Putting It All Together: Using Socio-technical Networks to Predict Failures

Conference Paper · November 2009

DOI: 10.1109/ISSRE.2009.17 · Source: DBLP

5 authors, including:

Christian Bird
Microsoft
80 PUBLICATIONS **3,300** CITATIONS
SEE PROFILE

Brendan Murphy
Microsoft
43 PUBLICATIONS **2,543** CITATIONS
SEE PROFILE

Premkumar Devanbu
University of California, Davis
222 PUBLICATIONS **8,715** CITATIONS
SEE PROFILE

Putting it All Together: Using Socio-Technical Networks to Predict Failures

Christian Bird[1], Nachiappan Nagappan[2], Harald Gall[3], Brendan Murphy[2], Premkumar Devanbu[1]
[1]University of California, Davis, USA
[2]Microsoft Research
[3]University of Zurich, Switzerland
{cabird,ptdevanbu}@ucdavis.edu {nachin,bmurphy}@microsoft.com gall@ifi.uzh.ch

Abstract

Studies have shown that social factors in development organizations have a dramatic effect on software quality. Separately, program dependency information has also been used successfully to predict which software components are more fault prone. Interestingly, the influence of these two phenomena have only been studied separately. Intuition and practical experience suggests, however, that task assignment (i.e. who worked on which components and how much) and dependency structure (which components have dependencies on others) together interact to influence the quality of the resulting software. We study the influence of combined socio-technical software networks *on the fault-proneness of individual software components within a system. The network properties of a software component in this combined network are able to predict if an entity is failure prone with greater accuracy than prior methods which use dependency or contribution information in isolation. We evaluate our approach in different settings by using it on Windows Vista and across six releases of the Eclipse development environment including using models built from one release to predict failure prone components in the next release. We compare this to previous work. In every case, our method performs as well or better and is able to more accurately identify those software components that have more post-release failures, with precision and recall rates as high as 85%.*

1. Introduction

Software failures are becoming increasingly important and costly. A study by the National Institute of Standards and Technology in the U.S. estimated that the annual cost of software bugs is about $59.5 Billion [1]. At the corporate level, the ability to identify and correct software defects prior to release saves a company from increased cost, customer dissatisfaction, and loss of marketshare. As such, there is a rich history of tools which can automatically identify software defects before end-users encounter them. It can be very difficult and expensive to test all of the components of a large and complex system. However, the complexity inherent in large software systems can be leveraged to aid in locating those components which are particularly defect prone. Encouraging results in prior research indicate that it is possible to predict which components are likely locations of defect occurrence using a component's development history, and dependency structure.

In this paper, two key properties of software components in large systems are dependency relationships (which components depend on or are depended on by others), and development history (who made changes to the components and how many times). Thus we can link software components to other components a) in terms of their dependencies, and also b) in terms of the developers that they have in common. This linkage has been used to construct software component networks. Prediction models based on the topological properties of components within them have proven to be quite accurate [2], [3], [4]. Components which play key roles and are central in these networks tend to be more failure prone than components in the surrounding areas.

We argue that these forms of information should be used together. The intuition behind our approach is that software components may be related through important but *different* types of relationships. By aggregating these relationships our ability to predict failures will increase. We do this in two ways. First, we build each type of network separately and use network analysis on both to gather metrics for use in a predictive model. Second, we build a *socio-technical network* which combines the nodes and edges from both the dependency network and the contribution network and use metrics gathered from this network in a predictive model.

We evaluate our approach by collecting data from Microsoft Windows Vista and ECLIPSE development and using logistic regression analysis. Our regression models relate social network centrality measures of components with the number of post-release failures. Results of our empirical study show a strong correlation between the centrality of software components and the number of post-release failures and indicate that combining dependency and contribution data results in prediction models that have higher recall and precision.

We make the following contributions in this paper:

1) We present (to the best of our knowledge) the first paper in the software engineering or Computer Supported Cooperative Work (CSCW) community that combines the developer and the code level view of software systems to predict code quality.
2) We present an approach for predicting failure prone software components using development history and

dependency information that performs better than previous research.
3) We compare our approach directly with prior network analysis based prediction methods by evaluating each on the same data sets.
4) We show that our method works in multiple, diverse contexts with different processes by using it on large code bases in both a traditional industrial setting (Windows Vista) and an open source software (OSS) setting (Eclipse).
5) We demonstrate how this approach can be used in practice by accurately predicting failure prone components in one release based on models built from prior releases.

2. Background and Prior Work

Our work arises from two lines of work: studies of *social networks* within project teams, and *technical networks* of components within systems.

Technical networks have been used in previous work to build prediction models for failures. Zimmermann *et al.* [4], [3] constructed networks from dependency information for binaries and subsystems in Windows Server 2003. This study used social network analysis on dependency information to build prediction models for post-release failures. Their results indicated that models built on social network metrics were better indicators of future failures than models based on standard source code metrics. Their approach leveraged SNA metrics to capture both local and global effects of network connectivity on defect-proneness.

Prior work has also shown that software artifact properties are directly influenced by social network properties of teams, such as their the *email* interactions, and their *contribution* history. of developers. In earlier work, Bird *et al.* [5] constructed *email* social networks from open source project mailing lists and found that social network analysis measures were highly correlated with development activity. In addition, they found that global connectivity measures such as *betweenness* [6] were better indicators of development activity than local measures such as degree centrality. Pinzger *et al.* [2] used *contribution history* to construct the networks of binaries and the developers that contributed to them. They found that measures such as degree centrality, closeness centrality, and Bonacich power (see [7], [8], [9], [10] for a survey of these measures) in contribution networks also had very good predictive power in determining failure-prone binaries.

