Graph-Based Analysis and Prediction for Software Evolution

Pamela Bhattacharya Marios Iliofotou Iulian Neamtiu Michalis Faloutsos
Department of Computer Science and Engineering
University of California, Riverside, CA, USA
{*pamelab,marios,neamtiu,michalis*}*@cs.ucr.edu*

***Abstract*—We exploit recent advances in analysis of graph topology to better understand software evolution, and to construct predictors that facilitate software development and maintenance. Managing an evolving, collaborative software system is a complex and expensive process, which still cannot ensure software reliability. Emerging techniques in graph mining have revolutionized the modeling of many complex systems and processes. We show how we can use a graph-based characterization of a software system to capture its evolution and facilitate development, by helping us estimate bug severity, prioritize refactoring efforts, and predict defect-prone releases. Our work consists of three main thrusts. First, we construct graphs that capture software structure at two different levels: (a) the product, i.e., source code and module level, and (b) the process, i.e., developer collaboration level. We identify a set of graph metrics that capture interesting properties of these graphs. Second, we study the evolution of eleven open source programs, including Firefox, Eclipse, MySQL, over the lifespan of the programs, typically a decade or more. Third, we show how our graph metrics can be used to construct predictors for bug severity, high-maintenance software parts, and failure-prone releases. Our work strongly suggests that using graph topology analysis concepts can open many actionable avenues in software engineering research and practice.**

***Keywords*-Graph science; software evolution; software quality; defect prediction; productivity metrics; empirical studies**

I. Introduction

Improving software maintenance and development is an involved and costly task, with direct financial impact. According to Gartner, global software expenditures for 2010 amounted to $229 billion [1], with large vendors such as Microsoft and IBM reporting multi-billion dollar costs for software development each year [2], [3]. A large part of development costs—an estimated 50 to 90 percent of total costs—is due to software evolution [4], [5], [6], [7]. Despite these high costs, software is notoriously unreliable, and software bugs can wreak havoc on software producers and consumers alike—a NIST survey estimates the annual cost of software bugs to be about $59.5 billion [8]. At the same time, understanding and constructing rigorous software evolution models remains a significant research challenge [9].

Recently, graph-based analysis of complex systems has experienced a resurgence, under the name of Network Science (or mining of graph topology). There is a good reason for this: topology analysis of graphs has revolutionized the modeling and analysis of complex systems in many disciplines and practical problems. For example, graph-based methods have opened new capabilities in classifying network traffic [10], [11], modeling the topology of networks and the Web [12], [13], and understanding biological systems [14], [13]. What these approaches have in common is the creation of graph-based models to represent communication patterns, topology or relationships. Given a graph model, one can unleash a variety of techniques to discover patterns and communities, detect abnormalities and outliers, or predict trends.

The overarching goal of this work is to find whether graph-based methods facilitate software engineering tasks. Specifically, we use two fundamental questions to drive our work:

(a) How can we improve maintenance by identifying which components to debug, test, or refactor first?, and

(b) Can we predict the defect count of an upcoming software release?

Note that our intention is not to find the best possible method for each question, but to examine if a graph-based method can help, through the use of an appropriately-constructed graph model of the software system. While we use these two indicative questions here, we believe there could be other questions that can be addressed with graph-based approaches.

Our thesis is that graph-based approaches can help to better understand software evolution, and to construct predictors that facilitate development and maintenance. To substantiate, we show how we can create graph-based models that capture important properties of an evolving software system. We analyze eleven open-source software programs, including Firefox, Eclipse, MySQL, Samba, over their documented lifespans, typically a decade or more. Our results show that our graph metrics can detect significant structural changes, and can help us estimate bug severity, prioritize debugging efforts, and predict defect-prone releases.

Our contributions can be grouped in three thrusts.

a. Topological analysis of software-based graphs can reveal properties about software process. We propose the use of graphs to model software at two different levels and for each level, we propose two different granularities.

At the software *product* level, we model the software structure, at the granularity of functions (function-level interaction) and modules (module-level interaction).

At the software *process* level, we model the interactions between developers when fixing bugs and adding new features. We use two construction methods: the *bug-based developer collaboration*, which captures how a bug-fix is passed among developers, and *commit-based developer collaboration* which represents how many developers collaborated in events other than bug fixes, by analyzing the commit logs.

b. Graph metrics capture significant events in the software lifecycle. We study the evolution of the graph models of these programs over one to two decades. We find that these graphs exhibit some significant structural differences, and some fundamental similarities. Specifically, some graph metrics vary significantly among different programs, while other metrics captures persistent behaviors and reveal major evolutionary events across all the examined programs. For example, our graph metrics have revealed major changes in software architecture in mid-stream releases (not ending in ".0"): OpenSSH 3.7, VLC-0.8.2 and Firefox 1.5 show big changes in graph metrics which, upon inspection, indicate architectural changes that trump changes observed in ".0" versions of those programs. Similarly, our *edit distance* metric has detected a major change in Samba's code structure in release 1.9.00 (Jan 22, 1995), due to major bug fixes and feature additions; the change is not apparent when examining other metrics such as eLOC.

c. Our graph metrics can be used to predict bug severity, maintenance effort and defect-prone releases. The cornerstone of our work is that our graph metrics and models can be used to suggest, infer, and predict important software engineering aspects. Apart from helping researchers construct predictors and evolution models, our findings can help practitioners in tasks such as: identifying the most important functions or modules, prioritizing bug fixes, estimating maintenance effort:

1. We show how *NodeRank*, a graph metric akin to PageRank, can predict bug severity.

2. We show how the *Modularity Ratio* metric can predict which modules will incur high maintenance effort.

3. We demonstrate that by analyzing the *edit distance* in the developer collaboration graphs we can predict failure-prone releases.

While these predictors might seem intuitive, we are the first to quantify the magnitude and lag of the predictors.

II. Graph Construction

We describe the methodology for graph construction, data collection, and computing graph metric values. An overview of our system is presented in Figure 1. We construct graphs from two main sources: the source code repository and the bug tracking system. [1] From the code repository, we extract commit logs, historic source code versions, patches, and source code-based developer interaction. From the bug tracker, we extract bug histories and bug fixing-based developer interaction. All these artifacts are related, and the relationships are captured using graph edges. Thus, a series of graphs emerges. The key observation of our approach is that information derived from these graphs can be used to understand how software evolves, and how to construct effective predictors for software engineering metrics, such as bug severity, and maintenance effort.

