highly congruent with automatically determined committer–author-based networks.*

4.2 Subject Projects

We selected 10 open source projects to evaluate our methods. We selected the projects to cover a broad range of project characteristics to most effectively demonstrate the broad applicability of our approach. The projects are listed in Table 1. All projects are relevant active projects with substantial data available. We selected open-source projects, so that other researchers can compare their methods to ours.

The projects vary by the following dimensions: (a) size (source lines of code from 50 KLOC to over 16 MLOC, number of developers from 15 to 1000), (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). Our subject project set covers all major points in the resulting multi-dimensional space, suggesting a broad applicability of our method.

For each project, we chose all relevant VCS data for one major release, covering a full development cycle. This ensures that all relevant software-development activities have been carried out, at least, once in appropriate detail and hence are reflected in the data. All releases occurred at about the same time.

Table 1: Subject projects

Project	Devs	KLOC	Languages	Domain
Linux	580	16 000	C	OS
Chromium	500	6 500	C, C++	User
Firefox	400	9 300	C++, JavaScript	User
GCC	70	6 200	C, C++	Devel
QEMU	50	780	C	OS
PHP	50	2 200	PHP, C	Devel
Joomla	30	1 300	PHP, JavaScript	Devel
Perl	30	4 500	Perl, C	Devel
Apache HTTP	15	2 200	C	Server
jQuery	15	50	JavaScript	Library

4.3 Network Visualization

As a by-product of network construction and analysis, decomposing the developer networks into community structures allows us to comprehensively visualize the complex interrelationships of technical collaboration in large-scale software projects. Figure 1 illustrates how knowledge of community structure can be visualized. 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 fractional participation in a community. The box background color is used to represent the strength of each community, calculated according to conductance (cf. Section 3.2.2). Green signifies a strong community, yellow a weak community, and red an anti-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. PageRank centrality was used to identify important developers and is represented by the size of each node. We filter the inter-community-edges to reduce clutter 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.

Figure 1 presents an example of a developer network visualization for QEMU v1.0 using our approach. The visualization was generated automatically from the developer network and required no manual involvement. Some possible conclusions one can draw from the visualization of QEMU is that it comprises four main communities, all of which are strong except for a single weak three developer community. All communities strongly collaborate with community ID 1 and are loosely coupled to each other. Community ID 1 likely plays a large role in coordinating the efforts of the entire developer group and thus has a high degree of influence. Most communities have, a least, a single developer, who is seen as quite important globally, and likely symbolizes a team leader. Team leaders often have considerable participation in other communities from their own. This kind of behavior is indicative of a development style where most inter-team integration work is mediated through the team lead. The less prominent developers (*i.e.*, smaller node size) are more often seen to make contributions only within their own group seen by mostly solid color nodes.

