Predicting Failures with Developer Networks and Social Network Analysis

Andrew Meneely[1], Laurie Williams[1], Will Snipes[2], Jason Osborne[3]
[1]Department of Computer Science, North Carolina State University, Raleigh, NC, USA
{apmeneel, lawilli3}@ncsu.edu
[2]Nortel Networks, Research Triangle Park, NC, USA. wbsnipes@nortel.com
[3]Department of Statistics, North Carolina State University, Raleigh, NC, USA
jaosborn@ncsu.edu

ABSTRACT
Software fails and fixing it is expensive. Research in failure prediction has been highly successful at modeling software failures. Few models, however, consider the key cause of failures in software: people. Understanding the structure of developer collaboration could explain a lot about the reliability of the final product. We examine this collaboration structure with the developer network derived from code churn information that can predict failures at the file level. We conducted a case study involving a mature Nortel networking product of over three million lines of code. Failure prediction models were developed using test and post-release failure data from two releases, then validated against a subsequent release. One model's prioritization revealed 58% of the failures in 20% of the files compared with the optimal prioritization that would have found 61% in 20% of the files, indicating that a significant correlation exists between file-based developer network metrics and failures.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics – *process metrics, product metrics*.

General Terms
Reliability, Human Factors, Verification

Keywords
Social network analysis, negative binomial regression, logistic regression, failure prediction, developer network

1. INTRODUCTION
Software fails and fixing it is expensive. If testers can find software failures early in the software development lifecycle, the estimated cost of fixing the software dramatically decreases [10]. Research in failure prediction has provided many models to assess the failure-proneness of files, and have been highly successful at predicting software failures [3, 8, 11, 21, 22, 24, 25, 28].

Few models, however, consider the key cause of failures in software: people. People develop software and people test software. For large software systems, many people need to work together to develop software. This collaboration has a structure – a structure governed by elements of human social interaction and software development processes. Understanding the structure of developer collaboration could tell us a lot about the reliability of the final product.

We examine this collaboration structure using a software development artifact common to most large projects: code churn information taken from revision control repositories. Code churn information has provided valuable metrics for failure prediction [21]. For example, a file with many recent changes tends to be more failure-prone than an unchanged file.

But what if that file was updated by a developer who has worked with a lot of other developers? Maybe a "well-known" developer is less failure-prone. Code churn information can also tell us how these developers collaborated: we know who worked on what and when. From there, we can form a social network of developers (also known as a developer network) who have collaborated on the same files during the same period of time. Social Network Analysis (SNA) quantifies our notion of "well-known" developers with a class of metrics known as "centrality" metrics.

The advantage of this developer network is that it provides a useful abstraction of the code churn information. With careful interpretation, one can use a developer network mid-development to identify potential risks and to guide verification and validation (V&V) activities such as code inspections.

Our research goal is to examine human factors in failure prediction by applying social network analysis to code churn information. Failure prediction models have been successful for other areas (such as static analysis [16]), so the empirical techniques of model selection and validation have all been used with static code metrics [20]. We introduce file-based metrics based on SNA as additional predictors of software failures.

A case study was conducted of a large Nortel networking product consisting of over 11,000 files and three million lines of code to build and evaluate the predictive power of network metrics. System test and post-release failure data from Nortel's source repositories and defect tracking system were used in our study.

The rest of this paper is organized as follows: Section 2 summarizes the background of Social Network Analysis and related work in failure prediction and developer networks. Section 3 introduces our developer networks, their associated metrics, and the analysis in failure prediction. Sections 4 and 5 summarize our case study of the Nortel product. Sections 6 and 7 summarize our work and outlines future work, respectively.

SIGSOFT 2008/FSE-16, November 9--15, Atlanta, Georgia, USA

2. BACKGROUND AND RELATED WORK

In this section, we introduce the network metrics we will use in our failure prediction model. We also present network analysis and summarize fault/failure prediction models most similar to ours in terms of using developer information or in terms of statistical analysis.

2.1 Definition of Network Metrics

In this paper, we use several terms from network analysis [1, 6, 26] and define their meaning with respect to developer networks in Section 3.1. In network analysis, vertices of a graph are called **nodes**, and edges are called **connections**. A sequence of non-repeating, adjacent nodes is a **path**, and a shortest path between two nodes is called a **geodesic path**. Informally, a geodesic path is the "social distance" from one node to another. The longest geodesic path of a network is called the network's **diameter**.

2.1.1 Connectivity

Metrics that measure a node's direct connections to other nodes are **connectivity** metrics. The primary connectivity metric in our study is the **degree** of a node**.** Degree is the number of connections incident on a node. The degree of a node in a random network is modeled by the Poisson distribution [6], which is useful for determining if a node is a hub. A node is considered a **hub** if its degree is above a given threshold calculated from the inverse Poisson cumulative distribution function for a p-value less than 0.01[1]. A node is considered to be **disconnected** if it has no edges.

2.1.2 Centrality

Centrality metrics quantify how closely nodes are indirectly connected to other nodes in the network. Centrality can be measured by two metrics: closeness and betweenness. The **closeness** of node v is defined as the average distance from v to any other node in the network that can be reached from v. Formally, the closeness D_c of node v in graph G is defined as

$$D_c(v) = \left(\frac{1}{|V(G,v)|}\right)\sum_{t \in G} d_G(v,t) \quad (1)$$

where $d_G(v,t)$ is the distance (number of edges) from node v to node t and $|V(G,v)|$ is the number of nodes in the graph reachable from v.

The **betweenness** of node v is defined as the number of geodesic paths that include v divided by the total number of geodesic paths in the network. Formally, betweenness B_c of node v in graph G is defined as

$$B_c(v) = \sum_{s,t,v \in G, s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad (2)$$

where $\sigma_{st}(v)$ is the number of geodesic paths from s to t going through v, and σ_{st} is the total number of geodesic paths from s to t.

[1] Note that not all networks' degrees follow a Poisson, meaning some networks have more than 1% of their nodes being hubs.

2.2 Failure Prediction

The closest research relating to ours is the fault prediction model based on developer information proposed by Weyuker et al. [25] These researchers examined various releases of a large industrial software system to predict which files are most likely to contain the largest number of faults. Inspection guidance and automated testing efforts are among the applications intended for their fault prediction model. Their model is based on the negative binomial distribution and their model's variables, based on developer information, attempt to capture information about the amount and the type of developers who have worked on any given file. Validation for their model included a comparison with a working model based on static code metrics and churn information. Weyuker et al. reported finding 84.9% of the faults in 20% of the files with the developer information, where without the developer information, 83.9% of the faults were found. The amount of failures found using the optimal prioritization was not mentioned. Our models use some similar developer counts in combination with network metrics to predict failures.