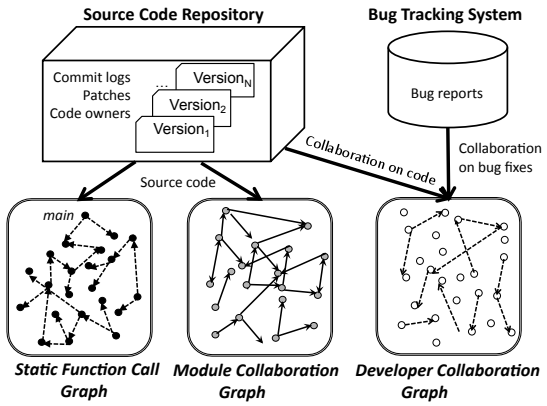


Figure 1. System overview.

A. Source Code-based Graphs

We use the source code to construct graphs at two abstraction levels: function (*call graph*) and module (*module collaboration graph*). To construct these graphs, we extended CodeViz [15], a static analysis tool, to extract function call information and global variable usage.

Call graph: The function call graph captures the static caller-callee relationship. If function A calls function B, the function call graph contains two nodes, A and B, and a directed edge from node A to node B. Our data set contains several applications written in a combination of C and C++; for virtual C++ methods, we add edges to soundly account for dynamic dispatch. Function call graphs are essential in program understanding and have been shown effective for recovering software architecture for large programs [16].

Module collaboration graph: This graph captures communication between modules, and is coarser-grained than the function call graph. We construct the module collaboration graph as follows: if a function in module A calls a function in module B, the graph contains a directed edge from A to B. Similarly, if a function in module A accesses a variable defined in module B, we add an edge from A to B. Module collaboration graphs help us understand how software components communicate.

B. Developer Collaboration Graphs

We build developer collaboration graphs to analyze how developers communicate as software evolves. We build two kinds of graphs, as described below:

[1] The graph datasets are available online at http://www.cs.ucr.edu/~neamtiu/graph-data-icse12

Application	Time span	Releases	Language	Size (kLOC)		Use
				first release	last release	
Firefox	1998-2010	92	C,C++	1,976	3,780	↗, 1, 2, 3
Blender	2001-2009	28	C,C++	253	1,144	↗, 1, 2
VLC	1998-2009	83	C,C++	144	293	↗, 1, 2
MySQL	2000-2009	13	C,C++	815	991	↗, 1, 2
Samba	1993-2009	78	C	5	1,045	↗, 1
Bind	2000-2009	171	C	169	321	↗
Sendmail	1993-2009	55	C	25	87	↗
OpenSSH	1999-2009	77	C	12	52	↗, 1
SQLite	2000-2009	169	C	17	65	↗
Vsftpd	2001-2009	59	C	6	15	↗
Eclipse	2001-2010	27	Java	828	1,903	3

Table I
APPLICATIONS' EVOLUTION SPAN, NUMBER OF RELEASES, PROGRAMMING LANGUAGE, SIZE OF FIRST AND LAST RELEASES.

Bug-based developer collaboration: To build the *Bug-Developer Collaboration graphs*, we use the bug tossing graphs we constructed in previous work [17]. When a bug is assigned to developer D_1 and he/she is unable to resolve it, the bug is reassigned to developer D_2 and we add a directed edge from D_1 to D_2 in our graph.

Commit-based developer collaboration: The second kind of developer collaboration graph, which we term *Effort-Developer Collaboration graphs*, traces how developers have collaborated in events other than bug fixes, by analyzing commit logs. We add an undirected edge between developers D_1 and D_2 if they have worked on the same file.

C. Applications

We base our study on eleven popular open source applications written mainly in C, or combinations of C and C++. We select applications that have: (a) long release history, (b) significant size (in lines of code and modules), (c) a large set of developers who maintain them, (d) a large user base, who report bugs and submit patches. The above criteria are necessary for making meaningful statistical, behavioral, and evolutionary observations.

Table I lists the wide-range of applications involved in our study, along with some key properties. *Firefox* is the popular web browser from the Mozilla suite. *Blender* is a 3D content creation suite. *VLC* is a cross-platform multimedia framework, player and server. *MySQL* is a relational DBMS. *Samba* is a tool suite that facilitates Windows-UNIX interoperability. *Sendmail* is the leading email transfer agent today. *BIND* is the leading DNS server on the Internet. *OpenSSH* is the standard open source suite implementing secure shell protocols. *SQLite* is a popular library implementation of an SQL database engine. *Vsftpd* is the FTP server in major Unix distributions. *Eclipse* is a popular IDE.

The second column shows the time span we consider for each application, the third column contains the number of official releases within that time span; we analyzed all these releases. Column 4 shows the main language(s) the application was written in; some of the applications have small parts written in other languages, e.g., JavaScript. Columns 5 and 6 show application size, in effective LOC, for the first and last releases. The last column shows the studies and hypotheses we use each program for: ↗ means we have used that application in the evolution study (Section IV); the numbers indicate whether we have used that application when testing hypotheses *H1*, *H2*, or *H3* (Sections V, VI, and VII, respectively).

The long time spans we consider—Samba has grown by a factor of 200x over 16 years— allow us to analyze evolution rigorously, obtain statistically significant results, and observe a variety of change patterns in the graphs.

For each application, we have used its website to obtain the source code of official releases. We used applications' version control systems for extracting file change histories and patches. Finally, we extracted bug information from application-specific bug tracking systems.

III. METRICS

We introduce the graph metrics, and the software engineering concepts, which we will use in our work.

A. Graph Metrics

For each metric, we indicate if it is calculated on a directed and undirected graph. Our graphs are initially directed, but can be trivially transformed into undirected graphs, by ignoring the directivity of the edges.

Average degree (directed graph): In a graph $G(V, E)$, V denotes the set of nodes and E denotes the set of edges. The average degree is defined as: $\bar{k} = \frac{2|E|}{|V|}$

Clustering coefficient (undirected graph): The clustering coefficient $C(u)$ of a node u captures the local connectivity, or the probability that two neighbors of u are also connected. It is defined as the ratio of the number of existing edges between all neighbors of u over the maximum possible number of such edges. Let $|\{e_{jk}\}|$ be the number of edges between u's neighbors and k_u be the number of u's neighbors. Then, we have:

$$C(u) = \frac{2|\{e_{jk}\}|}{k_u(k_u - 1)}$$

The metric is meaningful for nodes with at least two neighbors. A graph's *average clustering coefficient* is the average clustering coefficient over all the nodes.