4.4 Existence of Community Structure

To test Hypothesis 1, we applied our function-based method

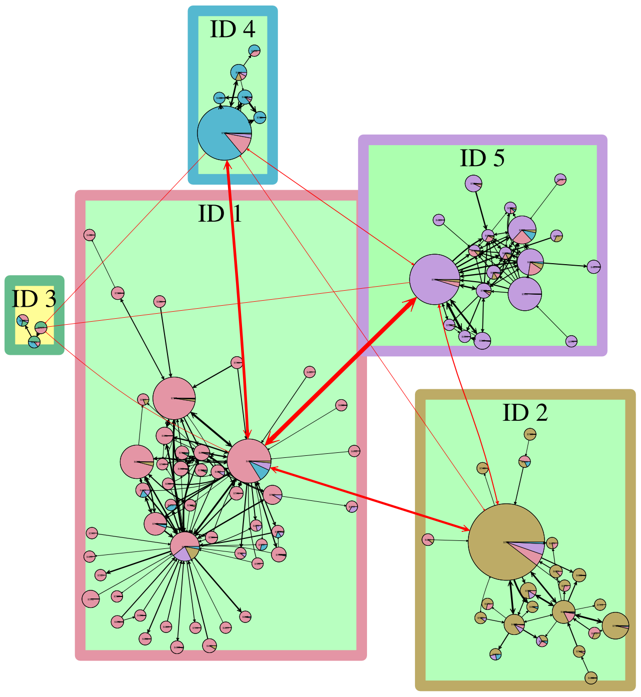


Figure 1: The community decomposition of a developer network constructed using our function-based method from QEMU v1.0. The color of a node represents the community membership, pie chart style icons indicate a node having membership in multiple communities. The shaded domain, on which each community lies, indicates the significance of the community as per the conductance metric, green = strong community, yellow = weak.

to construct developer networks for all subject projects. We expected meaningful community structure to exist as a result of an organized software-development process, in at least, some of the subject projects, and we now verify whether our method is able to identify the community structure, and we evaluate it using the community-verification procedure described in Section 3.2.2.

We were able to identify a high degree of community structure for all subjects except jQuery. Figure 2 illustrates the results for Linux. The distribution of conductance values for the randomized developer network is show in black, and the conductance value for the non-randomized developer network is shown as a green point. A low conductance value indicates high community structure. In Figure 2, we see that the conductance value of the non-randomized network does not fall within the distribution of conductance values for the randomized network. Therefore, we conclude that the community structure seen in the developer network is meaningful, because there is an unlikely probability that it could have resulted from a random process. The developers must be influenced in some way to form organized groups that are responsible for particular system's components or functionality. In conclusion, we reject the null hypothesis that the observed developer networks exhibit random levels of community structure, thus *confirming Hypothesis 1*. The complete set of results is summarized in Table 2.

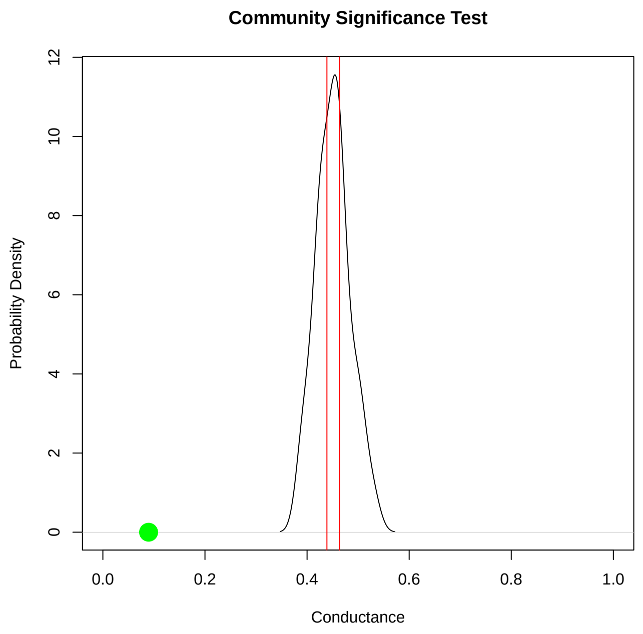


Figure 2: Community-structure verification (see Section 3.2.2) performed on Linux v3.7. The probability distribution for the random graph conductance (black) with confidence intervals (red) show a significant separation from the conductance value (green) of the function-based developer network.

4.5 Comparison with State of the Art

To verify Hypothesis 2, we performed a comparison between the file-based method and our function-based method for constructing developer networks (see Section 3.1.1) The comparison draws attention to the deficiencies of the state-of-the-art approach in identifying important structural characteristics of the developer network.

Figure 3 illustrates the inability of the state-of-the-art approach to identify a statistically meaningful community structure. All communities identified by the state-of-the-art file-based method (left side) are weak. The conductance of the communities is on the order of what is expected from a randomized developer network, indicated by the yellow background color. This result suggests that the method has failed to capture the organized structure of developer collaboration. In contrast, our function-based method (a more fine-grained approach to construct the developer network) is capable of identifying several strong communities in the same project. Furthermore, the state-of-the-art file-based method has constructed an over-connected network, in which nearly every developer is contributing in every community, visible by the large number of multicolored nodes. In contrast, the developer network constructed using our method has identified developers who most often make contribution only within one or two communities, visible by the large number of single-colored nodes.