Meneeley *et al.* [11] created networks that consisted solely of developers where edges between developers were based on collaboration on common files. They used social network analysis to assign values of metrics such as betweenness, degree, and closeness to developers. The value of a metric for a file was based on the values of the developers that contributed to that file (such as maximum, sum, or average of a metric for developers for a file). They evaluated their approach on an industrial product from Nortel. They found that a model using these metrics explained 60% of the variance of failures during the testing phase, but only 2.6% of the post-release failures.

Combining social and technical networks has recently become a subject of study. Socio-technical networks encode connections between people, connections between technical artifacts and connections between people and artifacts. Although we do not include developer social interaction via email, IM, or other communication media, we do capture collaboration through joint work artifacts.

Amrit *et al.* [12] first proposed use of socio-technical networks which he calls "affiliation networks" in the context of evaluating Conway's Law and claimed that important information is embedded in the topology of these networks. He posited that "We can use the idea of the affiliation network to improve current design, execution, and productivity of software process models".

Indeed, Valetto *et al.* [13] do just that by examining socio-technical networks and measuring the *socio-technical congruence*, degree of communication and coordination between developers who are related to the same software component or dependent software components.

Cataldo *et al.* [14], [15] looked at time to resolution for modification requests (MRs) at a commercial software development company. They found that time to resolution for an MR, a, with a dependency on another MR, b, was decreased if those responsible for a had some form of contact (geographical locality, IRC, etc.) with those responsible for b. They find that developers take 32% less time to complete tasks when this form of "congruence" is in the socio-technical network.

3. Theory

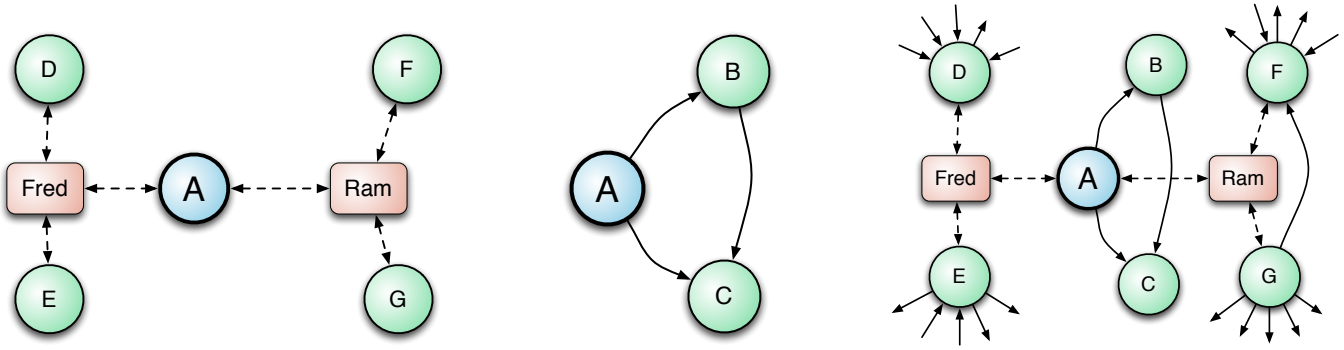
In this section we present arguments for combining dependency and contribution topological information when examining defect proneness.

3.1. Network Definitions

We begin by formally describing the three networks of binaries that are used in our analysis. To illustrate our methods, we include a running example of a simple software system and it's corresponding networks in figure 1. In these networks, circles represent software components and boxes represent developers. A solid edge between two components is directed and denotes a dependency relationship. A dashed edge between a developer and a binary is undirected and denotes a source code contribution from the developer to the binary.

Contribution Network

The contribution network captures the contributions of developers to software components within the system. We use a mapping from source files to software components along with development logs which include which developers contributed to which files to build a bipartite network.



(a) Contribution Network (b) Dependency Network (c) Socio-technical Network

Figure 1: Examples of software component networks. Circles denote components and rectangles, developers. Solid lines are directed dependencies and dashed lines are undirected and represent contributions.

Formally, we define this network as follows. Let S be the set of software components, and D be the set of developers that made commits to the source code for these components. The *contribution* network is then $G_c = (V_c, E_c)$ where the vertices are $V_c = D \cup S$ and $E_c \subset D \times S$ is the set of edges such that $(d, s) \in E_c$ if developer d contributed to software component s. Edges are weighted based on the number of commits made to a software component s by a developer d. Note that edges in this graph are undirected to allow for paths flowing in either direction, since developers act as "bridges" between components. The *contribution* network for our example software system is depicted in figure 1a. More specifics are given in [2].

If two components have the contributions from the same developer, then the components have "shared authorship". The contribution network captures shared authorship between components, and thus, in a sense captures shared expertise between components. If Ram authors two components, A and F, then he represents a person who has knowledge and responsibility of both A and F. Components with many connections are those which share authorial responsibility with many other components; such components might share many cross-cutting concerns [16] with other components. Components that are on many paths in a network but have a low number of connections are likely to lie on organizational boundaries. These are not subject to a high level of shared authorship, but mediate between others that have highly shared ownership. Such components represent critical bottlenecks for expertise flow; they might also be locations where organizational boundaries are crossed, and thus be loci for communication breakdowns or bottlenecks.

Dependency Networks

A dependency network models the dependency relationships between the software components within the system. Figure 1b shows a simple dependency graph. We use the software component dependency relationships as applicable for the domain, language, and granularity of the system (e.g. callgraphs, class inheritance or coupling in ECLIPSE, library type and function dependencies within Windows Vista). We refer to this graph as the *dependency* graph and denote it formally as $G_d = (V_d, E_d)$ where the vertices V_d is the set of software components and $E_d \subset V_d \times V_d$ is the set of directed edges, such that $(v_1, v_2) \in E_d$ if component v_1 has a dependency on v_2. The example shows a system in which both `A` and `B` are dependent on `C` and `A` is also dependent on `B`. We refer the reader to prior work ([4]) for more detail regarding the construction of these networks.