NodeRank (directed graph): We define a measure called *NodeRank* that assigns a numerical weight to each node in a graph, to measure the relative importance of that node in the software—this rank is similar to PageRank [18], which represents the stationary distribution of the graph interpreted as a Markov chain. There are several ways for defining and calculating the PageRank. Here, we use the following recursive calculation. For a node u, let $NR(u)$ be its *NodeRank*, and let the set IN_u contains all the nodes v that have an outgoing edge to u. We assign equal NoderRank

values to all nodes initially. In every iteration, the new $NR(u)$ is the sum over all $v \in IN_u$:

$$NR(u) = \sum_{v \in IN_u} \frac{NR(v)}{OutDegree(v)}$$

We stop the iteration when the NodeRank values converge. Note that to enable convergence, at the end of every iteration, we normalize all values so that their sum is equal to one. Intuitively, the higher the *NodeRank* of a vertex u, the more important u is for the program, because many other modules or functions depend on it (i.e., call it). Similarly, in the developer collaboration graph, a developer D with a high $NR(D)$ signifies a reputable developer.

Graph diameter (undirected graph): is the longest shortest path between any two vertices in the graph.

Assortativity (undirected graph): The assortativity coefficient is a correlation coefficient between the degrees of nodes on two ends of an edge; it quantifies the preference of a network's nodes to connect with nodes that are similar or different, as follows. A positive assortativity coefficient indicates that nodes tend to link to other nodes with the same or similar degree. Assortativity has been extensively used in other Network Science studies. For instance, in social networks, highly connected nodes tend to be connected with other high degree nodes [19]. On the other hand, biological networks typically show disassortativity, as high degree nodes tend to attach to low degree nodes [20].

Edit distance (directed graph): The metrics we described so far characterize a single program release. To find out how program structure changes over time, we introduce a metric that captures the number of changes in vertices and edges between two graphs, in our case between successive releases. The *edit distance* $ED(G_1, G_2)$ between two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ is defined as $ED(G_1, G_2) = |V_1| + |V_2| - 2 * |V_1 \cap V_2| + |E_1| + |E_2| - 2 * |E_1 \cap E_2|$.

Intuitively, if G_1 and G_2 model software structures for releases 1 and 2, then high values of $ED(G_1, G_2)$ indicate large-scale structural changes between releases.

Modularity ratio (directed graph): Standard software engineering practice suggests that software design exhibiting high cohesion and low coupling provides a host of benefits, as it makes software easy to understand, easy to test, and easy to evolve [6]. Therefore, we define the *modularity ratio* of a module A as the ratio between its cohesion and its coupling values: $ModularityRatio(A) = \frac{Cohesion(A)}{Coupling(A)}$ where $Cohesion(A)$ is the total number of intra-module calls or variable references in A; $Coupling(A)$ is the total number of inter-module calls or variable references in A.

B. Defects and Effort

Defect density: We use defect density to assess external application quality. To ensure accuracy, we extract (and cross-check) information from bug databases and bug information extracted from change logs. With the bug information at hand, we then associate a certain bug to a certain version: we use release tags, dates the bug was reported, and commit messages to find the version in which the bug was reported in, and we attributed the bug to the previous release.

Effort: To measure development and maintenance effort, we counted the number of commits and the churned eLOC (sum of the added and changed lines of code) for each file for a release, similar to previous work by other researchers [21], [22]. This information is available from the log files. The defect density and effort computations are not a contribution of this paper—we extracted this information in prior work [23]. Nevertheless, everything else, from graph construction to analysis is new for this work.

IV. A Graph-based Characterization of Software Structure and Evolution

Most prior work on source code-based graph analysis has focused on characterizing single releases [24], [25], [26], [27], [28], [29], [30], [31], [32] or analyzing limited evolution time spans [33], or a longer evolution time span for a single program [34]. Therefore, one of the objectives of our study was to analyze complete lifespans of large projects and observe how the graphs evolve over time. This puts us in a position to answer questions such as:

Can graph metrics detect non-obvious "pivotal" moments in a program's evolution?

Are there invariants, metric values and evolution trends that hold across all programs?

We now proceed to showing how these metrics evolve over time for our examined applications and discuss how these changes and trends in graph metrics could affect various software engineering aspects, both for the product and for the process. The numeric results, i.e., metric values for the first and last releases, are shown in Table II. The evolution charts are in Figure 2. The data and figures refer to function call graphs.

Nodes and edges: The initial and final number of nodes and edges are presented in Table II. Due to lack of space, we do not present evolution charts. However, we have observed that some programs exhibit linear growth (Bind, SQLite, OpenSSH, MySQL) and some super linear growth (Blender Samba, VLC) in terms of number of nodes over time. The same observation holds for the evolution of the number of edges. This is intuitive, since, as shown in Table I, program size (eLOC) grows over the studied spans for all programs.

The only outlier was Sendmail where we noticed that neither the number of nodes, nor the number of edges increase, although eLOC increases (cf. Table I). We believe this to be due to the maturity of Sendmail—code is added to existing functions, rather than new functions being added, hence the increase in the size (in eLOC) of each function but no increase in the number of functions. The number of eLOC per node differ significantly across programs, from 10 to 323; the number of eLOC per edge ranged from 5

Application	Metric values											
	Nodes		Edges		Avg. degree		Clust. coeff.		Diameter		Assortativity	
	first	last	first	last	first	last	first	last	first	last	first	last
Firefox	9,332	28,631	89,045	787,297	19.08	54.39	0.062	0.111	12	16	-0.022	-0.07
Blender	5,525	30,955	14,567	80,304	5.27	5.18	0.094	0.091	17	30	-0.126	-0.097
VLC	445	5,049	961	15,131	4.31	5.99	0.122	0.095	10	14	-0.194	-0.086
MySQL	4,980	6,631	19,291	29,707	7.47	5.34	0.114	0.082	18	17	-0.113	-0.142
Samba	110	11,674	408	51,136	7.41	8.76	0.146	0.128	7	12	-0.287	-0.181
Bind	5,133	6,718	10,337	15,573	4.02	7.80	0	0.110	21	17	-0.165	-0.12
Sendmail	1,072	599	3,089	1,435	5.76	4.79	0	0	9	11	-0.198	-0.201
OpenSSH	214	1,030	773	4,056	7.22	7.87	0.187	0.141	6	12	-0.181	-0.224
SQLite	290	2,046	496	4,241	3.42	4.16	0	0	12	15	-0.245	-0.126
Vsftpd	597	982	921	1,712	3.08	3.48	0	0	11	12	-0.202	-0.206

Table II
METRIC VALUES (FUNCTION CALL GRAPHS) FOR *first* AND *last* RELEASES.

to 150. Values of both metrics decrease with evolution for Firefox, Blender, VLC, MySQL, OpenSSH and SQLite; and increase for Samba, Bind, Sendmail and Vsftpd.