To compare the function-based and file-based methods further, we constructed developer networks using both methods and evaluated the community structure using conductance. The scatter plot in Figure 4 shows each project as a single data point. The location of the data point depends on the mean value of the community conductance and the size across all communities identified in the project. The scatter plot reveals a distinct separation between the state-of-the-art and the function-based method. The state-of-the-art method places most projects in the upper and right portions of the plot, suggesting that it identifies less meaningful and larger communities. Our function-based method places most projects in the lower left corner of the plot, where communities are both significant and relatively compact.

Table 2: Community Conductance

Project	Release Range	Original	Random
Linux	3.5.0–3.6.0	0.05	0.88
Chromium	27.0.1–28.0.1	0.20	0.74
jQuery	1.7.0–1.8.0	0.49	0.75
Joomla	3.0.0–3.1.0	0.57	0.84
GCC	4.8.0–4.8.1	0.01	0.48
PHP	5.3.0–5.4.0	0.15	0.80
QEMU	1.3.0–1.4.0	0.12	0.65
Apache HTTP	2.2.0–2.3.0	0.27	0.80
Perl	5.17.0–5.18.0	0.49	0.66
Firefox	18.0.0–19.0.0	0.11	0.79

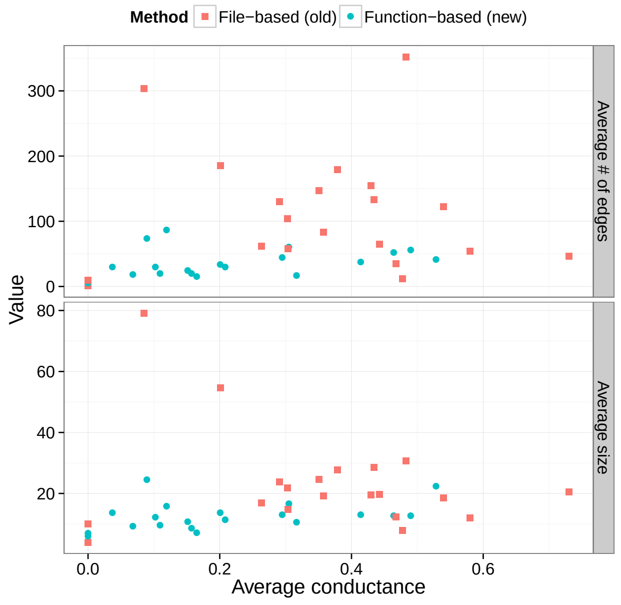
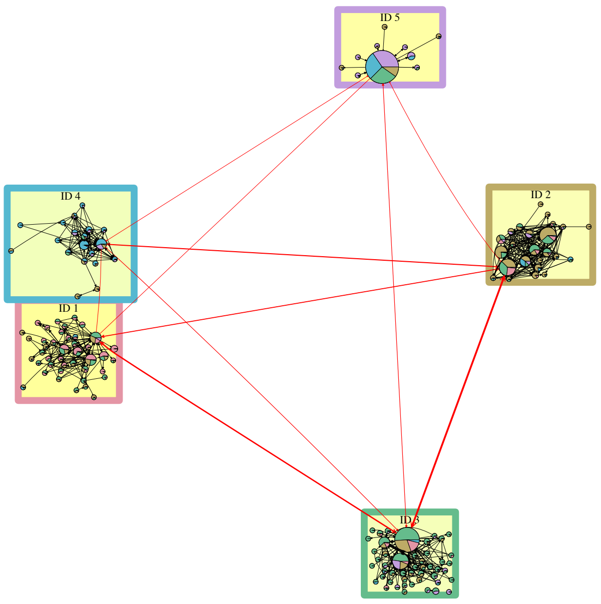
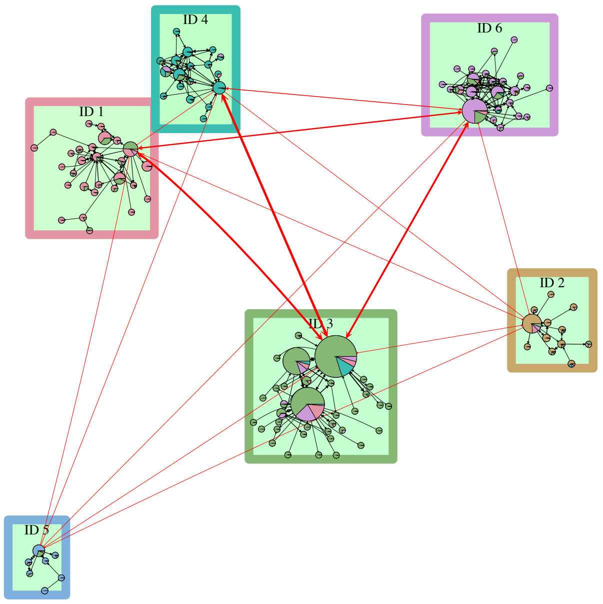


