Does Calling Structure Information Improve the Accuracy of Fault Prediction?

Yonghee Shin[1], Robert Bell[2], Thomas Ostrand[2], Elaine Weyuker[2]

[1]*Department of Computer Science*
North Carolina State University
Raleigh, NC, 27606
yonghee.shin@ncsu.edu

[2]*AT&T Labs - Research*
180 Park Avenue
Florham Park, NJ 07932
(rbell, ostrand, weyuker)@research.att.com

Abstract

Previous studies have shown that software code attributes, such as lines of source code, and history information, such as the number of code changes and the number of faults in prior releases of software, are useful for predicting where faults will occur. In this study of an industrial software system, we investigate the effectiveness of adding information about calling structure to fault prediction models. The addition of calling structure information to a model based solely on non-calling structure code attributes provided noticeable improvement in prediction accuracy, but only marginally improved the best model based on history and non-calling structure code attributes. The best model based on history and non-calling structure code attributes outperformed the best model based on calling and non-calling structure code attributes.

1. Introduction

Billions of lines of software are in use, and billions of dollars are spent each year creating and fixing them. A sizeable percentage of this cost is spent identifying and removing faults [1]. Late fixes of faults cost more than when they are detected and fixed early in the development life cycle [6]. Predicting where faults are likely to reside helps organizations allocate more testing resources to the most fault-prone parts of the software early, thereby reducing the cost of fault identification and leading to the efficient creation of high-quality software. This is particularly useful since faults often display a Pareto-like distribution with the majority of faults occurring in a small percentage of files[8, 18, 19].

Prior fault prediction studies have used software code and history information that can be collected automatically. History attributes include counts of faults and changes in previous releases, and information about developers who have interacted with the code. Some code attributes, such as file size, file type, code complexity based on code structure, and code dependency information, can be obtained directly from the source code of the current release. Other attributes such as the age of a file or whether the file was new in the previous release require a version management system or an equivalent means of accessing code of previous releases.

While new attributes have been proposed and subsets of proposed attributes have been used to predict faults, the effectiveness of individual attributes in conjunction with other code and history information has not been investigated in depth. Attributes that were useful separately in fault prediction might not be useful when combined with other attributes in a model because they are closely correlated.

This paper extends our prior work on fault prediction, which used code and history information [20, 25]. In addition to the attributes we have used previously, we now investigate the use of calling structure relationships. This information might enhance fault prediction accuracy, because interface faults are very common [4], and it is believed that many software faults at the method level involve changes to the methods that either call or are called by the affected method. We investigate the effectiveness of calling structure attributes that can be obtained directly from source code, as well as attributes that combine calling structure and history information.

Our empirical study uses code, history, and calling structure information obtained from 35 releases of an industrial business maintenance system developed over a period of almost ten years. The goal of the prediction was to identify the top 20% of files that are predicted to contain the most faults. Although prediction at method

level would be more precise than at file level, we performed our analysis in terms of files because our fault information was available only at file level.

The rest of the paper is organized as follows: Section 2 describes related work. Section 3 defines calling structure and describes the calling structure attributes we used in detail. Section 4 describes the project under study and the data collection methods used. Section 5 explains our modeling and prediction methods. Section 6 provides prediction results. Section 7 mentions the limitations of our study and Section 8 contains conclusions and future work.

2. Related work

There have been many studies to predict the locations and the number or probability of faults using code, design, and history information [2, 3, 5, 7, 9, 13-18, 20, 22, 25, 26]. In this study, we investigate the additional contribution of calling structure attributes to prediction models whose variables include both code and history attributes.

Basili and Perricone [4] analyzed the distribution of faults from a large set of projects mostly written in Fortran. They classified the faults into five types including initialization, control structure, interface, data, and computation faults. Interface faults are associated with structures existing outside of a module such as method calls. Among the five categories, interface faults accounted for the largest portion (39%). This result is one motivation for our study of calling structure information, as it indicates that method calls can be a significant source of faults.

Using calling structure information is not new. Fan-in (the number of method calls to a given module from other modules) and fan-out (the number of method calls to other modules from a given module) [11] have been used for fault prediction [2, 18, 26]. Cohesion (the degree of interaction between elements within a module via method calls or access to data structures) and coupling (the degree of interaction between elements in other modules via method calls or access to data structures) [21] also have been studied previously in the context of fault prediction [2, 3, 7, 22, 26].