Average degree: Intuitively, the degree of a function or module indicates its popularity. The average degree of a graph helps quantify coarseness: graphs with high average degrees tend to be tightly connected [35]. In Figure 2 we show the evolution of this metric for each program. We find that for all programs but MySQL, the average degree increases with time, albeit this increase tends to be slight, and the value range is 2–10. One interesting aspect to note is the average degree of Firefox, which is orders of magnitude higher, ranging from 20 to 60.

On further investigation we found that the graph topology for Firefox differs significantly from the remaining projects. The three notable observations are: (1) the majority of the nodes have low degree (average degree less than 20) and they are not connected with each other, (2) a large group of high-degree nodes (average degree 200–800) are interconnected with each other and form a dense core in the graph, and (3) most of the low degree nodes are connected with this dense core. We found that this group of nodes with high degree and high interconnectivity are part of the common library in Mozilla used by the majority of the products, including Firefox. On the contrary, in the other projects where the average degree is low, we found that: (1) the majority of nodes have degree close to the average degree of the graph, and (2) there are very few nodes of high degree and very few of them connect with each other.

Clustering coefficient: As defined in Section III-A, the clustering coefficient is a measure of how well-connected the local neighborhoods are in a graph. Zero values of this coefficient indicate a bipartite graph. High values of clustering coefficient indicate tight connectivity and violate good software engineering practice, because graph nodes do not form a hierarchy of levels of abstraction (due to the presence of horizontal and backward edges), which complicates program understanding, testing, and evolution [6], [36].

In our case (Table II), we found that for Vsftpd, SQLite, Sendmail and all but the last release of Bind, the clustering coefficient values are zero throughout the project's lifetime, suggesting bipartite graphs; we verified that indeed these programs have bipartite call graphs. In the case of VLC, we found that in version 0.7.0 there was a sudden rise in the clustering coefficient value. On further investigation we found that the Flac demuxer code was rewritten for this version; although the function signatures remained the same from the previous version, there was a significant change in intra-module calls in the demuxer module leading to an increase in clustering coefficient. For the remaining programs, we find that clustering coefficients are remarkably similar: their range is 0.08–0.20 and values decrease over time, with the exception of Firefox.

Number of nodes in cycles: Cycles in software structure affect software quality negatively. For example, cycles in the module collaboration graph indicate circular module dependencies, hence modules that are hard to understand and hard to test: "nothing works until everything works, " as per standard software engineering literature [36], [6]. Cycles in the call graph indicate a chain of mutually recursive functions, which again are hard to understand, test, and require a carefully orchestrated end-recursion condition. An increase in the number of nodes in cycles from one release to another would signify decrease in software quality, and indicate the need for refactoring. We observed that for Samba, MySQL, and Blender the number of nodes in cycles increases linearly with time, for OpenSSH, Bind, SQLite, and Vsftpd the number remains approximately constant with time, whereas for VLC and Sendmail there is no clear trend. For MySQL, a sudden increase in number of nodes in cycles is noticeable in version 5.0 (Oct. 2005); on further investigation we found that newly-added functions in the InnoDB storage engine code form strongly connected components in the graph, leading to an increase in the number of nodes in cycles. For reasons mentioned above, even a constant number of nodes in cycles (let alone an increasing one) is undesirable.

Graph diameter: From a maintenance standpoint, graphs of high diameter are undesirable. As the diameter measures distance between nodes, graphs with large diameter are more likely to result in deep runtime stacks, which

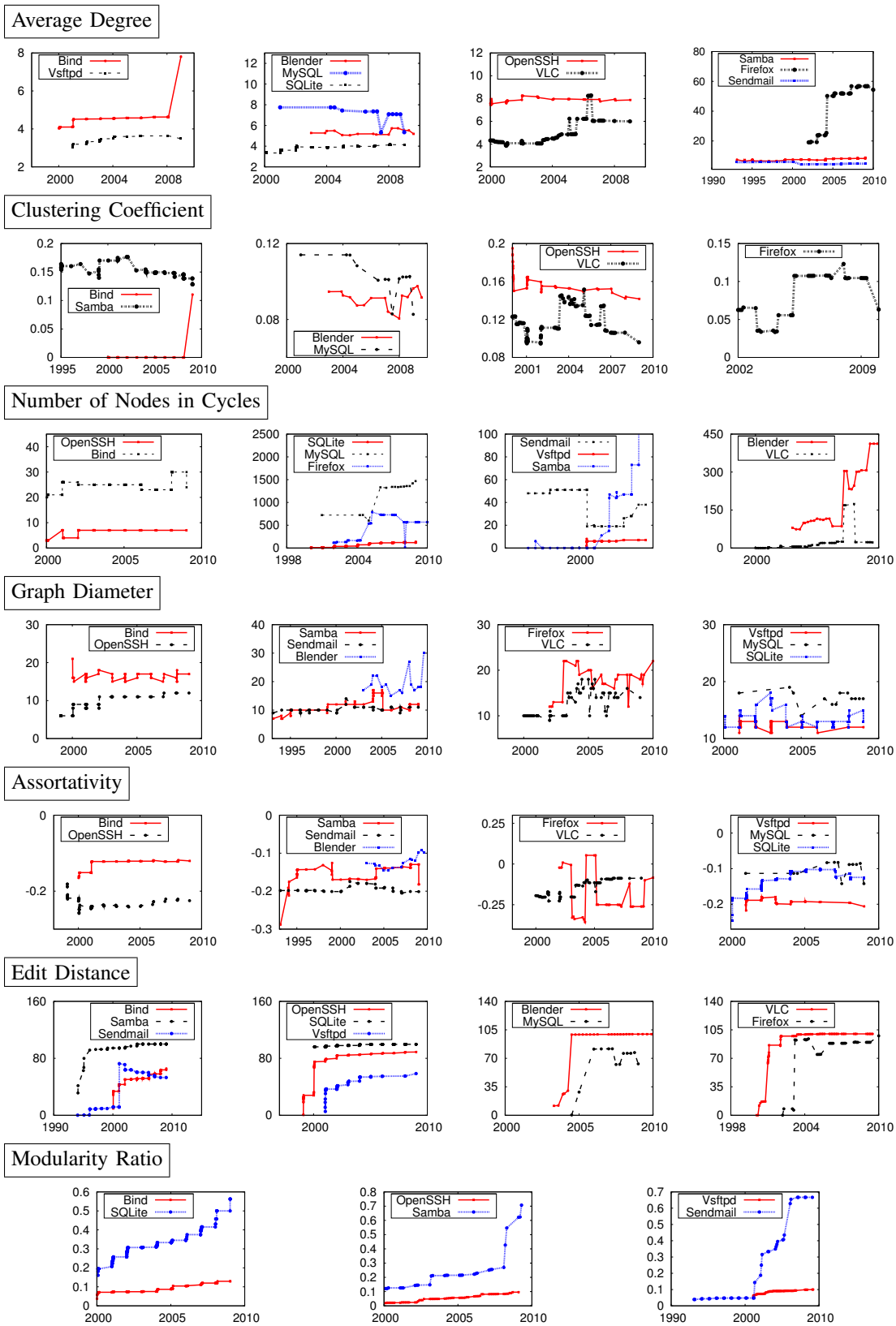


Figure 2. A graph-based characterization of software evolution; *x*-axis represents time.