Figure 4: Scatter plot of all projects analyzed with the file-based approach and the function-based approach. Each data point represents a project. An approximate, but distinct clustering is visible between the two analysis methods resulting from the ability of the function-based approach to resolve small and significant communities.

The density plot of mean community conductance across all analyzed projects in shown in Figure 5. The expected approximate Gaussian distribution is visible in both methods, and there is also a clear separation between the distributions.



(a) File-based developer network



(b) Function-based developer network

Figure 3: Developer networks constructed from QEMU v1.1. All communities in the file-based network are weak, as indicated by the yellow background color. The function-based method identified several strong communities indicated by the green background color.

The function-based method identifies a significantly stronger community structure in comparison to the state-of-the-art method. We performed a two-sided Wilcoxon test and verified that the difference in the distributions is statistically significant.

In summary, we conclude that the state-of-the-art method fails to identify meaningful community structure, thus justifying the usefulness of a more fined-grained approach that considers source-code structure to identify developer collaboration. We therefore *confirm Hypothesis 2*.

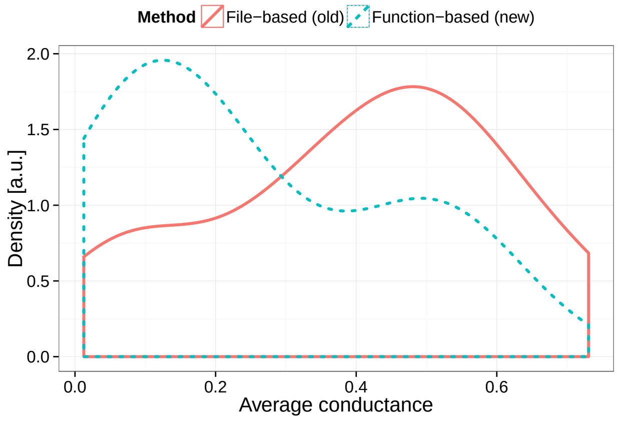


Figure 5: Density plot of mean community conductance computed for each project. Each project was analyzed using the file-based approach and the function-based approach.

Another informal, but nonetheless important validation method for the quality of the detected communities would be a survey among key contributors of the projects under study. While this is well beyond the scope of this paper, we have used our expert knowledge on the Linux kernel [34] to inspect the meaning of the derived clusters. While the results of this are hard to formalize, let us remark that most communities can be immediately associated with a particular line of development given the names of community members and their inferred interrelations. As an example, consider Figure 6, which shows the developer community for the in-kernel hypervisor KVM, with interpretation details given in the caption.