Arisholm and Briand [2] used code and OO design metrics, including the number of method calls, cohesion and coupling, to predict faults in a Java legacy system. They also used code quality metrics, code change information, and fault history. Using their prediction model, the estimated potential savings in verification effort was 29%.

Nagappan and Ball [15] used software dependencies and code change metrics to predict post-release failures on Windows Server 2003. They measured the dependency within a module and between different modules, and showed that dependency information was effective for failure prediction. However, they did not describe how much the dependency information improved results from a model using code change information alone. They predicted only post-release failures, while our goal is to predict both pre- and post-release faults, and, in fact, the vast majority of observed faults in the system we studied are pre-release faults.

Zimmerman and Nagappan [26] used data and call dependency information obtained from network analysis to see whether the role of a module within a network provides useful information for fault prediction. Their combined model of network metrics and code complexity had about 10% higher correlation with faults than their models using only one or the other set of variables.

Jiang et al. [12] compared the effectiveness of code-based metrics to design-based metrics of software for predicting fault-proneness, and observed that the code-based metrics generally outperformed the design-based ones. Their code metrics include many of the Halstead metrics as well as line and parameter counts. Their "design metrics" are mostly counts of graph properties of modules which they obtained from the code. Only one of them, the number of calls to other functions, is similar to the calling structure attributes that we use. Because their study analyzes the predictive ability of their combined sets of all code or design metrics, it is not possible to separate out the effect of only the calling structure for comparison with our results.

In [20], three of this paper's authors used lines of source code, file type, new file status, file age, code change history, and fault history to predict 20% of files containing an average of 83% of pre- and post-release faults in two large industrial systems. Extending the model to include developer information [25] improved the prediction accuracy to 84.9%. However, neither of these studies used calling structure information.

None of the above-cited studies contained an in depth assessment of the effectiveness of calling structure information separately. Therefore, to extend those studies, in this paper we consider new attributes that combine calling structure with either file status or history information. We then investigate the improvement in fault prediction accuracy by multivariate models that combine those attributes.

3. Calling structure attributes

Calling structure represents the relationship of invocation between methods of a program. In Figure 1, Methods X, Y, and Z are invoked or called by Method Q. The methods that are invoked by Q are Q's *callees*.

Methods A and B invoke or call Method Q. The methods that invoke Q are Q's *callers*. File F is a *caller* of file G, and G is a *callee* of F if F contains a method that calls some method(s) in G.

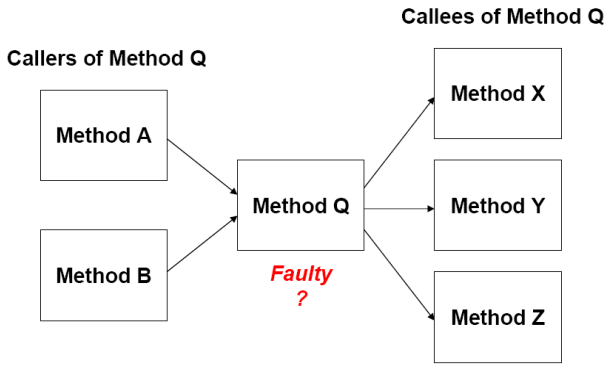

Figure 1. Calling Structure

We are interested in whether the calling structure information is useful in predicting faults for the following four reasons. First, a large percentage of faults are known to be interface faults, where interface faults are defined as faults associated with external module (method) or global data structures that are used by the current module [4]. Therefore, the more method invocations in a file, the more fault-prone the file might be. Second, new files have had less chance to be tested than older files. Therefore, interactions between new methods and existing methods might not have been sufficiently tested and therefore have a greater chance of having undiscovered faults.

Third, changes to a method frequently induce changes to other parts of the code. When a method X is changed (for example, to add or remove a parameter) method Q, which invokes X, may also require changes. Prior research showed that code change frequency is a good predictor of faults [9, 20].