hinder debugging and program understanding. As shown in Table II (columns 10 and 11) and in Figure 2, we notice that for our programs the diameter tends to stay constant or vary slightly, and the typical value range (10–20) is similar across all programs.

Assortativity: As explained in Section III-A, high values of assortativity indicate that high-degree nodes tend to be connected with other high degree nodes; low assortativity values indicate that high-degree nodes tend to connect with low-degree nodes [20]. As shown in Table II (columns 12 and 13) and in Figure 2, we notice that all the values of assortativity for all the programs are negative, which implies that, similar to biological networks, software networks exhibit disassortative mixing, i.e., high degree nodes tend to connect to low degree nodes and vice versa.

In Firefox, low assortativity stems from code reuse. Mozilla has its own function libraries for, e.g., memory management, and these functions were used by many other nodes. As a result library functions and modules exhibit very high degrees, and connect to many low degree nodes, hence contributing to low assortativity. In the future, we intend to further investigate the relationship between assortativity and code reuse.

Edit distance: This metric, as defined in Section III-A, captures the dynamic of graph structure changes, i.e., how much of the graph topology changes with each version. In Figure 2 we show how the graph edit distance changes over time. We find the same pattern for all programs: after a steep rise, the edit distance plateaus or increase slightly, i.e., is a step-function. This observation strengthens the conclusions of prior research [33], namely that software structure stabilizes over time, and the only tumultuous period is toward the beginning. We found that these steep edit distance rises are due to major changes and they indicate that software has reached structural maturity. For example, the pivotal moment for Samba is release 1.9.00 (Jan 22, 1995), where 131 modification requests were carried out, whereas for the versions prior to 1.9.00, the average number of modification requests per release was 15. We also computed the eLOC difference for release 1.9.00 and found it to be 2kLOC, *less* than the 3kLOC average of the previous releases, which shows how graph-based metrics can reveal changes that would go undetected when using LOC measures.

Modularity ratio: This metric reveals whether projects follow good software engineering practice, i.e., whether, over time, the cohesion/coupling ratio increases, indicating better modularity. This turned out to be the case for all programs, except Firefox version 1.5 (see bottom of Figure 2, and Figure 4; we show VLC, Blender, MySQL and Firefox in a separate figure because we used modularity ratio for prediction).

Discussion: We are now in a position to answer the questions posed at the beginning of this section. We have observed that indeed, software structure is surprisingly similar across programs, in absolute numbers (see the similar ranges for average degree, clustering coefficient, graph diameter), which suggests intrinsic lower and upper bounds on variation in software structure. There are also similarities in trends and change patterns (cf. edit distance, clustering coefficient, modularity ratio) which suggest that programs follow common evolution laws. For those releases where graph metrics change significantly, we found evidence that supports the "pivotal moment" hypothesis. For example, in Firefox, we find significant changes in average degree, clustering coefficient, and edit distance for release 2.0 (Oct. 2006); indeed, release notes confirm many architectural and feature enhancements introduced in that version. Similarly, for OpenSSH we found that one such moment was release 2.0.0beta1 (May 2000), a major version bump from prior release (1.2.3), that incorporated 143 modification requests, whereas the average modification requests per release until that point was 27 and this change is reflected in significant change of values for clustering coefficient, edit distance, and assortativity metric. This evidence strengthens our argument that graph metrics are good measures that can reveal events in evolution.

Bug Severity	**Description**	**Rank**
Blocker	Blocks development testing work	6
Critical	Crashes, loss of data, severe memory leak	5
Major	Major loss of function	4
Normal	Regular issue, some loss of functionality	3
Minor	Minor loss of function	2
Trivial	Cosmetic problem	1
Enhancement	Request for enhancement	0

Table III
BUG SEVERITY: DESCRIPTIONS AND RANKS.

So far our discussion has centered on changes in structural (graph) metrics and understanding how software structure evolves. We now move on to discussing how structural metrics can be used to predict non-structural attributes such as bug severity, effort, and defect count.

V. PREDICTING BUG SEVERITY

We present a novel approach that uses graph-based metrics associated with a function or module to predict the severity of bugs in that function or module. When a bug is reported, the administrators first review it and then assign it a severity rank based on how severely it affects the program. Table III shows levels of bug severity and their ranks in the Bugzilla bug tracking system. A top priority for software providers is to not only minimize the total number of bugs, but to also try to ensure that those bugs that do occur are low-severity, rather than *Blocker* or *Critical*. Moreover, providers have to do this with limited numbers of developers and testers. Therefore, a bug severity predictor would directly improve software quality and robustness by focusing the testing and verification efforts on highest-severity parts.

We use *NodeRank* to help identify critical functions and modules, i.e., functions or modules that, when buggy,

are likely to exhibit high-severity bugs. As discussed in Section III-A, *NodeRank* measures the relative importance of a node—function or module—in the function call or module collaboration graphs, respectively. By looking up the *NodeRank*, maintainers have a fast and accurate way of identifying how critical a function or module is. We now state our hypothesis formally:

H1: Functions and modules of higher *NodeRank* will be prone to bugs of higher severity.

Data set: We used six programs: Blender, Firefox, VLC, MySQL, Samba, and OpenSSH for this analysis. We collected the bug severity information from bug reports. For each bug report we collected the patches associated with it and from each patch we found out the list of functions that were changed in the bug fix. Therefore, we have information about how many times a function has been found buggy, and what the median severity of those bugs was.