Zimmerman and Nagappan [27] applied network analysis to dependency graphs for predicting failures in files. By applying metrics of centrality and network motifs to the directed dependency graphs of source code, the researchers found that central components were more failure-prone. Furthermore, network metrics proved to identify 60% of the critical, failure-prone binaries, which was better than object-oriented complexity metrics that only identified 30%. In addition to using centrality metrics of closeness and betweenness, Zimmerman and Nagappan used similar statistical regression techniques for their analysis that we used.

Mockus and Weiss [18] used metrics based on developer information for failure prediction to assess risk in a large industrial software system. Developer metrics included counts of distinct developers and a quantitative measurement of developer experience in terms of recent changes of the current project, experience in the subsystem, and in the product overall. They used step-wise variable selection to construct a logistic regression model for estimating post-release failures. Our models do not quantify developer experience, however, a discussion of experience can be found in Section 4.4.3.

Hudepohl et al. [16] used developer information in combination with various other metrics to create a risk assessment tool at Nortel called EMERALD. The developer information was a measurement of experience similar to the variables used by Mockus and Weiss. EMERALD's developer variables, however, incorporated developer experience in terms of Nortel career, as opposed to specific projects. For example, one of the experience measurements was the count of the number of developers who were within their first ten code updates while working at Nortel as a way to identify inexperienced developers. EMERALD's other variables included complexity metrics, customer usage metrics, churn information, and past failure counts from both testing and post-release phases. Hudepohl et al. reported that over half of the field failure patches were correctly identified as "red" (highest risk) in 20% of the files.

Arisholm and Briand [3] identified developer experience and skill level as fundamental factors affecting fault-proneness in an object-oriented system. Since they had no data on skills and experience of developers, they did not consider developer information in their model. Nonetheless, they used a stepwise

logistic regression model and a cross-validation classification analysis to validate their results. Most of the variables in their model could be classified in the categories of object-oriented metrics and code churn information. Their results from cross-validation analysis showed less than 20% false positives and false negatives, with an estimated verification effort savings of 29%. We used developer information in our model, however, not based on skill but on the structure of developer connections within the developer network.

Arisholm et. al [4] examined several data mining techniques used for fault prediction and validated their work on a large telecommunications product. The authors also discuss techniques of data collection, model selection, and model validation. Some of the discussed data mining techniques include logistic regression, neural networks, and decision trees. We applied similar techniques in the area of model selection and validation.

Nagappan and Ball [21] used metrics based on code churn data to predict defect density in Windows Server 2003. Their hypothesis was on comparing the predictive power of relative code churn metrics to absolute code churn metrics. A relative code churn metric, as defined by Nagappan and Ball, is one that is normalized by parameters such as lines of code, files counts, etc. Multiple linear regression, Principle Component Analysis, and step-wise variable selection were all used to make predictions about defect density. Data splitting was used to validate the predictive power of the chosen model and to show that relative code churn metrics are more powerful than absolute code churn metrics. Along with the use of code churn metrics, similar statistical techniques to ours were used, such as multiple linear regression, logistic regression, and step-wise variable selection.

2.3 Network Analysis

The idea of constructing a developer network based on source repository information is not new [14, 15, 23]. However, the studies in SNA in software engineering have been directed toward studying communication and learning, not to do failure prediction, as in our case.

Gonzales-Barahona and Lopez-Fernandez [14] propose the idea of creating developer networks from source repositories as a method of characterizing projects. Their main focus was to organize Open Source projects into various categories based on models of collaboration. The developer networks that Gonzales-Barahona and Lopez-Fernandez propose are constructed in a similar manner as ours, except that the edges of the graph are weighted based on number of files the pair has collaborated on. The idea of weight in their network introduces variations on the centrality and connectivity metrics, such as a "clustering coefficient". In addition to a developer network, they used a module network – where two modules were connected if they were committed together. We decided on using a non-weighted developer network for simplicity in our study, however we will pursue more sophisticated network analysis such as those presented by Gonzales-Barahona and Lopez-Fernandez in our future work (see Section 7).

Huang and Liu [15] used SNA based on source repositories to examine the learning process in Open Source projects. Their primary analysis involved using Legitimate Peripheral Participants, a network-based theory proposed by Lave and Wenger [17]. Huang and Liu concluded that developers could be divided into core and non-core groups, which loosely affected a "project's vitality and popularity" [15].

One significant difference between our study and other studies of developer networks is that ours is based on a proprietary product. Intuitively, collaboration in an Open Source project will be much different than that of a closed-source product. For example, companies have much more control over the organization of their developers. Network Metrics based on proprietary products are possibly more related to the development process used by the company than to the nature of human collaboration.

3. SNA-BASED FAILURE PREDICTION

In this section, we present our approach for the selection and validation of SNA-based failure prediction models. We utilize this approach with data from a Nortel product, as described in Section 4. The product of our model is a prioritization (ordering) of files so that developers and testers can guide verification and validation activities such as code inspections.

3.1 Developer Network Metrics

A developer network is an estimation of the structure of collaboration in a software development project. We define a social network based on developer connections within a software development project. In our developer network, two developers are connected if they have both made a change to at least one file in common during the same release. The result is an undirected, simple graph where each node represents a developer and edges are based on whether or not they have worked on the same file during the same release.

SNA provides quantitative measures of the structure of a network. The goal of performing network analysis using developers as nodes is to quantitatively determine where a developer lies in the overall structure of the network. Informally, we are trying to quantify how "well-known" a developer is in the context of the project. A "well-known" developer, for instance, might have many *direct* connections, that is, a developer is directly connected to many other developers. Metrics that measure developers' direct connections to others are **connectivity** metrics, which are described in subsection 2.1.1 (i.e. degree, hub, and disconnected). Alternatively, a "well-known" developer may also, for instance, be connected to other developers who are connected to many other developers, and so on. That is, a developer may be "well-known" by how closely connected he or she is by *indirect* connections (by geodesic paths greater than one). Metrics that measure how developers are indirectly connected to the rest of the network are **centrality** metrics, which are described in subsection 2.1.2 (closeness and betweenness).

Each metric captures a different aspect of a developer's place in the network. A developer's **degree** is equal to the number of other developers he or she worked on source files with. A **hub** is a developer with a high degree. A **disconnected** developer is the sole modifier of the files he or she updated in a release. A developer with high **betweenness** is generally more central to the network, as a central developer would lie on more geodesic paths than a non-central developer. A developer with low **closeness** means that their average social distance is low, implying he or she is well-known.