Fourth, structured design with high cohesion and low coupling is generally accepted as a good approach leading to high system understandability, reusability, and maintainability [21]. When a module has low cohesion and high coupling to other modules, the module is often difficult to understand, reuse, and test. Changes to one module might be unnecessarily propagated to other modules in such cases. All of these problems can result in low quality software.

We quantified the following attributes of calling structure as model variables to investigate the influence of the above four aspects for predicting faults. We include a rationale for each attribute.

- *The number of callers and callees*

Since interfaces tend to have many faults, the more a method calls other methods, the more the method might have faults. We are also interested in determining whether a method is likely to be more faulty if it is called by many other methods.

- *The number of new callers and callees*

If a method interacts with callers or callees for the first time (either new code or existing code), the method itself or the file containing that method might have a high number of faults.

- *The ratio of the number of internal calls in a file to the total number of calls (both internal and external) associated with the file. (cohesion)*

If low cohesion and high coupling lead to more fault-prone design, then the higher the ratio of calls within a file to the total number of calls associated with the file, the fewer faults might be in the file.

- *The number of prior new callers and callees*

Just as was the case with new callers and callees, interaction with methods that were new in the prior release might not have had sufficient testing. Therefore, the more new callers or new callees in the prior release, the more a method might have faults.

- *The number of prior changed callers and callees*

Changes to a method's callers and callees often require that the method must also be changed, leading to higher fault-proneness for the method. Changes in the prior release N-1 are counted as the number of changes from the beginning of the development of Release N-1 until the current time of change history data collection.

- *The number of prior faulty callers and callees*

The rationale here is very similar to the previous attribute, but faulty callers or callees of Q might imply an even greater possibility of faults in Q. In addition, if the faults in a method are due to incorrect assumptions or misunderstanding of the requirements or specification, the developer might also have made incorrect assumptions or have misunderstandings of the methods that have a calling relationship with the faulty method.

The fault and change data we collected is at file level, and in general we do not know which particular method(s) in a file have been changed to correct a defect. Therefore, we measured all the above attributes at file level. For example, the number of prior faulty callers of file F counts the number of faulty files in the previous release that call F in the current release.

4. System information and data collection

4.1. System under study and fault information

The subject system of our empirical study is a business maintenance system that has had 35 releases and has been in the field for close to ten years. Although it is coded in many different programming

languages, we included in this study only C, C++, and C++ with embedded SQL files, as those were the file types for which we were able to obtain calling structure information. The numbers of files of these types ranged between 233 and 395 for the releases studied, with the number of files constantly increasing as the system matured. The total size of the studied code ranged from approximately 135 KLOC to 400 KLOC, again constantly increasing with time. In all releases, these files made up the vast majority of the total system, typically consisting of over 70% of the files, and over 90% of the LOC.

The development organization used a combined version control and change tracking system. A *modification request* (MR) is issued any time a change is necessary, and the completed MR specifies which files were changed. To determine the number of faults in each file during a given release, we counted the number of files listed in MRs issued by testers and those issued by customers [24]. Very few faults were typically reported by customers post-release. A total of 1048 faults were reported across the 35 releases, ranging from two faults in Release 33 and Release 35 to 84 in Release 5. The average number of faults per release was just under 30. Although this is the same system that we studied earlier [25], the fault counts are different here, because we included only source files written in C, C++ or C++ with embedded SQL, while the earlier study included all executable files such as shell scripts.

4.2. Data collection

To see whether the use of calling structure attributes can improve the accuracy of fault prediction beyond the accuracy obtained by previous fault prediction models, we collected code and history attributes combined with calling structure information.

Code attributes are ones that can be determined from the source code of the current or previous releases. These include size (KLOC), file type, new file status, whether the file was new in the prior release, and many of the calling structure attributes. History attributes include developer information, the number of faults in prior releases, and the number of changes made in prior releases. Note that a simple comparison of two versions of a file is enough to determine whether or not a file has changed from one release to the next, but cannot reveal how many changes were made.

Some of the calling structure attributes described in Section 3 can be obtained directly from the source code (e.g., the number of callees), while some of these attributes require both code and history information (e.g., the number of faulty callees in the prior release).