For the same reason as above, we cannot provide a formal evaluation of the effectiveness of PageRank in this paper. However, using expert knowledge on the Linux kernel led to a confirmation of our calculations (details are given on the supplementary website). Using simpler, established techniques like measuring the amount of contributed code to evaluate developer influence turned out to be an inferior approach: For instance, technical media widely reported that Microsoft had advanced to a leading Linux contributor during development cycle 3.0, as the company had contributed substantial amount of code to the virtualization subsystem.[4]. Our method, however, categorized the contributions as quite un-influential, again without any a-priori knowledge. This is a much better reflection of reality since the added functionality was small and the influence on the overall kernel very restricted.

[4]See, for instance, the article `http://www.zdnet.com/blog/open-source/top-five-linux-contributor-microsoft/9254`

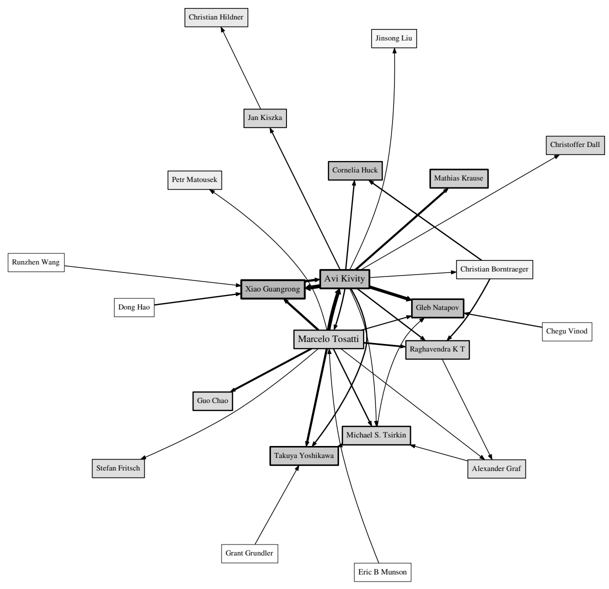


Figure 6: The community responsible for the development of the in-kernel KVM hypervisor, derived from the Linux cycle 3.7. An edge from A to B indicates that A has tagged B's work. Darkness of the node background increases with the amount of changed lines; the frame width scales with increasing number of commits.

Kivity and Tossati jointly maintain the sub-project, which is directly visible by their central placement in the graph. Guangrong, Natapov, Huck and Tsirkin are regular strong individual contributors close to the maintainer center, while authors with less importance (at least in the cycle under consideration—other cycles attribute a stronger role to Graf and Kiszka) are placed in increasing distance to the core contributors. Albeit the decomposition method has no a-priori knowledge about social aspects of Linux kernel development, the derived results paints an accurate and satisfactory picture.

4.6 Tag-Based and Committer–Author-Based Network Similarity

To verify Hypothesis 3, we constructed developer networks for three revisions of Linux kernel development using the tag-based and committer–author-based methods. We chose Linux as a test subject because it is well known, has undergone several major revisions, has around one-thousand developers per release, and enforces a strict tagging convention for every commit (which the other subject projects do not do). For each revision, we compare the tag-based and committer–author-based networks using a graph matching strategy based on vertex similarity, by computing the Jaccard index for each common vertex [19]. Figure 7 shows the results as a density plot. We see bimodal distribution with peaks at 100% similarity (perfect match) and 50% similarity. The average taken over the three revisions is 70% agreement between the two networks. The probability of have a 70% match over a graph of around 1000 nodes and 4000 edges only by coincidence is very unlikely. We conclude that the high degree of congruence is a result of both graphs arising from the same underlying process and *confirm Hypothsis 3*.