Results: We were able to validate *H1* for our study. We correlated the median bug severity of each function and module with its *NodeRank*. The results are shown in column 2 of Tables IV (for functions) and V (for modules). As a first step, we focus on nodes with a *NodeRank* in Top 1% since bug severity for functions and modules exhibit a skewed distribution where Top 1% of the nodes are affected by majority of the bugs. Note that for sizable programs such as Firefox, the number of nodes exceeds 25,000, hence even Top 1% can mean more than 250 functions. We find the correlation between *NodeRank* and *BugSeverity* to be high: 0.6—0.86. This suggests that *NodeRank* is an effective predictor of bug severity, and can be used to identify "critical" functions or modules. We have also computed correlation values between function bug severity and standard software engineering quality metrics (cyclomatic complexity[2], interface complexity[3]). As can be seen in columns 3–4 of Table IV we found these values to be close to zero, meaning that these metrics are not effective in identifying critical functions.

The node degree is not a good predictor of bug severity. We investigated if the node degree could be just as good a severity predictor as the NodeRank. The answer was no. We compute the correlations between bug severity and node in- and out-degrees. The results are shown in Table IV, columns 5–6 (functions), and Table V, columns 3–4 (modules); note how in- and out-degrees are poor bug severity predictors.

We also compute the *NodeRank–BugSeverity* correlation for the remaining 99% of the nodes and found similar trends. As expected, in the lower end of the NodeRank, there is significant statistical noise, which makes estimating a correlation coefficient difficult. However, there is definitely a clear high level trend between NodeRank and BugSeverity even in the absence of a well-defined linear correlation. Overall,

[2]McCabe's cyclomatic complexity is the number of logical pathways through a function [37].

[3]Computed as the sum of number of input parameters to a function and the number of return states from that function [38].

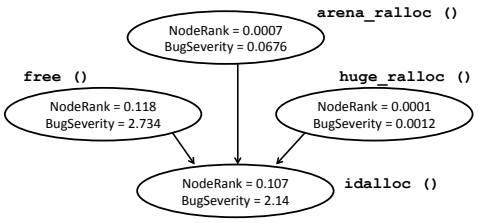


Figure 3. Firefox call graph and bug severity (excerpt).

our work suggests a useful practical approach: in a resource-constrained testing and verification setting, one should start with the nodes with high NodeRank value. We observe that the correlation coefficients between BugSeverity and NodeRank for Firefox's function-call graphs are smaller than in other projects. Firefox shares some bugs (rooted in shared functions, and to a lesser extent, shared modules) with other Mozilla projects (e.g., Thunderbird) which makes attributing a bug to Firefox alone difficult; this noise in the data set counts for the slightly lower correlation compared to other projects.

To illustrate this correlation, in Figure 3 we present an excerpt from Firefox's call graph. Within each node (function) we indicate that node's *NodeRank*, as well as the average *BugSeverity* for past bugs in that function. As we can see, verification efforts should focus on functions free() and idalloc(), as their *NodeRank*s are high, which indicates that the next bugs in these functions will be high-severity, in contrast to functions arena_ralloc() and huge_ralloc() that have low *NodeRank*s and low *BugSeverity*.

Program	NodeRank	Cyclom. complex.	Interface complex.	In degree	Out degree
Blender	**0.60**	0.07	0.17	0.10	0.05
VLC	**0.83**	0.19	-0.06	-0.09	-0.003
MySQL	**0.77**	-0.05	-0.11	-0.06	-0.06
Samba	**0.65**	-0.207	-0.19	0.23	-0.06
OpenSSH	**0.86**	0.003	0.12	0.04	-0.34
Firefox	**0.48**	0.16	-0.28	0.18	-0.26

Table IV
CORRELATION OF *BugSeverity* WITH OTHER METRICS FOR TOP 1% *NodeRank* FUNCTIONS (P-VALUE = 0.01).

Program	NodeRank	In degree	Out degree
Blender	**0.79**	-0.04	-0.0008
VLC	**0.82**	0.21	-0.11
MySQL	**0.73**	-0.20	-0.24
Samba	**0.78**	-0.02	-0.10
OpenSSH	**0.65**	-0.22	-0.19
Firefox	**0.704**	-0.17	-0.38

Table V
CORRELATION OF *BugSeverity* WITH OTHER METRICS FOR TOP 1% *NodeRank* MODULES (P-VALUE = 0.01).

VI. PREDICTING EFFORT

A leading cause of high software maintenance costs is the difficulty associated with changing the source code,

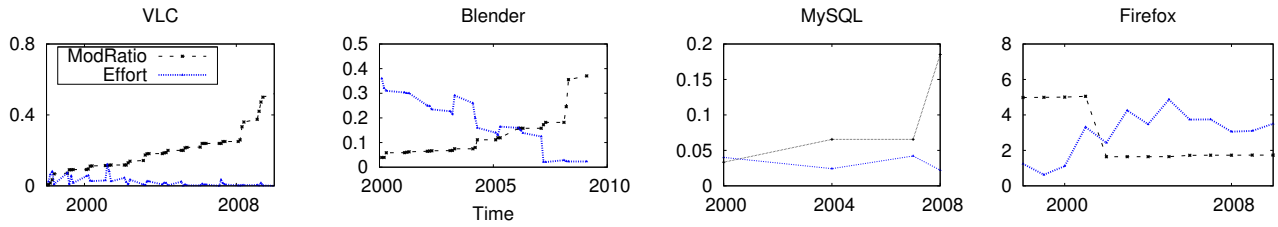


Figure 4. Change in *ModularityRatio* with change in *Effort*; *x*-axis represents time.

Program	Lag	F-prob
Blender	1	0.0000015
VLC	3	0.3673
Firefox	1	0.00056

Table VI
GRANGER CAUSALITY TEST RESULTS FOR *H2*.

e.g., for adding new functionality or refactoring. We propose to identify difficult-to-change modules using the novel module-level metric called *Modularity Ratio*, defined in Section III. Intuitively, a module A's modularity ratio, i.e., $Cohesion(A)/Coupling(A)$ indicates how easy it is to change that module. To quantify maintenance effort, the number of commits is divided by the churned eLOC for each module in each release—-this is a widely used metric for effort [23]. Therefore, our hypothesis is formulated as follows:

H2: Modules with higher *ModularityRatio* have lower associated maintenance effort.

Data set: We used four programs, Blender, Firefox, VLC and MySQL for this analysis. The effort data for these programs is available from our prior work [23].

Results: We were able to validate *H2* for our study. We found that, as the *ModularityRatio increases* for a module, there is an associated *decrease* in maintenance effort for that module, which means the software structure improves. In Figure 4 we plot the results for each program. The *x*-axis represents the version; for each version, in gray we have the mean modularity ratio, and in blue we have mean maintenance effort. We ran a Granger causality test[4] on the data in the graph. We use causality testing instead of correlation because of the presence of time lag; i.e., our hypothesis is that changes in modularity ratio for one release would trigger a change in effort in one of the future releases. As shown in Table VI, we obtained statistically significant values of *F-prob* for the Granger causality test on modularity ratio and effort.[5] The lag value indicates that a change in modularity ratio will determine a change in effort in the next release (Blender, Firefox) or in three releases (VLC).