Table 1. List of attributes used in this study

Attribute	Definition	T
Non-calling structure attributes		
KLOC	Thousand lines of source code	C
RelNum	Release number	C
NewFile	New file status (1: new file, 0: existing file)	C
PriorNewFile	New file status in the prior release	C
CPP_File	Boolean indicator of C++ file type	C
C_File	Boolean indicator of C file type	C
SQLC_File	Boolean indicator of embedded SQL file type	C
PriorFaults	Number of faults in the prior release	H
PriorChanges	Number of changes in the prior release	H
PriorDevelopers	Number of developers in the prior release	H
CumDevelopers	Cumulative number of developers changing file prior to the current release	H
Calling structure attributes		
Callees	Number of callee files	C
Callers	Number of caller files	C
NewCallees	Number of new callee files	C
NewCallers	Number of new caller files	C
PriorNewCallees	Number of new callees in the prior release	C
PriorNewCallers	Number of new callers in the prior release	C
Cohesion	Ratio of number of internal calls in a file to the total number of calls associated with the file	C
PriorChangedCallees	Number of changed callee files in the prior release	C
PriorChangedCallers	Number of changed caller files in the prior release	C
PriorFaultyCallees	Number of faulty callee files in the prior release	B
PriorFaultyCallers	Number of faulty caller files in the prior release	B

To collect calling structure attributes, we used a publicly available tool, Doxygen [10], which generates a reference manual from documented source code. Doxygen can also generate code structure information

including the callers and callees of a method. We abstracted the method level calling structure generated by Doxygen to the file level. Table 1 lists the attributes we used in this study as independent variables for prediction models. The column labeled T represents the type of attribute*:* C (code), H (history), or B (both). To reduce the skewness of data, we performed a log transformation on KLOC and a square root transformation on PriorFaults, PriorChanges, PriorDevelopers, CumDevelopers, and all the calling structure attributes except Cohesion.

4.3. Correlations

Table 2 shows correlation coefficients between the number of faults in a file and the attributes. Spearman rank correlation is the correlation between the ranks of the measures of attributes, rather than on the magnitude of their measures, and is not sensitive to outliers. For completeness, we also provide Pearson correlation coefficients, which reflect the extent of linear association between attributes. All of the attributes were positively correlated with faults except for CPP_File, C_File, and Cohesion (Pearson correlation).

Table 2. Correlation between faults and attributes

Attribute	Spearman Rank Correlation	Pearson Correlation
KLOC	**0.221**	**0.231**
SQLC_File	**0.151**	**0.135**
CPP_File	**-0.130**	**-0.118**
C_File	**-0.050**	**-0.040**
PriorChanges	**0.302**	**0.314**
PriorFaults	**0.306**	**0.365**
NewFile	**0.045**	**0.032**
PriorNewFile	**0.039**	**0.049**
PriorDevelopers	**0.300**	**0.303**
CumDevelopers	**0.171**	**0.228**
Callees	**0.178**	**0.195**
Callers	**0.072**	**0.084**
NewCallees	**0.067**	**0.028**
NewCallers	**0.040**	0.015
PriorNewCallees	**0.080**	**0.081**
PriorNewCallers	**0.056**	**0.057**
PriorChangedCallees	**0.246**	**0.253**
PriorChangedCallers	**0.181**	**0.160**
PriorFaultyCallees	**0.254**	**0.278**
PriorFaultyCallers	**0.194**	**0.184**
Cohesion	**0.040**	-0.008

Correlations over 0.4 are considered to be strong in fault prediction studies [18, 26]. The Spearman correlation coefficients in our study ranged between 0.040 and 0.306 and were therefore not strong. However, all of the correlations except for NewCallers and Cohesion were statistically significant at the 0.01 level, which indicates that the correlation was not likely to have occurred by chance. These are shown in bold in Table 2.

Even though the correlations are not strong, we nonetheless built a fault prediction model because our goal is not just to look at the improvement caused by using calling structure information when building fault prediction models, but also to look at the relative effectiveness of each attribute in the predictions. Besides, as we will see in the following sections, low correlation does not necessarily lead to low prediction accuracy, depending on the goal of the prediction. This has also been noted by Ohlsson and Alberg [18].