As a side remark, we consider the bimodal distribution to result from the fact that most commits in the Linux kernel involving either two or three developers. In the case of two developers, it is often the author and commuter who acknowledge their contribution with a tag on the commit. In this case, the committer–author network captures the collaboration entirely resulting in one maxima at 100% agreement. In the case of three developers involved in a commit, normally the committer, author, and a reviewer or tester tags the commit. In this case the committer–author network misses the involvement of the additional person resulting in the second maximum at 50% agreement.

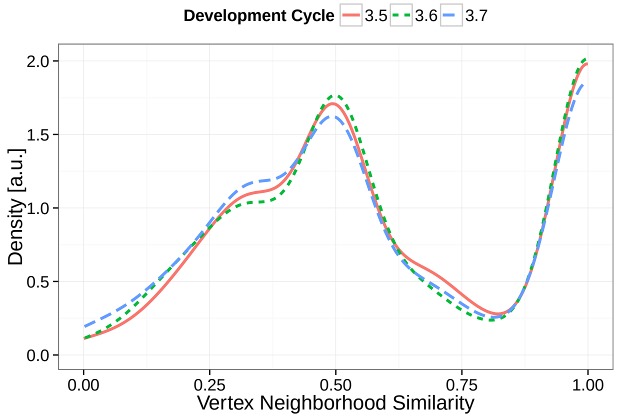


Figure 7: Comparison between tag-based and committer–author-based networks. Each curve represents the network comparison based on a single revision of Linux. The difference between each developers neighborhood is shown as a density plot. We see that there is a good match, with 70% agreement.

5. THREATS TO VALIDITY

The collaborative activities in software development are numerous and diverse, and many of them are not captured directly in VCSs. For this reason, research has questioned whether developer networks constructed from VCSs are capable of capturing real-world collaborations. Interestingly, recent results provide evidence that this is really the case [35], and we base our work on these results. We conjecture that a finer-grained definition of collaboration, as in our function-based approach, does not change this picture. Additionally, the validity of the committer–author-based network is established through congruence with the tag-based network, which is regarded as representing real-world collaboration.

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.

Furthermore, we selected the 10 subject projects not randomly, which threatens external validity. Instead, we aimed at projects of different but substantial sizes, with long and active development histories, and from different application domains. Furthermore, we used Git as a technological basis (with one exception, which did not change the picture). Although one can never safely generalize from such a sample, we are confident that the conclusions drawn from our evaluation demonstrate the potential of our approach and motivate replication studies. Even if our results applies only to Git repositories or open-source projects, we still cover a broad spectrum of development projects.

Finally, analyzing the causal relationship between development activities and community structure is outside the scope of this work and requires a major effort in further work. So, at this point, it is unknown whether organized software development causes communities to form, or imposing community structure leads to organized development. Either way, identifying such a casual relation would be very interesting. Our work can serve as a basis to address this issue in further work.

6. RELATED WORK

Networks constructed from e-mail archives of open-source projects have been extensively studied [4, 41]. In these networks, developers are represented as nodes, and edges are placed between people who have exchanged e-mail. Bird et al. [5] investigated developer organization and community structure in 4 open-source projects. They used modularity as the community significance measure to prove the existence of statistically significant communities. Our work differs by examining technical collaboration networks from source-code contributions instead of communication networks based on e-mail. Additionally, we improve on their community-detection method by identifying overlapping communities, and we use conductance as a more appropriate community metric (see Section 3.2.2).

Extracting technical collaborative data from the VCS in the form of a graph structure was first done by Lopez-Fernandez et al. [31, 32]. Two developers were assumed to collaborate when they both made a commit to the same software module. Software modules were classified manually using expert knowledge of the project, and they have shown that committer networks exhibited small-world properties using basic social network measures. Our method differs in that it is fully automated, it focuses on extracting and identifying fine-grained collaborations, and it identifies statistically meaningful communities.