Other researchers [14, 26] who have defined developer connections similarly have referred to these networks as "collaboration networks." Although developer networks do provide evidence of possible collaboration between developers, we are hesitant to use such a term for our research without emphasizing that we are estimating collaboration via churn-based

metrics. The asynchronous and often remote nature of working through a repository may imply that not all collaboration is being captured by our developer network. Actual social relationships, geography, and communication are not explicit factors in our network, only the system's historical records. Two developers working on the same file around the same time, however, indicates that a possible collaboration is taking place. We propose, therefore, that our developer network is an *estimate* of developer collaboration.

One must note that network metrics take into account the structure of the network as opposed to making absolute measurements regarding developers. For example, when incorporating developer information into a failure prediction model, one may look to measurements of a developer's experience or merits. For example, a developer may work with twenty other developers in one project and be considered a hub, but may work in another project with twenty other developers and not be considered a hub. One developer's metrics may change based on distant collaborative changes in the network. Network metrics take into account the structure of the group, not the individual in isolation.

While developer networks are interesting abstractions on code churn information, we must also emphasize that a large number of qualitative generalizations of developer networks can be made. For example, consider the interpretation of hub developers. The mere presence of hub developers in a network could imply a workload imbalance, eliciting a possible reassignment of tasks. Alternatively, one could view hub developers as more crucial to the project, considering them "domain experts". Both conclusions are sensible, but offer differing interpretations. As a result, one must rely on empirical analysis and careful interpretation to determine the meaning of these metrics for effective process improvement. Though the goal of this paper is to show that developer networks can be used for failure prediction, the evidence that network metrics are viable for failure prediction indicates the need for further investigation of developer networks as a useful abstraction for process improvement. A brief analysis of the developer networks from our case study can be found in Section 5.

3.2 Illustrating the Developer Network Metrics

All of the network metrics described in the prior section will be referred to as developer-based network metrics, since the metrics are calculated for each developer (as opposed to a development artifact). To apply these metrics to failure prediction, however, we also need "file-based" metrics, or metrics calculated on a per-file basis. Each file-based metric should reflect the network metrics of developers who updated the file throughout the file's history. To calculate a file's network metrics, we examine a file's update history in the source code repository, list all of the distinct developers who updated the file, and calculated the sum, maximum, and average of each developer metric over the file's history. For example, Max of Betweenness on file F is the maximum of all developer Betweenness values for the developers who updated F. Values are calculated per-developer, not per-update, so if a developer updated a file twice, his or her metrics would only be used once.

To better understand how developer- and file-based network metrics are calculated, consider the following example. Suppose we are initially given the churn information in Table 1. In our example, we have developers Alison, Bob, Chad, and Deb. We have three files, X, Y, and Z. Our system has two releases, P and Q. Note that files need not be executable, and releases need not be consecutive. For instance, the top line of churn information table can be read as "Alison updated file X during release P".

Table 1: Churn information for the network metrics derivation example

Developer	File	Release
Alison	File X	P
Bob	File X	P
Bob	File Y	P
Chad	File Y	P
Alison	File Y	Q
Bob	File Z	Q
Chad	File Z	Q
Deb	File Z	Q

First, we build a developer network from churn information (see Figure 1). Our definition of a connection between two developers is that both worked on at least one file in common during the same release. For instance, Alison and Bob worked on file X during release P, so they are connected. Note that Alison and Chad are *not* connected, even though they both worked on file Y, as the updates were during different releases. The resulting developer network is shown in Figure 1.

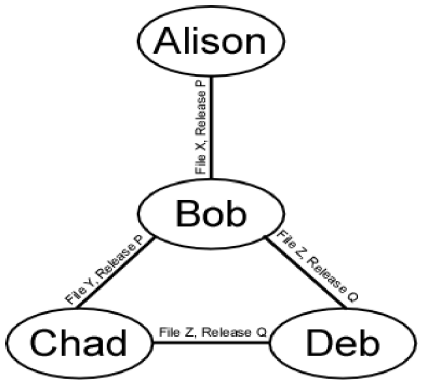

Figure 1: Resulting developer network from the example

The second step is to calculate degree and centrality metrics for each developer. Table 2 details the measures of our example. The degree of Chad is exactly 2, for instance. Closeness for Chad is (1+1+2)/3=4/3 because the shortest distance to Bob and Deb are each 1, and the shortest distance to Alison is 2.

To calculate Betweenness, we must first list out all of the shortest paths. Since this particular network is connected, we know there must be 6 geodesic paths: Alison-Bob, Alison-Bob-Chad, Alison-Bob-Deb, Bob-Chad, Bob-Deb, and Chad-Deb. Betweenness is calculated by counting how many shortest paths a particular developer is included in. Bob, for example, is on 5 out of the 6 shortest paths in the network, giving Bob a Betweenness of 5/6. The other three developers are only on the shortest paths beginning with themselves, so they all have a Betweenness of 3/6.

One may observe that a developer who has made many updates over time (which has been used to estimate a developer's "experience" [16, 19]) is not necessarily related to his or her centrality. Alison, for example, made two changes to two distinct files over two releases, yet was not considered as central as Deb, who only made one update to one file in one release.

Table 2: Developer-based metrics calculated from Figure 1

Developer	Degree	Closeness	Betweenness
Alison	1	5/3 ≈ 1.67	3/6
Bob	3	1	5/6
Chad	2	4/3 ≈ 1.33	3/6
Deb	2	4/3 ≈ 1.33	3/6

Once the developer-based metrics have been calculated, the last step is to calculate network metrics on a per-file basis. Using the churn information from Table 1, we determine the sum, max, and average of the developer metrics for each developer who updated the file. File-based network metrics are detailed in Table 3, showing only the summation for each metric. For example, file Y was updated by Bob, Chad, and Alison over its entire history, making its "Sum of Degree" metric 3+2+1=6 (each developer's degree can be found in Table 2).

Table 3: File-based network metrics from churn information in Table 1 and developer-based metrics in Table 2

File	Sum of Degree	Sum of Closeness	Sum of Betweenness
File X	4	8/3 ≈ 2.67	8/6 ≈ 1.33
File Y	6	4	11/6 ≈ 1.83
File Z	7	11/3 ≈ 3.67	11/6 ≈ 1.83

The resulting file-based network metrics are the candidate metrics we used to correlate with failures in the system.

3.3 Independent and Dependent Variables

As described in Section 3.1, we calculated over each file's history the sum, average, and maximum of each developer-based network metric as candidate independent variables for our model. Other relevant metrics, such as number of updates and code churn, are also added to our list of candidate metrics. These metrics are added to our candidate list as a control for simplicity: if a simpler metric (e.g. code churn) can be an adequate model, then there is no reason to use a more complex metric (e.g. Max of Betweenness). Furthermore, if our final model includes metrics from both categories without being over-fit, we can conclude that the network analysis metrics provide additional modeling power that code churn metrics could not provide.