VII. PREDICTING DEFECT COUNT

Intuitively, a stable, highly cohesive development team will produce higher-quality software than software produced by a disconnected, high-turnover team [39]. Therefore, we are interested in studying whether stable team composition and structure will lead to higher levels of collaboration, which in turn translates into higher quality software. We are in a good position to characterize team stability by looking at the evolution of developer collaboration graphs. To measure how much graph structure changes over time we use the edit distance metric defined in Section III-A. Concretely, we hypothesize that periods in software development that show stable development teams will result in periods of low defect count. To test this, we form the following hypothesis:

H3: An increase in edit distance in Bug-based Developer Collaboration graphs will result in an increase in defect count.

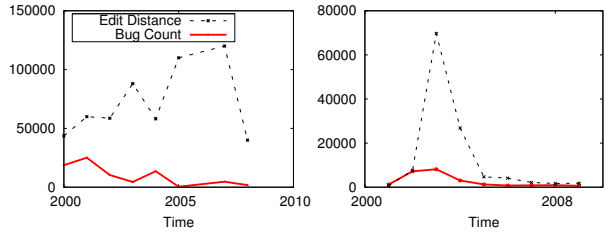
Data Set: We used the Firefox and Eclipse bug reports to build the developer collaboration graphs. For Firefox, we analyzed 129,053 bug reports (May 1998 to March 2010). For Eclipse, we considered bugs numbers from 1 to 306,296 (October 2001 to March 2010).

Results: We were able to validate *H3* for our study. From bug reports, we constructed *Bug-based Developer Collaboration* graphs as explained in Section II-B. We constructed these graphs for each year, rather than for each release, as some releases have a small number of bugs. Next, we computed the graph edit distance from year to year and ran a correlation analysis between edit distance for year Y and defect count at the end of year Y. We found that there is a strong positive correlation between these two measures, as shown in Figure 5. This shows that team stability affects bug count; our intuition is that, when developers collaborate with people they have worked with before, they tend to be more productive than when they work with new teammates. A similar finding has been reported by Begel et al. [40] for commercial work environments: working with known teammates increases developer productivity. Although open source projects lack any social structure and central management, team collaboration does affect software quality.

Bug-tossing based collaboration is a better defect predictor than commit-based collaboration. We have also tested the same hypothesis with commit-based developer collaboration graphs; however we did not find any correlation between edit distance in those graphs and effort, which suggests that bug-tossing graphs are more useful than commit-exchanges for studying developer relationships in open source projects.

[4]The Granger causality test is a statistical hypothesis test for determining whether one time series is useful in forecasting another.

[5]We cannot claim statistically significant results for MySQL due to small sample size; effort values for only 4 versions of MySQL were available.



(a) Eclipse, correlation = 0.504 (b) Firefox, correlation = 0.759

Figure 5. Change in collaboration graph *Edit Distance* v. *Defect Count*.

VIII. Related Work

Software Network Structural Properties: Several prior efforts have examined single-version structural properties; graphs represent class collaboration [25], [26], [32], [28], [41], module dependency [24], inheritance and aggregation [31]; metrics used include degree distribution, degree correlation, clustering, and verifying power-law relationships. Other efforts have looked at graph motifs [27], motif stability [29], and structural complexity [30].

Graphs and Evolution: Vasa et al. [33] studied the type dependency graphs of 12 Java projects and their evolution over one year using three degree-based metrics: fan-in, fan-out and branch count. Wang et al. [34] studied the evolution of the Linux kernel (223 versions) using complex networks analysis. They used node degree distribution and average path length of the call graphs as metrics and found that the call graphs of file system and drivers modules are scale-free small-world complex networks and observed strong *preferential attachment* growth.

Our study differs from the aforementioned efforts in three significant ways: (1) we analyze a broad range of large projects written in C, C++, or both; in addition, we study multiple releases of the same project which allows us to analyze the evolution of graph topologies, (2) we use graph-based metrics as software quality predictors, and (3) we look at developer-collaboration graphs in two large, widely-used open source projects which reveal how social networking among developers affect software quality.

Software Networks for Failure Prediction: Zimmermann et al. [42] construct source code dependency graphs in Windows Server 2003. They used four different measures of complexity to describe these dependency graphs and to predict the failure-proneness of a given source code artifact. Premraj et al. [43] and Tosun et al. [44] replicated Zimmermann et al.'s approach on open source software. Schroter et al. [45] performed an empirical study of 52 Eclipse plug-ins and built a model based on the USES relationships between software components to predict failure-prone components, based on its design data and which are the most failure-prone components in the project. Nagappan et al. [46] showed that dependency graphs built from software component dependencies can be used as efficient indicators of post-release failures specifically for Windows Server 2003. Holmes et al. [47] showed how static and dynamic call graphs of the same program can be used to predict which functions may be affected by a modification.

Our study is different from these efforts in two significant ways: (1) we analyze multiple releases of the same project which allows us to analyze the evolution in the topologies of these graphs, (2) we propose NodeRank, a metric which is powerful in identifying critical spots in the software. Additionally, our *ModularityRatio* metric is capable of predicting maintenance effort.

Bug Severity Prediction: Menzies et al. [48] and Lamkanfi et al. [49] used text classification to predict the severity of a bug from the text of its bug report, which yields up to 90% and 65–85% accuracy, respectively. In contrast, our *NodeRank* works at both function and module level and can predict bug severity *before* a bug report is filed.

Developer Collaboration: Abreu et al. [50] studied developer communication frequency for Eclipse JDT and found that this frequency is positively correlated with the number of bugs in the project. Bird et al. [51] studied coordination among developers by analyzing 7 years of the Apache developer mailing list, and found a strong relationship between the levels of email activity and source code activity for a developer. Pinzger et al. [52] used a heterogeneous graph that associates developers and module for Microsoft Windows Vista and found that modules located in periphery of the network are less error prone. Their analysis also shows that the number of developers and number of commits are significant predictors for the probability of post-release failures.

Our work differs in two ways from these prior efforts: (1) we look at multiple versions of developer collaboration by constructing these graphs for each year and analyzing how they change over time, and (2) we show that there exists a high positive correlation between edit distance between these successive developer graphs and the defect count.