One interesting finding is that the correlations between faults and callee-type attributes (Callees, PriorNewCallees, etc) was always higher than the correlations between faults and the corresponding caller-type attributes (Callers, PriorNewCallers, etc), indicating that the number of methods called by a method is a better fault indicator than the number of its callers.

5. Modeling and prediction

Our prediction goal is to identify a small percentage of files that contain most of the system test and post-release faults. We use negative binomial regression (NBR) models for fault prediction because they model situations that have nonnegative integers as outcomes, and also have provided high accuracy in fault prediction in prior studies [23]. Like Poisson regression, NBR models the expected value of the dependent variable as the exponential of a linear combination of the independent variables as in formula (1). In other words, the log of the number of expected faults, μ, is a linear combination of variables. NBR is useful for modeling distributions of data whose sample variance is greater than that predicted by a simple Poisson model.

$$\mu = e^{\alpha+\beta_1 X_1+\beta_2 X_2+\beta_3 X_3+\ldots} \quad (1)$$

After estimating the number of faults using the negative binomial model, we sort the files in decreasing order of the estimated number of faults and calculate the percentage of correctly predicted faults that are in the top 20% of the ordered files. To predict faults in Release N, we build models using the data obtained from Release 1 to Release (N-1). Then, we apply the

model to the data collected for Release N before the beginning of the system testing period for Release N.

In Section 6.1 we present univariate models for each of the attributes listed in Table 1. In succeeding sections, we describe multivariate models that use combinations of code, history, and calling structure attributes. Non-calling structure code attributes are used in all the models.

The multivariate models are constructed by starting with a univariate *base model*, and then adding one variable at a time. The base model uses the single attribute among those being considered that provides the highest prediction accuracy, and each succeeding step adds the single attribute that most improves the accuracy. In Table 5 and Table 6 the starting attribute is PriorChanges, while in Table 7, it is KLOC.

We add the newly chosen attribute only when the attribute is statistically significant and the prediction accuracy of the model improves by adding the attribute. Otherwise, we add the second best attribute. By looking at the effect of each attribute in the context of multivariate models, we can estimate the marginal value of the attribute in the presence of other attributes.

The model construction stops when the accuracy of the last model cannot be improved by adding any more variables.

6. Prediction results

In this section, we present the prediction results using negative binomial regression. We first investigate the effectiveness of individual code and history attributes in Section 6.1. In Section 6.2, we investigate the effectiveness of code and history attributes, and in Section 6.3, we include the calling structure attributes. In Section 6.4, we examine the effectiveness of models based only on code and calling structure, without history attributes. Because history information is not always readily available and may be cumbersome to collect, an organization might prefer to use attributes that can be obtained from source code alone, especially if the prediction results are close to those from the models requiring history attributes.

We controlled all the models with RelNum to avoid attributing patterns associated with a particular release to some other variables. For example, after controlling for release number, new files typically have much higher fault rates than old files. However, Release 1 contains a large proportion of new files. If Release 1 had very few faults, a model without release number would draw the incorrect conclusion that new files tend to have relatively few faults.

6.1. Univariate models

We began by building univariate models to determine the fault prediction ability of each attribute alone. Table 3 shows the percentage of faults within the predicted top 20% of files. The results are averages over Releases 3 through 35, and are listed in descending order of the prediction accuracy.

As seen in Table 3, a model based on only the number of changes in the prior release, used to identify the "worst" 20% of the files, identified files containing 68.82% of the actual faults. The second best predictor was the number of developers in the prior release, and the third one was the size (KLOC). Even though the correlation was not strong, the prediction accuracy was fairly high for some attributes including KLOC.

Table 3. Percentage of faults within the predicted top 20% of files, univariate models

Attribute	R3-35
PriorChanges	68.82
PriorDevelopers	67.44
KLOC	67.00
CumDevelopers	63.22
PriorChangedCallees	60.09
Callees	57.44
PriorFaultyCallees	54.40
PriorFaultyCallers	50.84
PriorChangedCallers	48.96
PriorFaults	48.04
Callers	38.08
SQLC_File	37.18
CC_File	33.02
PriorNewCallers	28.95
PriorNewCallees	26.68
NewCallees	25.58
PriorNewFile	24.55
NewFile	24.15
NewCallers	23.48
C_File	23.36
Cohesion	10.56

Notice that it is possible to consider this type of fault prediction that looks at relatively small numbers of files because faults tend to occur with a Pareto-like distribution [19]. For example, Table 4 shows the cumulative distribution of file sizes and faults for Release 20. Files with more than 1000 LOC made up 22.8% of the release, and contained 83.8% of the faults (shown as the shaded rows of Table 4). However, 95.8% of all the files contained no faults, regardless of their sizes. This example helps explain why we can often predict a high percentage of faults within a small

percentage of files. If faults were more uniformly distributed, this would not be possible.