Transformations of all metrics such as log, square root, and inverse were considered for each metric to avoid skew. Since the candidate regressions are generalized linear regressions, a transformation of a metric can result in linear data. Plotting a single metric at a time can provide insight into the most appropriate transformations.

The candidate metrics with descriptions are listed in Table 4.

Table 4: Candidate file-based metrics

Metric	Description
Code Churn	The number of lines of code that were either added or changed over the history of this file
Updates	The number of updates to the repository that included this file
Developers	The number of distinct developers who have updated this file over its history
(Sum/Average/Max) of Degree	The (sum/average/maximum) of each developer's degree over a file's history
(Sum/Average/Max) of Closeness	The (sum/average/maximum) of each developer's Closeness over a file's history
(Sum/Average/Max) of Betweenness	The (sum/average/maximum) of each developer's Betweenness over a file's history
Number of Hub Developers	The number of distinct hub developers who update this file

We had two dependent variables in our study: the number of system test failures for a file, and the number of post-release failures for a file. The number of system test and post-release failures per file was calculated from code churn information joined with trouble reports. One update to a file is associated with at most one trouble report, and a trouble report can involve multiple files. To determine the number of failures a given file had, we defined a failure as a trouble report that resulted in a fix. The number of failures for a given file, therefore, is equal to the number of trouble reports that involved that file. Trouble reports that did not result in a fix were not considered since they were never traced to specific files. Each trouble report was labeled as either from testing or post-release.

3.4 Model Selection and Validation

To discover a correlation between candidate metrics and system failures, we need to select a predictive model. Model selection is the process of finding the best combination of variables and a regression which can explain the variance in our data (i.e. model our data). The model selection process for regression requires two types of data sets: a training set and a validation set. The training set is used in the training stage to determine the weights of the variables in the model and to calculate goodness-of-fit statistics. Goodness-of-fit statistics are measures of how well the model fits the training set. The validation set is held out of the analysis until the final model has been selected.

We define two training sets and therefore develop two models: a system test model and a post-release model. With the training set in place, we choose regressions for model selection. Our three candidate regressions are all generalized linear regressions previously used [12] for predicting failure count data: negative binomial regression, Poisson regression, and the logistic regression. The negative binomial and Poisson regressions estimate the number of failures for a given file, by which we rank for our prioritization. The logistic regression predicts the probability that a file had at least one failure, and our ranking was based on that estimated probability.

The process of statistic regression analysis of network metrics can be enumerated in four steps: initial model selection, final model

selection, model validation, and further analysis. For the latter three steps, we evaluate our model with two evaluation criteria: Spearman rank correlation coefficient, and a comparison of the predicted prioritization versus the optimal. Since we are providing a *file-based prioritization* (ordering) to guide V&V activities, we must use statistics to evaluate prioritizations. The Spearman rank correlation coefficient is used to estimate the correlation between the rank of our predicted values and the rank of our observed values. One important note is that the square of the correlation coefficient is equal to the percentage of variance explained by the model (e.g. a prediction with a correlation coefficient of 0.6 explains 36% of the variance in the data).

In addition to the Spearman rank coefficient, we also examine how our prioritization fared in comparison with an optimal prioritization. The optimal prioritization is found by sorting all of the files by their observed failure counts. Examining the optimal prioritization is important in this kind of analysis because one must make a comparison with how good the ordering could have been, which can vary greatly from product to product.

Each of the four steps is now discussed:

Step One: Initial model selection. Model selection is done by systematically forming models of the training set with combinations of the candidate variables, including transformations, and the candidate regressions. Combinations in which the variables are known to be strongly associated with each other are not considered (e.g. different transformations of the same metric). The fitting of each model produces the beta-weight (weighted contribution) of each variable to the model. Goodness-of-fit statistics of each model are evaluated as the training error (i.e. lack of fit), and models with the poorest training error are discarded. Model fitting and goodness-of-fit statistics were calculated in SAS v9.1 using `proc genmod`. Training error measures included maximum likelihood significance tests on the each of the partial regression coefficients evaluated at the $p<0.05$ level, and the overall log-likelihood of the model.

Possible reasons for poor training error include too few or non-explanatory variables, also known as the model being under-fit as it does not explain enough of the variance in the training set. Models can also result in poor training error because of multicollinearity; that is, having variables that are strongly associated with each other. The output of Step One is an initial set of candidate models that have low training error.

Step Two: Final model selection. Models with low training error are considered for the final model. A low training error does not always imply an accurate model: a model could "memorize" the training examples and not be good for prediction, known as an over-fit model. In the second phase of model selection, models with low training error are cross-validated to evaluate their predictive power. Cross-validation, also called rotation estimation or hold-out validation [7], is an estimation technique used to provide accurate prediction values based on the training set. In particular, cross-validation is good for catching over-fit models. In cross-validation, the original training set is randomly partitioned into a training partition and a validation partition. For regression, training means the beta-weights of the model are calculated based on the training partition. Predictions are made on the validation partition[2] using the model developed from the training partition. In ten-fold cross-validation, the data set is randomly split into ten portions. Training and validation is done ten times, with each portion being the validation set exactly once and the other nine partitions compose the training set. Since the set is split into ten samples, the union of all samples is the original set.

We calculate the Spearman rank correlation coefficient between the predicted and observed values for each of the ten partitions separately. The output of this step is the average and standard deviation of the ten correlation coefficients for each of the models. The two models with the highest average correlation coefficient and the lowest standard deviation become our final models to be validated.

Step Three: Model validation. When our best system test model and post-release model are each selected from Step Two, they are evaluated against the validation set (which has been left out of the whole process until the final validation). As discussed before, our two evaluation criteria are (a) Spearman rank correlation coefficient between the estimated values and the observed values, and (b) examining the difference between our predicted prioritization and an optimal prioritization.

Step Four: Further Analysis. Once a model has shown to be adequately predictive, the last step is to evaluate how it well it might work in practice. First, we compare the model to a "classic" source-lines-of-code (SLOC) model. We choose the SLOC model as a baseline of comparison as it has been used as a failure prediction metric in the past [25]. Second, to determine if network metrics provide extra predictive power, we compare the model with a model containing only code churn metrics and not network metrics, and vice versa. Third, to assess network metrics as an *early* indicator, we evaluate the model as if it were halfway through the development phase. Fourth, we investigate possible latent factors influencing the model. We investigated a possible latent factor involving the imbalance of developer experience by attempting to incorporate a known metric for developer experience/effort into our model. Lastly, we analyze our developer network by itself for possible interpretations for process improvement.