A pure software dependency graph (such as a call graph, or a systems dependence graph) capture flow of information and/or control within a large system. In such a graph, the strength and degree of a component's connections to its immediate neighbors (a local property) indicates how strongly it is coupled with other components. This can be expected to influence the degree to which a component is defect prone. Likewise, high closeness centrality (a measure of the average distance of a component to every other component) might indicate that a component is in the "core" part of the system.

Socio-Technical Networks

Prior work [2], [11], [3], [4] indicates that the likelihood of a component to fail is strongly related to it's topology in networks based on different types of relationships. In fact, the above two networks capture different types of phenomena that might lead to defect introduction and defect importance.

In an effort to understand the differences between the different prior network based approaches, we implemented predictive models based on dependency and contribution networks and performed a manual inspection of the defect-prone software components that were misclassified by *both* models. Figure 1 illustrates a common scenario that we encountered. Component `A` represents a defect prone component that was not identified by either approach. Figure 1b shows that the dependency network is small, with `A` having two dependencies and no dependents. In addition Figure 1a

shows that only two developers contribute to A and that they also work on a few other components. However, Figure 1c shows that the other binaries that Fred and Ram contribute to have many dependencies and dependents. These binaries play key roles in the system and a number of them are defect prone. The fact that Fred and Ram contribute to these components *and* A, represents some form of latent relationship between A and D, E, F, and G. It is only after considering *both* kinds of relationships that the import of A becomes apparent. We found many instances of this scenario (and also it's dual, in which components developed by *many* developers were connected to another, defect prone, component via simple dependency relationships) in our manual inspections.

We therefore lift the level of abstraction by claiming that binaries have a multitude of relationship types. Common developers and dependencies are two of a number of possible software component relationships. These relationships encode a multi-mode relationship graph [6], with different types of nodes (developers and software components in our case) and different types of edges: various forms of dependencies, and contributions from developers. While examining the relationships between software components separately has proven useful, we claim that they should be considered in concert.

We create a socio-technical network by combining dependency and contribution relationships into on graph. Both of the above networks deal with information & control flow. The joint network then, captures the interaction between the two.

We refer to this network as the *socio-technical* network, G_a. One issue in aggregating the vertices and edges is the fact that G_c is weighted and undirected while G_d is directed and unweighted. To resolve directedness, we add two directed edges (one in either direction) between a developer d and a software component s if d contributed to s. To resolve the issue of weights, we include the number of commits from d to s as the weight on contribution edges and set the weight of software dependencies dependencies to 1. Many social network measures (such as betweenness), convert the network to an unweighted network prior to analysis. For those that can operate on both weighted and unweighted, we perform the analysis on both versions of the network and use both measures in our model. For instance, degree on the unweighted contribution network represents the number of distinct developers who contributed to a component while the weighted degree is the number of actual commits.

3.2. Social Network Analysis Measures

We calculate network analysis measures on a per component basis within the software component networks. These measures can be broadly divided into two categories. Global measures examine the position of the component within the context of the entire network and include *betweenness*, *Bonacich Power*, and *eigenvector centrality*. Local measures only take into account the neighborhood of nodes within one or two hops of software component. These include measures such as *degree*, *size* of the network, and *edge density*. We refer the reader to a more thorough review of social network analysis measures by examining, [7], [8], [9], [10] and the comprehensive online help in [17]. In these discussion, an *edge* between two components represents a relationship between the component and may represent different relationship in different types of networks.

Global Measures

Betweenness centrality [7] is a measure of brokerage or information flow. A *geodesic* as a shortest path between two components in a network. Betweenness is the number of geodesics that a particular component lies on. In standard SNA, this measure quantifies the degree to which an individual in a network mediates information flow, and thus a measure of social status. A component may have few links, but high betweenness, if a component acts as a bridge between two otherwise disconnected groups of components. This may represent a software component that acts as a point of contact or an interface. It could also represent a component that is contributed to by a developer who works on components in disparate parts of the system (*e.g.* a developer who makes minor contributions to the gui, but works mainly on the ECLIPSE compiler and static analysis). In either case, if many paths of relationships "flow" through a component, this indicates that many components and/or developers have an interest in the component.

Closeness centrality [7] measures the distance from a component to all other components (and possibly developers) in the network. Lower values indicate that the component is farther away from all other nodes.

Reachability [17] is a similar measure to closeness in that it uses the geodesics from a node to all other nodes. Higher values indicate a shorter average distance to other nodes in the network.

Eigenvector Centrality [8] is another measure of the importance of a component in a network. It is similar to Google's PageRank [18] in that connections to high valued nodes increase a node's value more than connections to low valued nodes.

Bonacich Power [9] measures centrality of an component based on the centrality of other nodes. A node may be considered central if it is connected to nodes that have connection to *many* other nodes. A node may be considered *powerful* if it is connected to nodes that have connections to *few* other nodes. Binaries that are more central may be more likely to have post-release failures. We use a positive Beta value of 0.2.

Structural Holes are gaps in anetwork. If a component, A has a connection to a neighbor, B, that no other neighbors are connected to, then A is in a more powerful position over B than the other neighbors. The absence of an edge between B and A's other neighbors represent structural holes. The following measures quantify properties of structural holes. We refer the reader to [10] for further detail.