Table 4. Distribution of faults in Release 20

File Size	Files	Cum Files (%)	Faults	Cum Faults (%)	Files with Faults
> 8001	4	1.1	12	37.4	4
4001 – 8000	4	2.2	6	48.6	2
2001 – 4000	21	8.1	8	70.3	3
1501 – 2000	23	14.4	0	70.3	0
1001 – 1500	30	22.8	5	83.8	2
501 – 1000	97	49.7	6	100	4
0 – 500	181	100	0	100	0
Total	360		37		15 (4.2%)

6.2. Fault prediction without calling structure attributes

Table 5 shows fault prediction results for multivariate models that do not contain calling structure attributes. We call these the "H" models, since they are based on History attributes (in addition to code attributes). The first column indicates the model ID. The second column presents the variable that was newly added to the model in the previous row. The third column provides the average, over Release 3 to Release 35, of the percentage of faults located in the predicted top 20% of files. The fourth column (Δ) represents the percentage that a model's prediction accuracy was improved over the previous row's accuracy. In the Total Improvement row, column 3 is the increase in prediction accuracy from the base model to the final model, and column 4 is the percentage increase of the final row's accuracy from the base model. The percentage of total improvement in the fourth column is computed by dividing the total improvement in the third column by the value in the third column for the base model.

Table 5. Prediction results without calling structure attributes

ID	Model	R3-35	Δ (%)
H1	**PriorChanges + (RelNum)**	68.82	-
H2	H1 + **KLOC**	72.18	4.9
H3	H2 + **SQLC_File**	75.11	4.1
H4	H3 + **NewFile**	77.44	3.1
H	H4 + **PriorNewFile** + **PriorFaults** + **CumDevelopers**	77.55	0.1
	Total Improvement	8.73	12.7

KLOC, SQLC_File, and NewFile each improved the accuracy of the base model and the succeeding models by several points, while PriorNewFile, PriorFaults, and CumDevelopers together improved the accuracy of Model H4 by only 0.1%. The final model was improved by 12.7% from the base model. This result is interesting because using only one history attribute (PriorChanges) and three code attributes (KLOC, SQLC_File, and NewFile), without other history information such as developer information or prior fault information, provided close to the best results.

6.3. Fault prediction with calling structure attributes

Table 6 shows the prediction results obtained when we used all the attributes. These are the "HC" models, because they are based on both History and Calling structure (plus code attributes). The fourth column (Δ) again represents the improvement of predictive accuracy from the model in the previous row. The fifth column (ΔC) represents the improvement of predictive accuracy by adding calling structure attributes.

The attributes selected by the method described in Section 5 were the same as the H Models through the first four attributes. PriorChangedCallees was the most effective attribute that could be added to Model HC4. The overall model improved by 13.8% from the base model, and 1.0% from the model without calling structure attributes (Model H). The improvement that can be attributed to the use of calling structure information was only 0.6%.

Table 6. Prediction results with all attributes including calling structure

ID	Model	R3-35	Δ (%)	ΔC (%)
HC1	**PriorChanges + (RelNum)**	68.82	-	-
HC2	HC1 + **KLOC**	72.18	4.9	-
HC3	HC2+ **SQLC_File**	75.11	4.1	-
HC4	HC3+ **NewFile**	77.44	3.1	-
HC5	HC4+ **PriorChangedCallees**	77.89	0.6	0.6
HC6	HC5+ **PriorFaults**	78.00	0.1	-
HC	HC6+ **PriorNewFile**	78.29	0.4	-
	Total Improvement	9.47	13.8	0.6