3.5 Threats to Validity

The goal of our model is to show that network metrics can adequately prioritize files based on estimated failures. A statistical issue with creating failure prediction models is the underlying problem of latent factors. Since correlation does not imply causation, there may be latent factors that influence both network metrics and cause system failures. One possibility for latent factors may be the design of the actual system. For example, if a central, hub developer works on files that many other developers work on, perhaps he or she works on an integrated layer of the system, whereas a non-hub developer may work relatively independently because he or she is working on, say, device drivers. This possible factor is certainly worth an investigation; however, our data had little information on the design of the system.

Another limitation of our approach is that every file must have churn history. Without a churn history, a file has no list of updating developers and no network metrics can be calculated. One way to mitigate this problem is to rely solely on code churn metrics and to count the new files as "fully churned." This limitation highlights the need to integrate network metrics into full models which incorporate many metrics, not just from code churn information.

[2] Not to be confused with the validation set mentioned in the next step, validation partitions are still part of the training set.

Finally, this study was conducted on a single project with a single data set. The developer network formed from this project's code churn data may be specific to the process and developers involved in the project. Further case studies are needed to determine if these results can be generalized.

4. NORTEL CASE STUDY

Sections 4.1 through 4.3 describe our data collection, models and validation. Section 4.4 addresses factors related to deployment of our model.

4.1 Study Context and Data Collection

We built and validated our prediction model with data from an industrial product at Nortel Networks, a telecommunications company. Telecommunications systems must have high reliability because failures can cause major disruptions in the daily life and workings of society. As a result, Nortel faces intense pressure for their verification and validation efforts to be as effective and efficient as possible.

Data was collected from three annual releases of a large, mature[3] networking product consisting of over 11,000 files and 3.17 million lines of code. About 2,500 files were churned during our training releases, meaning that only 2,500 of the 11,000 files had network metrics associated with them. As discussed in Section 3.5, only the 2,500 files could be examined in this study. Fortunately, most of the failures occurred in files that had been churned. System failure data and code churn information for the first two releases were used as a training set and the third release was held out as a validation[4] set. For the rest of this paper, we will refer to the training set releases as R_N and R_{N+1} and the validation set as R_{N+2}.

Our data set included churn information, system test failure data, and post-release failure data by file. The churn information is a table taken from the configuration management records which contains a row for each update made to the code, the file that was updated, the date of the update, the developer who made the update, lines of code added/changed/deleted, and an optional trouble report code for the update. One update to a file is associated with at most one trouble report, and a trouble report can involve multiple files. To determine the number of failures a given file had, we defined a failure as a trouble report that resulted in a fix. The number of failures for a given file, therefore, is equal to the number of trouble reports that involved that file. Trouble reports that did not result in a fix were not considered since they were never traced to specific files. Each trouble report was labeled as either from testing or post-release. Only updates to source code were included in our study, not documentation or other non-executable files.

4.2 Steps One and Two: Model Selection

The resulting models from Steps One and Two in Section 3.4 and their performance in cross-validation are as follows. Spearman correlation coefficients were calculated by the SAS v9.1 `proc corr` routine, which averages ranks in the case of a tie.

[3] The actual release number, R_N, has been removed to protect proprietary information.

[4] Technically, we perform a pseudo-validation set because validation requires random sampling.

4.2.1 System Testing Model

The resulting regression from the model selection process for the system testing failure model was a negative binomial regression. The five variables consisting of the metrics and their transformations are located in Table 5. The actual beta-weights are not included to protect proprietary information. Degree was positively correlated with failures and Closeness was negatively correlated, indicating that the files updated by central developers were less failure-prone.

By cross-validating the system testing model, the average of the ten Spearman rank correlation coefficients for the system test model was 0.778 with a standard deviation of 0.03. Squaring the coefficient means that 60.5% of the variance in the system test data was explained by our model. The strong correlation between predicted and observed values for the system test model indicates that the model is good for prediction.

Table 5: Variables of the test failure model

Metric	Transformation
Code Churn	Log
Updates	None
Developers	Square root
Sum of Degree	Square root
Sum of Closeness	None

Figure 2 shows the cumulative number of test failures found if files were prioritized using the union of the predicted values from cross-validation compared with the optimal ordering. For the first 20% of the files, our model was very close to optimal: we found 82% of the failures in 20% of the files, where 84% was optimal. The random ranking series in this figure and following represents the theoretical, unweighted average of all possible rankings.

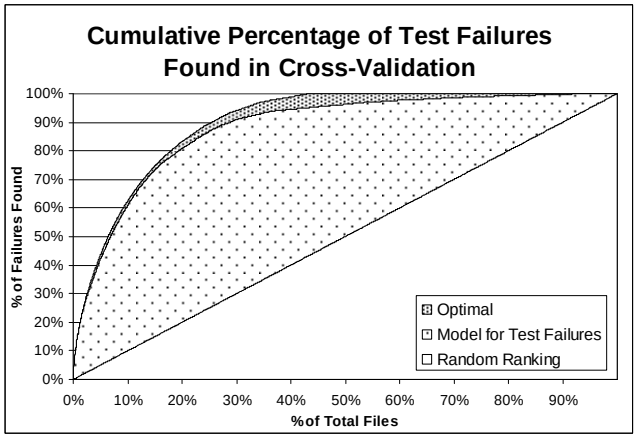

Figure 2: Cumulative percentage of test failures found based on the prioritizations of the cross-validation predictions

4.2.2 Post-Release Model

The resulting regression from the model selection process for the post-release model was a logistic regression based on the estimated probability of a file having any failures. The four variables consisting of the metrics and their transformations are

located in Table 6. The actual beta-weights are not included to protect proprietary information. As in the system testing failure model, however, Degree was positively correlated with failures and Closeness was negatively correlated, indicating again that the files updated by central developers were less failure-prone.

Table 6: Variables to metrics in the post-release failure model

Metric	Transformation
Updates	None
Developers	Square root
Sum of Degree	Square root
Sum of Closeness	None

By cross-validating the post-release model, the average of the ten correlation coefficients for the post-release model was 0.163 with a standard deviation of 0.07. Squaring the coefficient means that only 2.6% of the variance in the data was explained by our model. The significance test for each of the correlation coefficients was significant at $p<0.001$, thus the effect is weak, but significant.

We suspect the weak correlation is due to the post-release data not being independently and identically distributed, an underlying assumption of our regressions. A lack of independence may be due to separate customers who use the system under different operational profiles. Another possibility for the weak correlation is over-fitting. We believe over-fitting to be not as likely since post-release failure models with fewer variables did not yield better results.