- **Effective Size** - The number of components that are connected to a component minus the average number of edges between these components.
- **Efficiency** - Normalizes the effective size of the network by the total size of the network.
- **Constraint** - Measures how strongly a component is constrained by it's neighbors. The idea is that neighbors that are connected to other neighbors can constrain a component.
- **Hierarchy** - Quantifies constraint above is distributed across neighbors. When most of the constrain comes from a single neighbor, the hierarchy is higher.

Local Measures

Degree centrality [7] is a basic network measure. In an undirected unweighted network, degree is simply the number of edges incident upon an node. Weighted networks use a sum of the weights of the edges and directed networks include in-degree, and out-degree based on edge direction.

Ego network measures [17] are based on the neighborhood for any particular node. The node being evaluated is denoted ego, and the neighborhood includes ego, the set of nodes connected to ego an edge, and the complete set of edges between this set of nodes. The set of nodes connected can be chosen in the following three ways:

- **In-neighborhood** - nodes that have an edge directed towards ego
- **Out-neighborhood** - nodes that have an edge directed away from ego towards them
- **InOut-neighborhood** - nodes that have either of the above

We create all three ego networks for each node and compute the following ego network measures:

- **Size** - The number of nodes in the ego network
- **Ties** - Number of edges in the ego network
- **Pairs** - Number of possible directed edges in the ego network
- **Density** - Proportion of possible ties that actually are present (Ties/Pairs)
- **Weak Components** - Number of weakly connected components
- **Normalized Weak Components** - Number of weakly connected components normalized by size, i.e., (Weak Components/Size)
- **Two Step Reach** - The proportion of nodes that are within two hops of ego
- **Reach Efficiency** - Two Step Reach normalized by size of the network. Higher reach efficiency indicates that ego's primary contacts are influential in the network.
- **Brokerage** - Number of pairs of nodes that are connected only by ego. Thus ego acts as the sole broker for the pair
- **Normalized Brokerage** - Brokerage normalized by number of pairs
- **Ego Betweenness** - Betweenness of ego within its ego network

Table 1: Correlation of some network metrics with number of bugs in ECLIPSE 3.3 in each of the three networks. In all cases shown except eigenvector centrality the socio-technical metrics had higher values to a statistically significant degree. This is true of the majority of network metrics. The only metrics out of all that had a significantly higher value than socio-technical was eigenvector centrality.

- **Normalized Ego Betweenness** - Ego Betweenness normalized by size of the network

Correlation with Failures

As a preliminary study of Windows Vista and Eclipse, we examined the correlation of all of the above described SNA measures on the three graphs with failures. In over 90% of the cases, the SNA measures for the socio-technical network had higher correlations with post-release failures than the dependency and contributions networks to a statistically significant degree. We found only one case, *eigenvector centrality*, where a metric on a non-socio-technical network had a higher correlation at a statistically significant level. Due to space limitations, a comprehensive listing of correlations is prohibitive. We present a sample of these correlations in ECLIPSE 3.3 in table 1. Since the metric values were not normally distributed, a spearman rank-correlation was used. In both Windows Vista and ECLIPSE, the SNA measures on the socio-technical networks have much higher levels of correlation with failures than code complexity metrics such as number of functions, class hierarchy depth, lines of code, or cyclomatic complexity. This initial result is encouraging that combining software component relationships will increase the predictive power of defect prediction models.

Based on the above observations, conjectures, and preliminary results we state the following research hypotheses.

Hypothesis 1 - The role of a software component in the dependency network *and* its role in the developer contribution network together influence defect proneness.

Hypothesis 2 - Software components that play key roles in the joint socio-technical network are more prone to defects than those that don't.

Note the difference in these hypotheses. The first examines the roles played by a component in two networks and uses information from both. The second looks at a component's properties in the aggregate socio-technical network. We evaluate these hypotheses on two large software systems in the following sections.

4. Projects and Data Collection

In an effort to evaluate our approach in multiple contexts, we gathered data from two large software engineering efforts: one commercial, and one open-source project: Windows Vista and the ECLIPSE integrated development environment and examined post-release defects in these

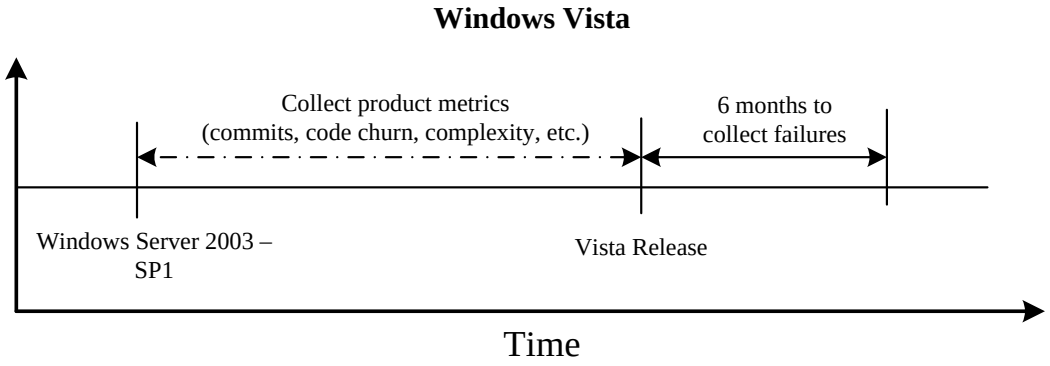


Figure 2: Timeline for Windows Vista data collection

systems. We use post-release failures because they are the most problematic. They clearly were not identified by pre-release inspection, testing or analysis tools, directly affect end-users, and are the most expensive to correct. Both systems have a long development history with large software teams. In addition, each can be decomposed into a system of software components: binaries (exe's, dll's, etc.) in the case of Vista and plugins for ECLIPSE. However, there are important differences between these projects.