Huang et al. improved Fernandez's work by automating module classification using knowledge of file directories [25]. A module was defined as a top level directory in the repository. The authors expressed that, using modules as the artifact to identify, collaboration may not provide detailed enough information to be useful. Advancements followed by narrowing the collaboration assumption to files rather than modules to identify more fine-grained collaboration [26]. Under their assumption, an edge would appear between two developers if both made a commit to the same file. While narrowing the collaboration assumption did help to identify more subtle features than the module based assumption, the authors noted that the networks were too dense to identify community structures. In our approach, we use an even finer-grained definition of collaboration, to reduce the density of the network and uncover the finer community structures. Previous work mostly applied metrics that do not produce meaningful visualizations, such as degree distributions or centrality plots, and no one has visualized community structure [32, 31, 25, 41, 35]. We are aware of one paper focused on visualization of developer collaboration using a file-based approach, however, community-detection was not possible without edge filtering due to the extreme density of the developer networks [26].

The tagging convention in Linux development has been utilized to construct a developer network consisting of people involved in reviewing, acknowledging and testing code, however, the network properties have not been analyzed [6]. We extended on 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 VCS actually captured the perceived real-world collaboration [35]. Developer networks have been extracted at the file level followed by a survey on whether two people connected in the developer network constructed from a VCS also perceived themselves as collaborating. The survey was performed for Linux and other projects. They found that, while certain connections in the network falsely suggested collaboration and certain collaboration was missing from the network, the developer network could be largely interpreted to reflect developer perceptions. We used the result of this work to propose that collaboration at the function level captures more authentic relationships between developers than at file level, by eliminating collaboration artifacts introduced by an overly coarse-grain collaboration assumption.

7. CONCLUSION

Being able to capture the complex collaborative relationships in large software projects is a valuable asset to project management, developer productivity, and software quality. Despite considerable advances, state-of-the-art methods fail to recognize the fine-grained organizational structures and prominent structural features that differentiate one software project from another, or they require manual effort to document the responsibilities of code changes.

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

We evaluated our approach on 10 popular 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 community structures, and we are able to identify and visualize them automatically using our approach, which is not possible using state-of-the-art methods. Furthermore, using expert knowledge, we were able to confirm that the automatically constructed developer networks for Linux indeed resemble real-world collaboration.

In further work, we shall evaluate the appropriateness of the automatically derived developer networks on further projects. Furthermore, we shall evaluate and include further sources of information to recover, represent, and analyze developer collaboration.

8. 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] M. Bianchini, M. Gori, and F. Scarselli. Inside pagerank. *ACM Transactions on Internet Technology (TOIT)*, 5(1):92–128, 2005.

[4] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 137–143. ACM, 2006.

[5] 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.

[6] 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.

[7] J. Bosch and P. Bosch-Sijtsema. From integration to composition: On the impact of software product lines, global development and ecosystems. *J. Syst. Softw.*, 83(1):67–76, 2010.

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

[9] F. P. Brooks. *The mythical man-month*, volume 1995. Addison-Wesley Reading, 1975.

[10] 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.

[11] 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.

[12] L. Colfer and C. Baldwin. The mirroring hypothesis: Theory, evidence and exceptions. *Harvard Business School Finance Working Paper*, (10-058), 2010.

[13] M. E. Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[14] 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.

[15] 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.

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

[17] C. Ebert and C. Jones. Embedded Software: Facts, Figures, and Future. *Computer*, 42(4):42–52, 2009.

[18] 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.

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

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

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

[22] M. S. Granovetter. The strength of weak ties. *American journal of sociology*, pages 1360–1380, 1973.

[23] 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.

[24] J. D. Herbsleb and D. Moitra. Global software development. *Software, IEEE*, 18(2):16–20, 2001.

[25] 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.

[26] 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.

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

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

[29] 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.

[30] P. Liggesmeyer and M. Trapp. Trends in Embedded Software Engineering. *IEEE Softw.*, 26(3):19–25, 2009.

[31] 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.

[32] 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.

[33] 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.

[34] W. Mauerer. *Professional Linux Kernel Architecture.* Wiley, 2008.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.