Figure 3 shows the cumulative number of post-release failures found if files were prioritized using the union of the predicted values from cross-validation compared with the optimal ordering and a random ordering.

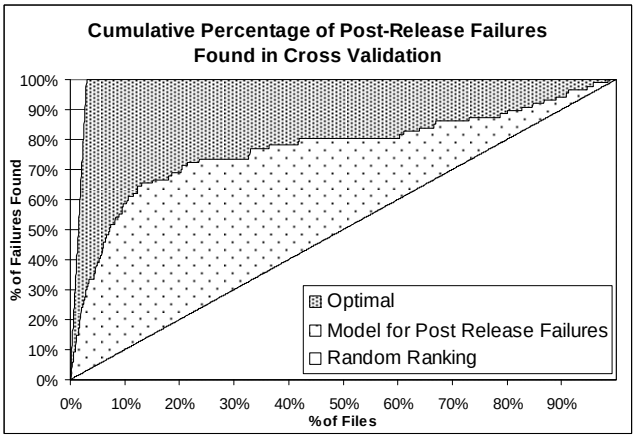

Figure 3: Cumulative percentage of post-release failures found based on the prioritizations of the cross-validation predictions and the optimal prioritization

The post-release model found 81% of the failures in 20% of the files, where optimal is 100%. The cross-validation results from the post-release failure model were not quite as close to optimal as the system testing model, however, the prioritization was still greater than a random prioritization, implying that our model is still better than no model at all.

4.3 Step Three: Model Validation

In this section, we discuss the empirical validation of our models in the context of our case study. We used next-release validation as our final validation.

When system test failure data for the next release of Nortel's large networking product became available, we were able validate our system test model trained from R_N and R_{N+1} against the latest release, R_{N+2}. Since the product had been deployed for only a few months, the data for post-release failures was not available for analysis. Therefore, we considered cross-validation as our final evaluator of our post-release model. Table 7 gives metadata on the dataset from release R_{N+2}.

Table 7: Metadata on release R_{N+2} validation dataset

Number of files deleted between R_{N+1} and R_{N+2}	835
Number of pre-existing files churned in R_{N+2}	2,035

We compared our predictions taken from building a model with R_N and R_{N+1} data to the observed failure counts of R_{N+2}. The failure counts for files that were new in R_{N+2} were not handled; the model cannot make a prediction for files that did not exist at the time of the prediction. The Spearman rank correlation coefficient for our test model in next-release validation was 0.741 ($p<0.01$). Figure 4 shows the cumulative percentage of failures found based on our prioritization. The "Model for Test Failures" area is our prioritization, and the darker region is the optimal prioritization. Our model found revealed 58% of the failures in 20% of the files compared with the optimal prioritization that would have found 61% in 20% of the files. By comparing the optimal region in Figure 4 with the optimal region in Figure 2 (R_{N+2}'s optimal region is more flat), one can see that the failures were distributed over more files in R_{N+2}. Nevertheless, our model was still considerably close to optimal.

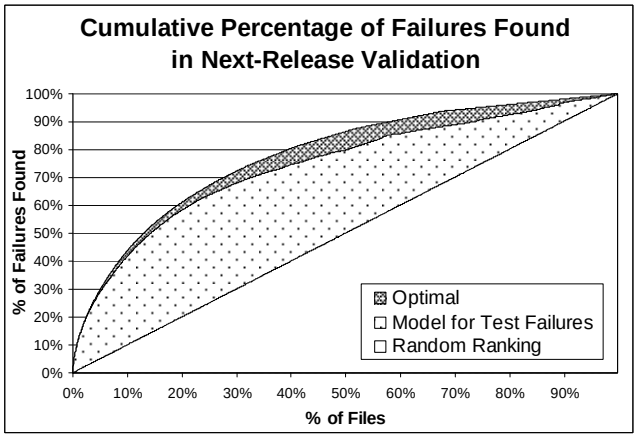

Figure 4: Cumulative percentage of test failures found on pre-existing files in R_{N+2}.

To illustrate the value of using our model, the rate of actual discovery of failures by the Nortel system test team is charted in Figure 5, that is, the cumulative percentage of failures found per week of testing. The added "Baseline" series denotes the cumulative percentage of failures found if the testing department had found the exact same number of failures each week (equivalent of a random ranking). The "Optimal" series is if the

testing department had their highest failure-finding weeks first. Of particular note is that the time period in which the testing department struggled to find failures was in the first eight weeks of testing (or, in the first 18% of the total testing time). One explanation for this behavior is that, with the new release of the system, testers initially do not know where to start testing. The ordering in Figure 4, however, shows that the strongest period of time, the time when the model is near or at the optimal prioritization, is in the first 30% of files tested. Our model performs best at finding many failures early, where the testers need guidance the most.

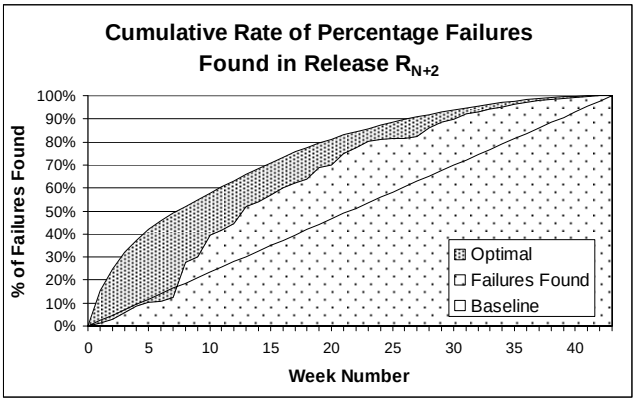

Figure 5: Cumulative percentage of observed failures found over time during R_{N+2}

4.4 Step Four: Further Analyses

Further analysis into the practicality of network metrics in failure prediction models follow.

4.4.1 Comparison with Similar Models

The previous validations show that our metrics performed well for failure prediction when used in the system test model that we chose. Our model includes metrics directly from code churn information in combination with the metrics from SNA (see Section 4.2.1). To show what kind of contribution our network metrics are making to the model, we performed our ten-fold cross validation analysis on three additional models: a source-lines-of-code (SLOC) model, a code churn only model, and a network metrics only model.

We performed the same model selection process as described in Section 3.4 with the SLOC metric as our only candidate variable, considering transformations as well. Our final SLOC model included a log-transformation of the SLOC metric in a negative binomial regression.