Process Differences: Although backed by IBM, ECLIPSE is an open source project (and thus represents an OSS-hybrid project [19]) which accepts contributions from volunteers. These unpaid volunteers can be assigned tasks. However, there are no monetary or employment repercussions for not performing tasks (though there are "reputation" repercussions). As an open source project, most developers communicate via written electronic channels such as irc, mailng lists, and a bug tracking system. Developers can come and go at will. Vista, on the other hand, is developed completely in-house at Microsoft. The software teams that developed Vista were clearly delineated and largely static. In addition, most software teams are geographically collocated and therefore enjoy face-to-face interaction.

Domain Differences: Windows Vista is an operating system. Thus it performs tasks ranging from low-level operations such as scheduling read sequences for hard disks to high-level operations such as displaying error messages to users. ECLIPSE is a Java integrated development environment and has a narrower range of functionality, although it ranges from static analysis and dynamic compilation to graphical user interfaces and source code management network protocols.

Language Differences: The majority of ECLIPSE is written in one programming language: Java. While mostly portable via use of the JVM, ECLIPSE still needs to deal with multiple platforms (Linux, OS X, Windows). Windows Vista is a combination of C, C++, assembly and .NET managed code (mostly C#). As an operating system, it runs directly on the hardware, but does support a variety of hardware devices and configurations.

Similarities: Both software systems are extensible. The ECLIPSE architecture is very flexible and allows developers to add functionality in the form of *plugins* which are dependent on base functions performed by the core ECLIPSE code. Hardware vendors and third parties extend Vista through the development of drivers, additional libraries, and binaries, all of which are also dependent on core functionality provided by the operating system.

Data collection: Windows Vista

We use a *binary* as the level of granularity for software components in Windows Vista. A binary is an executable (.exe), a library (.dll), or a driver (.sys). Vista comprises over 4,000 binaries. Microsoft collects software failure data at the granularity level of binaries.

Our Windows Vista development data is drawn from the Vista source code repositories for all activity prior to release. A few thousand engineers contributed to the source code for Windows Vista during the development phase. We do not include changes to non-source portions of source files that are updated for building.

We measure post-release failures, on a per binary basis, for the six months following release, as shown in figure 2.

We also identify dependencies between binaries in Vista. Microsoft has developed an automated tool called MaX [20] that tracks dependency information at the function level, including calls, imports, exports, RPC, COM, and registry access. We used MaX to generate a system-wide dependency graph from both x86 and .NET managed libraries. We lift the dependencies from the function level to the binary level because our measure of failures is mapped to individual binaries.

Data collection: ECLIPSE

We mined development, dependency and defect data from the ECLIPSE project spanning from from 2001 to 2008. We focus on the 6 major releases for which we have complete pre-release development data and post-release bugs for (2.0 – 3.3). The CVS logs contain the developer contribution data used to construct the contribution network. We also gathered defect data in the form of bug records from the ECLIPSE bugzilla database and "linked" bugs to their source code

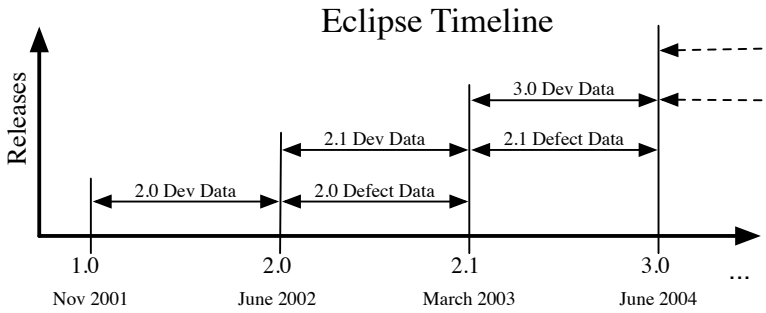


Figure 3: Data collection timeline for ECLIPSE

introduction by using techniques such as examining the log messages to find bug fixing changes [21]. We restrict the set of bugs to those that were opened after the release date for each release. As shown in figure 3, for each release R_i, we examine the development activity from the date of release for R_{i-1} to the date of release for R_i.

We use *plugins* as the level of granularity for software components (specifically, jars within plugins, as some plug-ins are composed of multiple jars) in ECLIPSE because the quality of the defect data is not as reliable at the source file level due to the low number of defects that most source code files have. In addition the majority of bug fixes are tied to multiple source files, but the same plugin. The number of plugin jars per release ranged from 90 in 2.0 to 250 in 3.3. We used only the plugins that are part the ECLIPSE project and exist in the ECLIPSE CVS repository.

We obtained the source files for each of the releases from the public ECLIPSE CVS repository and used the static analysis tool **Understand** from SciTools[1] to identify program dependencies. The dependencies are determined at the class level, and we use a mapping from classes to files and then to plugins to determine plugin dependencies. A class A may depend on a class or interface B if A inherits or implements from B, has a field of type B, calls a method in B, imports B, or creates an instance of B.

We calculate the dependencies between all classes in ECLIPSE and then lift the dependencies to the plugin level.

We used UCINET [17] to calculate each of the social network analysis measures described in section 3.2 on the contribution and dependency data collected for Windows Vista and each ECLIPSE release our study.

5. Methods and Analysis

In this section, we describe our data collection methods and analysis techniques.

5.1. Logistic Regression

We used logistic regression to examine the relationship between social network analysis metrics and post-release failures. Logistic regression is used to produce an estimated probability that a binary dependent variable will have a particular value. In our case, we categorize binaries into failure prone and not failure prone based on their number of post-release failures. We use social network metrics as predictor variables in the logistic model to predict if a binary is failure prone or not. One of the problems encountered when using network metrics within the model is *multicollinearity*. Use of logistic regression with multiple predictor variables makes an assumption that the predictors are all independent. In practice, we found that some measures have high levels of correlation. For instance, many binaries with high betweenness also have high degree. Unconstrained use of correlated predictor variables in regression models results in highly overtrained models with low quality parameter estimates and poor predictive power on new data [22].