IX. Conclusions

We have provided a graph construction method and a set of metrics that capture the structure and evolution of software products and processes. Using a longitudinal study on large open source programs, we have demonstrated that source code-based graph metrics can reveal differences and similarities in structure and evolution across programs, as well as point out significant events in software evolution that other metrics might miss. We have also shown that graph metrics can be used to predict bug severity, maintenance effort and defect-prone releases.

Acknowledgments. This research was supported in part by NSF grants CCF-1149632 and CNS-1064646. We thank the anonymous referees for their helpful comments on this paper.

References

[1] Gartner, "Gartner Trims Worldwide IT Spending Growth Forecast to 3.9 Percent for 2010," http://www.gartner.com/it/page.jsp?id=1393414.

[2] Microsoft, "10-K 2009 Annual Report," http://www.microsoft.com/msft/aspx/secfilings.aspx?DisplayYear=2009.

[3] IBM, "IBM 2009 Annual Report," http://www.ibm.com/annualreport/2009/.

[4] J. Koskinen, "Software maintenance costs," Sept 2003, http://users.jyu.fi/~koskinen/smcosts.htm.

[5] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.

[6] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Prentice Hall PTR, 2002.

[7] I. Sommerville, *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[8] NIST, "The economic impacts of inadequate infrastructure for software testing," Planning Report, May 2002.

[9] I. Neamtiu, G. Xie, and J. Chen, "Towards a better understanding of software evolution: an empirical study on open-source software," *JSME*, 2011.

[10] M. Iliofotou, B. Gallagher, T. Eliassi-Rad, G. Xie, and M. Faloutsos, "Profiling-by-association: A resilient traffic profiling solution for the internet backbone," in *CoNEXT'10*.

[11] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese, "Network monitoring using traffic dispersion graphs (tdgs)," in *IMC*, 2007.

[12] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On Power-Law Relationships of the Internet topology," *SIGCOMM'99*.

[13] R. Albert, H.Jeong, and A. Barabasi, "Diameter of the world wide web," *Nature*, vol. 401, 1999.

[14] Q. Yang, G. Siganos, M. Faloutsos, and S. Lonardi, "Evolution versus Intelligent Design: Comparing the Topology of Protein-Protein Interaction Networks to the Internet," in *CSB'06*.

[15] M. Gorman, "Codeviz," http://www.skynet.ie/ mel/projects/-codeviz/.

[16] J. Bohnet and J. Döllner, "Visual exploration of function call graphs for feature location in complex software systems," ser. SoftVis '06.

[17] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *ICSM*, 2010.

[18] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, 1998.

[19] M. McPherson, L. Smith-Lovin, and J. M. Cook, "Birds of a feather: Homophily in social networks," *Annual Review of Sociology*, vol. 27, no. 1, 2001.

[20] M. E. J. Newman, "Assortative mixing in networks," *Phys. Rev. Lett.*, vol. 89, no. 20, 2002.

[21] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE*, 2005.

[22] J. Fernández-Ramil, D. Izquierdo-Cortazar, and T. Mens, "What does it take to develop a million lines of open source code?" in *OSS*, 2009, pp. 170–184.

[23] P. Bhattacharya and I. Neamtiu, "Assessing programming language impact on development and maintenance: A study on C and C++," in *ICSE'11*.

[24] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM TOSEM*, vol. 18, no. 1, pp. 1–26, 2008.

[25] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," *Phys. Rev. E*, vol. 68, no. 4, p. 046116, 2003.

[26] S. Valverde and R. V. Solé, "Hierarchical small worlds in software architecture," *Dynamics of Continuous Discrete and Impulsive Systems: Series B; Applications and Algorithms*, 2007.

[27] S. Valverde and R. V. Sole, "Network motifs in computational graphs: A case study in software architecture," *Phys. Rev. E*, vol. 72, no. 2, 2005.

[28] R. V. Solé, R. Ferrer-Cancho, J. M. Montoya, and S. Valverde, "Selection, tinkering, and emergence in complex networks," *Complexity*, vol. 8, no. 1, 2002.

[29] Y. Ma, K. He, and J. Liu, "Network motifs in object-oriented software systems," *CoRR*, vol. abs/0808.3292, 2008.

[30] Y. Ma, K. He, and D. Du, "A qualitative method for measuring the structural complexity of software systems based on complex networks," in *APSEC '05*, pp. 257–263.

[31] C. S. Wheeldon, R., "Power law distributions in class relationships," in *SCAM'03*, pp. 45–54.

[32] S. Valverde, R. F. Cancho, and R. V. Solé, "Scale-free networks from optimal design," *EPL*, vol. 60, no. 4, 2002.

[33] R. Vasa, J.-G. Schneider, and O. Nierstrasz, "The inevitable stability of software change," in *ICSM*, 2007, pp. 4–13.

[34] L. Wang, Z. Wang, C. Yang, L. Zhang, and Q. Ye, "Linux kernels as complex networks: A novel method to study evolution," in *ICSM*, 2009.

[35] P. Mahadevan, C. Hubble, D. Krioukov, B. Huffaker, and A. Vahdat, "Orbis: rescaling degree correlations to generate annotated internet topologies," in *ACM SIGCOMM*, 2007.

[36] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, pp. 1053–1058, December 1972.

[37] T. J. McCabe, "A complexity measure," in *ICSE'76*.

[38] RSM Metrics, "http://msquaredtechnologies.com/m2rsm/docs/index.htm."

[39] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *FSE'08*.

[40] A. Begel, K. Y. Phang, and T. Zimmermann, "Codebook: Discovering and exploiting relationships in software repositories," in *ICSE*, 2010.

[41] A. Potanin, J. Noble, and R. Biddle, "Generic ownership: practical ownership control in programming languages," in *OOPSLA '04*, pp. 50–51.

[42] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *ICSE '08*.

[43] R. Premraj and K. Herzig, "Network versus code metrics to predict defects: A replication study," in *ESEM*, 2011.

[44] A. Tosun, B. Turhan, and A. B. Bener, "Validation of network measures as indicators of defective modules in software systems," in *PROMISE*, 2009.

[45] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *ISESE*, 2006.

[46] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," ser. ESEM, 2007.

[47] R. Holmes and D. Notkin, "Identifying program, test, and environmental changes that affect behaviour," in *ICSE*, 2011.

[48] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *ICSM*, 2008.

[49] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *MSR*, 2010.

[50] R. Abreu and R. Premraj, "How developer communication frequency relates to bug introducing changes," in *IWPSE-Evol*, 2009.

[51] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *MSR'06*.

[52] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *FSE*, 2008.