The "Code Churn Only" model includes the same variables from our final testing model (see Section 4.2.1), only without the network metrics: (i.e. Code Churn, Number of Updates, Number of Developers). Similarly, the "Network Metrics Only" model includes only the network metrics (i.e. Sum of Closeness, Sum of Degree). Negative binomial regression was used for both models as it was the same regression used in the final testing model.

The average and standard deviations of the Spearman rank correlation coefficients are detailed in Table 8; correlation coefficients were significant ($p<0.01$). The high standard deviation for the Code Churn Only model indicates that some partitions did not perform as well in cross-validation. Based on Fisher's z-transformation the correlation coefficient for the System Test Model, was significantly higher than the Code Churn Only model in each of the five folds ($p<0.05$).

Table 8: Spearman correlation coefficients from cross-validation of the training set

	Average	Standard Deviation
SLOC Model	0.400	0.03
Code Churn Only	0.706	0.10
Network Metrics Only	0.514	0.05
System Test Model	0.778	0.03

Our results indicate that all four models are viable for producing a file prioritization based on predicted failures; however, our full model including network metrics performed the best. That the model performed better when adding the network metrics indicates that our metrics are explaining additional variance that code churn alone could not explain.

4.4.2 Network Metrics as an Early Indicator

Our case study's training set is from two full phases of development, so when our results indicate that our model is an accurate indicator of failures, our model would best be applied at the end of the development phase when all code churn information is available. In our collaboration with Nortel for this case study, having our model's predictions available between the development phase and the testing phase was satisfactory for their purposes. However, many testers design their test plans much earlier than the start of system test, so having our model predict failures earlier is more desirable.

Since churn data is available during the development phase, one could use our model at any time in the development phase. We ultimately envision our model being used as a form of process improvement, where developers would adjust their process based on our model's analysis of the current churn data. To empirically show that our model can be used early in the development phase, we performed our analysis of ten-fold cross-validation (described in Section 3.4) using data from only the first half of the development time during release R_{n+1}. The churn records were less than half the number of the final churn records and the developer network had about a quarter of its edges. Using only the data from the first half of development, our model performed an average Spearman rank of 0.693 with standard deviation of 0.02, with all correlation coefficients significant ($p<0.01$). Comparing these results with the result from our previous analysis (Spearman $r=0.778$), we conclude that having all of the data available is ideal; however, our model had a significant ranking correlation performed using only half of the data, so our model could provide valuable information early in the development phase and would be a good indicator for testers to use as they design their tests.

4.4.3 Developer Effort and Experience

Since correlation does not imply causation, one must always consider possible latent factors which could influence both the network metrics and cause system failures. For example, we mentioned the design of the system being a possible latent factor in Section 3.

Another of the possible latent factors influencing the accuracy of our model is the notion of a developer's experience and effort in the project. One might think that if a developer simply worked on the project more, he or she would generally be more central to the

network. We sought to quantify a developer's experience and determine if it could be a latent factor in our model.

Historically, researchers have quantified experience and/or effort in their models and have found their metrics to be significant [16, 19]. One of the more common ways to quantify effort [16] is to add up the number of lines of code that a developer has entered into the system for all previous updates. We will call this developer-based metric "Developer Churn." Each file's metric is the sum of all Developer Churn over each developer. We will call the file-based metric of Developer Churn "Effort." Effort will be highest for files updated by many developers who have made many updates in the past.

When incorporating the Effort metric into our model, we found that our model was over-fit, that is, too complex due to being highly associated with the other variables. Furthermore, we found that using Effort alone resulted in a Spearman rank correlation of 0.26 ($p<0.01$) on the training set, which is a weak correlation. Hence, we reject the Effort model for our Network Metrics.

There are other ways to measure developer experience and/or effort using churn data, and there is the possibility that such a metric would not result in an over-fit model. Although one may be able to construct a different metric regarding experience, we find that metrics based on SNA provide a stronger foundation for process improvement, as they lend themselves to more nuanced interpretations.

5. Discussion of the Network

Modeling collaboration with easily obtainable code churn information also provides opportunity for process improvement during the development phase. Before we can use this network for process improvement, however, we need to investigate the meaning behind its structure. We applied a few methods of Social Network Analysis to perform such an investigation.

In our case study, we analyzed the developer network of the training set in search for empirically-sound interpretations of the network. A summary of the network's metrics can be found in Table 9.

Table 9: Overall developer network metrics from the case study training set

Number of developers	161
Average degree (hub threshold)	19.78 (30)
Number of hubs	37 (23%)
Number of disconnected	11 (6.8%)
Network diameter	9
Average Closeness	2.77
Average Betweenness	0.93

Our developer network had a high rate of hub developers. Twenty-three percent (23%) of the developers are considered to be "hub" developers, that is, 23% of the developers worked with more than thirty other developers over the course of two releases. Considering that the hub threshold is calculated based on the Poisson distribution of a random network with a p-value of 1%, we can safely claim that this network's degree distribution does not follow a Poisson, and is not a random network. In fact, the degree distribution follows a power law. The standard test [1] we performed was a linear regression on the log-log scale of the degree distribution, and we obtained an r-squared of 0.81 (that is, 81% of the variance was explained by the linear regression).

Networks with a degree distribution of a power law are also known as "Scale-free" or "Small world" networks. The term "scale-free" comes from the notion that, as more nodes are added to the network, the diameter of the network does not change because as the number of nodes increases, peripheral nodes become hubs [1]. Scale-free networks are found throughout Social Network Analysis studies, particularly in collaboration networks. Examples of scale-free collaboration networks include email networks in open source software projects, movie actors, and authors of scientific publications [5, 9, 13].

In every scale-free network, the hub plays a vital role. While hubs provide a small diameter, removing hubs can result in a disconnected network. This lack of robustness could be a valuable indicator during development, perhaps for re-assignment of tasks or for code inspections. Network Analysis provides several metrics that quantify robustness [1]. A lack of robustness would indicate that the network would become disconnected should a developer leave. Furthermore, the precise relationship between hubs and the collective knowledge of the project is unclear. Based on this study, hubs appear to resemble the "gatekeeper" presented by Allen [2]. We were not able to interview the hubs in our case study to gauge their overall knowledge of the network.

Based on this study, however, we cannot provide a definitive set of process improvement guidelines as the network may not be complete. In this study, the *only* form of collaboration captured by the network is from code churn data, so some communication may be missed. To provide a basis for process improvement, one needs to analyze the network in both the context of failure prediction and in social network analysis.