We use principal component analysis (PCA) [23] to rectify this problem. PCA transforms the predictors' values for the set of training observations into a new set of observations in orthogonal dimensions (principal components) with no covariance. Each principal component is a linear combination of the predictor variables and the components are ordered by the amount of variance in the initial predictors that they capture. To avoid overfitting, we use only the minimum number of principal components that capture 95% of the variance in observations.

We train logistic models using four data sets for each software system: measures on the dependency network, the contribution network, all of the measures from both networks, and the measures from the socio-technical network.

We evaluate the ability of network measures to identify failure prone software components in two ways. As with prior work, for a particular software system, we randomly select two thirds of the software components to train the logistic prediction model. The model then predicts which of the components in the remaining one third are fault prone and the predictions are evaluated using the standard information retrieval (IR) measures of precision, recall, F-score, and ROC curve [24]. This procedure is repeated 50 times with different random splits and the mean of each IR metric is reported. We can determine if one data set yields better predictive power than another for a particular software system by performing a standard t-test on the IR results for both data sets.

In practice, it is not possible to use a predictive model in this way because it requires that you already have the

1. http://www.scitools.com/products/understand

Release	Network	Precision	Recall	F-Score	Nagel.
Vista	Dependency	0.707	0.547	0.617	0.284
	Contribution	0.774	0.650	0.706	0.506
	Combined	**0.787**	**0.681**	**0.730**	0.504
	Socio-technical	0.769	**0.705**	**0.736**	**0.520**
2.0	Dependency	0.667	0.779	0.705	0.532
	Contribution	0.808	0.854	0.824	0.702
	Combined	0.826	0.814	0.813	**0.909**
	Socio-technical	0.755	0.859	0.800	**0.747**
2.1	Dependency	0.693	0.753	0.710	0.626
	Contribution	0.675	0.780	0.719	0.607
	Combined	**0.755**	0.777	**0.758**	**0.805**
	Socio-technical	**0.747**	**0.809**	**0.770**	**0.689**
3.0	Dependency	0.631	0.737	0.673	0.494
	Contribution	0.681	0.683	0.673	0.353
	Combined	**0.745**	0.756	**0.743**	**0.616**
	Socio-technical	**0.767**	**0.777**	**0.769**	**0.600**
3.1	Dependency	0.579	0.718	0.634	0.391
	Contribution	0.639	0.646	0.629	0.295
	Combined	**0.693**	**0.796**	**0.735**	**0.689**
	Socio-technical	**0.820**	**0.800**	**0.806**	**0.668**
3.2	Dependency	0.698	0.780	0.731	0.495
	Contribution	0.614	0.720	0.654	0.371
	Combined	**0.835**	**0.866**	**0.846**	**0.816**
	Socio-technical	**0.792**	0.784	**0.785**	**0.572**
3.3	Dependency	0.693	0.743	0.711	0.433
	Contribution	0.725	0.669	0.688	0.356
	Combined	0.742	**0.780**	**0.754**	**0.686**
	Socio-technical	**0.820**	**0.831**	**0.823**	**0.727**

Table 2: Results of 50 random splits on one release. Bold values indicate that they are higher than the contribution and dependency networks to a statistically significant degree.

classification of two thirds of the software components *and* that the two thirds represent a random sample. Rather, one would expect to train a fault prediction model on one release of a software system to predict the failure prone binaries in the next release. The data from six releases of ECLIPSE can be used to evaluate our approach in exactly the way that a practitioner would use it.

Evaluating Regression Results

We use five IR measures to assess the quality of the predictive models: precision, recall, the F score, Nagelkerke coefficient of determination, and area under the ROC curve. All measures range from 0 to 1 with higher values indicating better performance.

Precision quantifies the type I errors by measuring the proportion of binaries that were classified as failure prone that actually were observed to be failure prone. Precision is calculated as follows:

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives}+\text{false positives}}$$

Recall measures the proportion of binaries that are actually failure prone that are classified as failure prone (type II errors). This is calculated as:

$$\text{Recall} = \frac{\text{true positives}}{\text{true positives}+\text{false negatives}}$$

These two measures present a tradeoff as it is possible to sacrifice one to improve the other. A traditional method of assessing both precision and recall is the use of a metric known as the *F score*. It is calculated as the harmonic mean of the precision and recall for a particular model.

$$\text{F score} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision}+\text{recall}}$$

Release	Network	AUC	Precision	Recall	F-Score	Nagel.
2.1	Dependency	0.867	0.750	0.688	0.717	0.505
	Contribution	0.867	0.596	0.918	0.723	0.685
	Combined	0.804	0.618	0.850	0.716	**0.860**
	Socio-technical	**0.889**	**0.775**	0.775	**0.775**	**0.721**
3.0	Dependency	0.809	0.789	0.625	0.698	0.598
	Contribution	0.795	0.764	0.688	0.724	0.581
	Combined	**0.830**	0.754	**0.708**	**0.730**	**0.699**
	Socio-technical	**0.864**	0.753	**0.761**	**0.757**	**0.668**
3.1	Dependency	0.776	0.691	0.667	0.679	0.448
	Contribution	0.744	0.656	0.716	0.685	0.313
	Combined	**0.860**	**0.732**	**0.732**	**0.732**	**0.540**
	Socio-technical	**0.878**	**0.765**	**0.815**	**0.789**	**0.579**
3.2	Dependency	0.810	0.734	0.663	0.697	0.376
	Contribution	0.775	0.673	0.750	0.710	0.260
	Combined	**0.910**	**0.900**	**0.759**	**0.824**	**0.623**
	Socio-technical	**0.866**	**0.766**	**0.847**	**0.805**	**0.642**
3.3	Dependency	0.804	0.759	0.710	0.733	0.471
	Contribution	0.765	0.654	0.609	0.631	0.351
	Combined	**0.857**	0.684	**0.870**	**0.766**	**0.782**
	Socio-technical	**0.927**	**0.827**	**0.847**	**0.837**	**0.558**

Table 3: Results of training a model on data from release $r-1$ to predict failure prone components in release r.

Nagelkerke coefficient of determination for a logistic regression is similar to the R^2 coefficient of determination in a linear regression model in that it measures explained variation and predictive discrimination.

Area under ROC curve (AUC): Receiver operating characteristic (ROC) curves are a non-parametric way to evaluate 2-class discriminant models. The curve plots the true positive rate against the false positive rate. The ideal discriminant has a 100% true positive rate, a zero false positive rate; random guessing essentially yields a diagonal line. The area under this curve (AUC) provides a measure of the quality of the discriminant function. For more details, see [25].

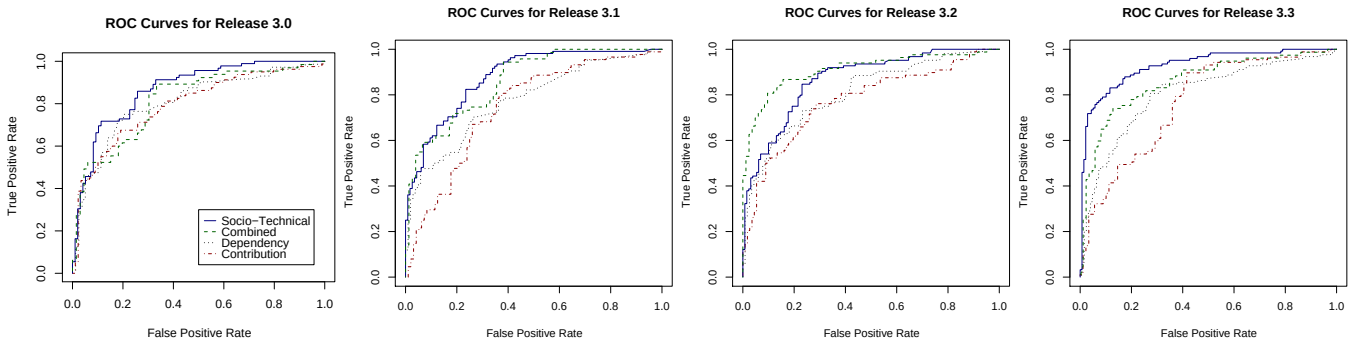
6. Results

We now discuss the results of using the above described analysis on our data.

In order to compare the predictive accuracy of the metrics, we calculated the values of recall, precision, F score, and Nagelkerke coefficient of determination for logistic regression on 50 random splits on each model for Vista and ECLIPSE. Table 2 shows the averages for each of these values per model. Bold values indicate that a wilcoxon test found the values for that model to be statistically higher than the dependency and contribution models at the $p < .05$ level. P-values were adjusted using Benjamini Hochberg adjustment for multiple hypothesis testing [26].

Vista Results

For our analysis on Windows Vista, both the combined and socio-technical models have better recall than the dependency and contribution models. This indicates that they have a lower false negative rate, i.e. are able to detect failure prone binaries better. Only the combined model has statistically significantly higher levels of precision. The socio-technical model has a similar number of false positives, i.e. non-failure prone binaries that are classified as failure prone, as the contribution model. In addition the F score of the



(a) Release 3.0 (b) Release 3.1 (c) Release 3.2 (d) Release 3.3

Figure 4: ROC curves for prediction models trained on data from release $r-1$ to predict failure prone components in release r. The blue solid line is Socio-technical, green dashed line is Combined, red dot-dash line is Contribution, and black dotted line is Dependency.

combined and socio-technical models both exceed that of the dependency and contribution models. Lastly, the Nagelkerke coefficient of determination is higher in the socio-technical model. This indicates that the socio-technical model best captures the variance in the number of post-release failures in the binaries. We note that neither the dependency network model nor the contribution network model were superior to *either* the combined or socio-technical models for any of the evaluation metrics.

The recall for the combined and socio-technical models in Vista exceeds the previous work by 3% and 5% respectively which is substantial given the thousands of binaries that shipped in Windows Vista.

Based on these findings, we conclude that *in the case of Windows Vista, hypotheses 1 and 2 are supported.*

ECLIPSE results

The results for our random splits within ECLIPSE are also shown in table 2. With the exception of release 2.0, the combined and socio-technical models perform markedly better than the dependency and contribution models. The improvement in this case is more dramatic. For instance, the increase in f-score exceeds 8% in all cases after release 2.0. The Nagelkerke coefficient is much better for both the combined and socio-technical models in every release of eclipse. As with Vista, there is no evaluation metric in any release in which the dependency or contribution models perform statistically better that the combined or socio-technical models. We therefore conclude that *with the exception of release 2.0, hypotheses 1 and 2 are supported for the* ECLIPSE *project.*

Prediction Across Releases

We also evaluated our approach more realistic setting, mirroring the way a prediction model might be used in an actual system history, using older releases to predict fault-prone components in a new release. We built logistic prediction models on release $r-1$ in order to predict fault prone plugins in release r. This type of predictive modeling is complicated by changing network size. As an example, betweenness of a node is based on the number of geodesics that a node lies on. As a network's size increases, the number of geodesics increases quadratically. Betweenness may be normalized by dividing all values by the maximum possible betweenness, but in practice, this over-inflates the betweenness for nodes in small networks. We overcome this practice through the use of *standard scores* (also known as z-scores) [27]. For every metric, m in a particular release, we standardize it by subtracting the mean from each observation, i, to center it around zero and dividing the result by the standard deviation.

$$z_{mi} = \frac{x_{mi} - \mu_m}{\sigma_m}$$

The distribution of the result always has mean 0 and standard deviation 1. We can then compare values of network metrics on software components in different networks more easily. Two binaries from different releases that have betweenness values two standard deviations higher then the mean will both have a standard score of 2.0. These standard scores are used to build the logistic prediction models as described in section 5.1.