6. SUMMARY

We developed and validated a failure prediction model based on SNA of developers in a large software system. Using the validation method of next-release validation, we found that our model performed significantly well in prioritizing files based on predicted failures. We have shown that developer networks are useful for failure prediction early in the development phase and provide a useful abstraction of the code churn data. Furthermore, the correlation between network metrics and failures introduces many possibilities for the use of SNA in software reliability. The ability to create metrics based on the structure of a group of developers could prove to be a powerful addition to current failure prediction models as the developer network can also provide a basis for process and organizational improvement. Further investigation into how developer networks vary among projects, processes, and domains would provide more insight into these intriguing metrics.

7. FUTURE WORK

Developer networks provide a promising foundation for several novel metrics to be introduced into failure prediction models. In future work, we hope to explore a more sophisticated analysis of a project's developer network and how developer information can be applied to products at the file level for failure prediction to improve our model. Examining connection weighting schemes, network robustness, clustering, and the evolution of developer networks over time are among the many possible areas to explore.

The applications of developer networks go beyond failure prediction; developer networks have implications to many areas of

software engineering. An investigation into how closely associated a developer network is to true collaboration is warranted. Comparisons of developer networks from different projects, processes, and domains should be made. Once we have a firm understanding of the developer network, we can begin to make proactive steps toward organizational improvement rather than reacting to the current state for V&V guidance.

8. ACKNOWLEDGMENTS

This research is supported by a research grant from Nortel Networks. We would like to thank the members of the Software Engineering Realsearch group at North Carolina State University along with Thomas Zimmerman for his feedback.

9. REFERENCES

[1] Network Analysis: Methodological Foundations. Berlin: Springer, 2005.

[2] Allen, T. J., Managing the Flow of Technology: MIT Press, 1977.

[3] Arisholm, E. and Briand, L. C., "Predicting Fault-prone Components in a Java Legacy System," in 2006 ACM/IEEE International Symposium on Empirical Software Engineering, 2006, pp. 8-17.

[4] Arisholm, E., Briand, L. C., and Fuglerud, M., "Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software," in 18th IEEE International Symposium on Software Reliability Engineering, 2007.

[5] Barabasi, A.-L. and Albert, R., "Emergence of scaling in random networks," Science, vol. 286, no.5439, pp. 509-512, 1999.

[6] Barabasi, A.-L. and Oltvai, Z. N., "Network Biology: Understanding the Cell's Functional Organization," Nature Reviews Genetics, vol. 5, no.2, pp. 101-113, 2004.

[7] Bengio, Y. and Grandvalet, Y., "No Unbiased Estimator of the Variance of K-Fold Cross-Validation," J. Mach. Learn. Res., vol. 5,pp. 1089-1105, 2004.

[8] Bernstein, A., Ekanayake, J., and Pinzger, M., "Improving Defect Prediction using Temporal Features and Non Linear Models," in Ninth International Workshop on Principles of Software Evolution: in conjunction with the 6th ESEC/FSE joint meeting, 2007, pp. 11-18.

[9] Bird, C., Gourley, A., Devanbu, P., Gertz, M., and Swaminathan, A., "Mining email social networks in Postgres," in 2006 international workshop on Mining software repositories, 2006, pp. 185-186.

[10] Boehm, B. W., Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[11] Denaro, G. and Pezz, M., "An Empirical Evaluation of Fault-Proneness Models," in 24th International Conference on Software Engineering, 2002, pp. 241-251.

[12] Gao, K. and Khoshgoftaar, T. M., "A Comprehensive Empirical Study of Count Models for Software Fault Prediction," Reliability, IEEE Transactions on, vol. 56, no.2, pp. 223-236, June, 2007.

[13] Girvan, M. and Newman, M. E. J., "Community Structure in Social and Biological Networks," The Proceedings of the National Academy of Sciences, vol. 99, no.12, pp. 7821-7826, 2001.

[14] Gonzales-Barahona, J. M., Lopez-Fernandez, L., and Robles, G., "Applying Social Network Analysis to the Information in CVS Repositories," in 2005 International Workshop on Mining Software Repositories, 2004.

[15] Huang, S.-K. and Liu, K.-m., "Mining Version Histories to Verify the Learning Process of Legitimate Peripheral Participants," in 2005 International Workshop on Mining Software Repositories, 2005, pp. 1-5.

[16] Hudepohl, J. P., Aud, S. J., Khoshgoftaar, T. M., Allen, E. B., and Mayrand, J., "Emerald: Software Metrics and Models on the Desktop," Software, IEEE, vol. 13, no.5, pp. 56-60, 1996.

[17] Lave, J. and Wenger, E., Situated Learning: Legitimate Peripheral Participation. Cambridge: Cambridge University Press, 1991.

[18] Mockus, A. and Weiss, D. M., "Predicting Risk of Software Changes," Bell Labs Technical Journal, vol. 5,pp. 169-180, 2002.

[19] Mockus, A., Weiss, D. M., and Zhang, P., "Understanding and Predicting Effort in Software Projects," in 25th International Conference on Software Engineering, 2003, pp. 274-284.

[20] Nagappan, N. and Ball, T., "Static Analysis Tools as Early Indicators of Pre-Release Defect Density," in 27th International Conference on Software Engineering, 2005, pp. 580-586.

[21] Nagappan, N. and Ball, T., "Use of Relative Code Churn Measures to Predict System Defect Density," in 27th International Conference on Software Engineering, 2005.

[22] Nagappan, N., Ball, T., and Zeller, A., "Mining Metrics to Predict Component Failures," in Proceeding of the 28th International Conference on Software Engineering, 2006, pp. 452-461.

[23] Ohira, M., Ohsugi, N., Ohoka, T., and Matsumoto, K.-i., "Accelerating Cross-project Knowledge Collaboration using Collaborative Filtering and Social Networks," in 2005 International Workshop on Mining Software Repositories, 2005, pp. 1-5.

[24] Ostrand, T. J., Weyuker, E. J., and Bell, R. M., "Locating Where Faults Will Be," in 2005 conference on Diversity in computing, 2005, pp. 48-50.

[25] Weyuker, E. J., Ostrand, T. J., and Bell, R. M., "Using Developer Information as a Factor for Fault Prediction," in Third International Workshop on Predictor Models in Software Engineering, 2007, pp. 8-8.

[26] Yu, L. and Ramaswamy, S., "Mining CVS Repositories to Understand Open-Source Project Developer Roles," in Fourth International Workshop on Mining Software Repositories, 2007, p. 4.

[27] Zimmermann, T. and Nagappan, N., "Predicting Defects using Network Analysis on Dependency Graphs," in 29th International Conference on Software Engineering, 2007.

[28] Zimmermann, T., Premraj, R., and Zeller, A., "Predicting Defects for Eclipse," in Third International Workshop on Predictor Models in Software Engineering, 2007, p. 9.