Table 3 contains the results. We do not show results for release 2.0 because we didn't have access to development and defect data for the 1.0 release. In this case, there was no random splitting, so there was no repeated model building and we cannot claim that a particular model performed better than another to a *statistically significant degree*. Rather the evaluation metrics in which for the combined or socio-technical models which had superior results than *both* the dependency and contribution models are shown in bold. In a further effort to illustrate the difference in performance between the models, we show the ROC curves for the models across releases in figure 4. Note that with the exception of the Combined model in release 2.1, both the combined and socio-technical models outperform the other models.

This is an important result in that it demonstrates that when our approach is used in a real-world setting as practitioners would use it, it continues to perform well.

7. Conclusion

In this paper we have shown that the topological properties of software component networks can be used to identify which components will have the most post-release failures. We further concluded that when multiple types of relationships are used in predictive models (dependency and contributions in our case), the predictive power is increased. We have evaluated our improved techniques in both a standard industrial setting in the case of Windows Vista and in an OSS context in the case of ECLIPSE— products in entirely different domains. This is evidence that the approach of using socio-technical networks to predict failure prone binaries is not process or domain specific and gives strong external validity to our approach. Further, we have shown that our approach is useful to practitioners in that defect prediction models can be trained on one release in order to be used in the next.

References

[1] "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology (NIST) Planning Report 02-3, May 2002, http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[2] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2008, pp. 2–12.

[3] T. Zimmermann and N. Nagappan, "Predicting subsystem failures using dependency graph complexities," pp. 227–236, 2007.

[4] T. Zimmermann and N. Nagappan, "Predicting defects using social network analysis on dependency graphs," in *Proc. of the International Conference on Software Engineering*, 2008.

[5] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 3rd International Workshop on Mining Software Repositories*, 2006.

[6] S. Wasserman and K. Faust, *Social network analysis: Methods and applications*. Cambridge University Press, 1994.

[7] L. C. Freeman, "Centrality in social networks I. Conceptual clarification," *Social Networks*, vol. 1, pp. 215–239, 1979.

[8] B. Ruhnau, "Eigenvector-centrality – a node-centrality?" *Social Networks*, vol. 22, no. 4, pp. 357 – 365, 2000.

[9] P. Bonacich, "Power and centrality: A family of measures," *The American Journal of Sociology*, vol. 92, no. 5, pp. 1170–1182, 1987. [Online]. Available: http://www.jstor.org/stable/2780000

[10] R. S. Burt, *Structural holes: The social structure of competition*. Cambridge, MA: Harvard University Press, 1995.

[11] 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*. ACM, 2008.

[12] C. Amrit, J. Hillegersberg, and K. Kumar, "A Social Network Perspective of Conway's Law," in *Proceedings of the CSCW Workshop on Social Networks, Chicago, IL, USA*, 2004.

[13] G. Valetto, M. Helander, K. Ehrlich, S. Chulani, M. Wegman, and C. Williams, "Using software repositories to investigate socio-technical congruence in development projects," *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, pp. 25–25, 2007.

[14] M. Cataldo, P. Wagstrom, J. Herbsleb, and K. Carley, "Identification of coordination requirements: implications for the Design of collaboration and awareness tools," *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pp. 353–362, 2006.

[15] 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 *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. New York, NY, USA: ACM, 2008, pp. 2–11.

[16] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr., "N degrees of separation: Multi-dimensional separation of concerns," in *International Conference on Software Engineering*, 1999, pp. 107–119. [Online]. Available: citeseer.nj.nec.com/tarr99degrees.html

[17] S. Borgatti, M. G. Everett, and L. C. Freeman, "UCINET 6 for Windows: Software for Social Network Analysis. Harvard, MA, Analytic Technologies," 2002.

[18] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford University, Tech. Rep., 1998.

[19] J. Berkus, "The 5 types of open source projects," 2007, march 20, 2007 http://www.powerpostgresql.com/5_types.

[20] A. Srivastava, J. Thiagarajan, and C. Schertz, "Efficient Integration Testing using Dependency Analysis," Microsoft Research, Tech. Rep. MSR-TR-2005-94, 2005.

[21] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*. New York, NY, USA: ACM, 2005, pp. 1–5.

[22] D. E. Farrar and R. R. Glauber, "Multicollinearity in regression analysis: The problem revisited," *The Review of Economics and Statistics*, vol. 49, no. 1, pp. 92–107, 1967. [Online]. Available: http://www.jstor.org/stable/1937887

[23] J. Jackson, *A user's guide to principal components*. Wiley-Interscience, 2005.

[24] F. W. Lancaster, *Information Retrieval Systems: Characteristics, Testing, and Evaluati on*, 2nd ed. Wiley, 1979.

[25] A. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

[26] Y. Benjamini and Y. Hochberg, "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[27] S. Dowdy, S. Wearden, and D. Chilko, *Statistics for research*, 3rd ed. John Wiley & Sons, 2004.