Figure 2 shows the prediction results for the final models with and without calling structure information (Model HC and Model H). The figure makes it easy to see that their performance is very similar over all the releases of the system, and that neither model was

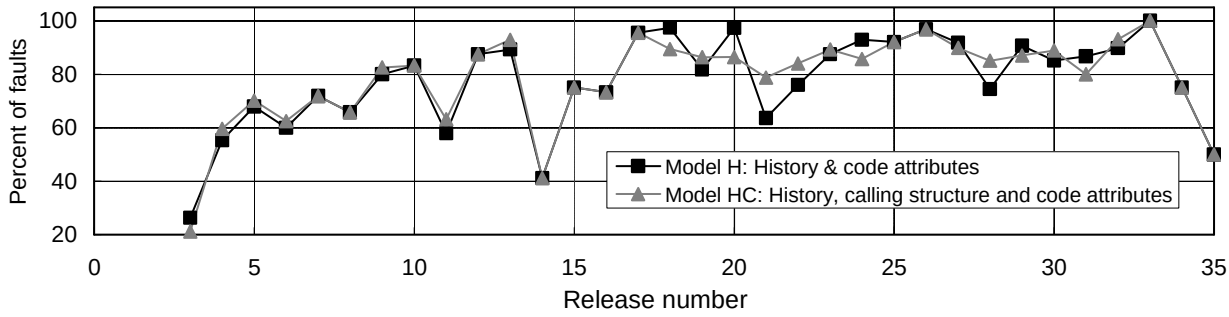

Figure 2. Comparison of models without and with calling structure attributes

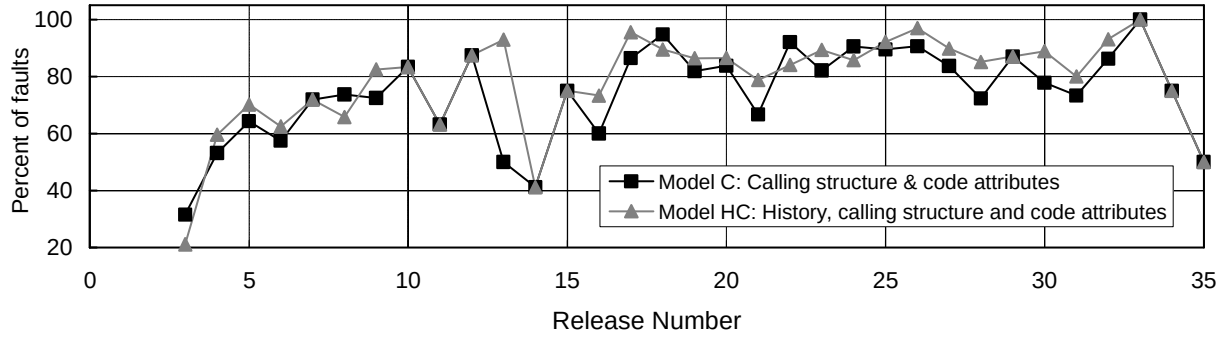

Figure 3. Comparison of models without and with history attributes

consistently superior. For all of the models, the prediction accuracy tends to improve until Release 13. The reason might be because the models have more training data available as the system matured. However, after Release 13, the prediction accuracy did not show clear evidence of improvement. We have not yet determined why the predictions for Release 14 are much less accurate than for other releases.

6.4. Fault prediction with code attributes alone

Even though the results in Section 6.2 and Section 6.3 show that history information is useful in predicting faults, many projects do not maintain bug tracking systems or specific bug locations are not available at the file level. Even when they are available, data mining to collect history information about the changes, faults, and developer activities can be difficult and expensive. In this section, we evaluate prediction models that use attributes obtainable only from source code, including calling structure. Table 7 shows the results observed using these "C" models. The effective calling structure attributes for the final model were PriorNewCallees, Cohesion, and PriorNewCallers. Cohesion had a negative coefficient as we expected from Section 3. PriorNewCallees also had a negative coefficient. Other attributes had positive coefficients.

Table 7. Prediction results using code attributes alone

ID	Model	R3-35	Δ (%)	ΔC (%)
C1	**KLOC + (RelNum)**	67.00	-	-
C2	C1+ **PriorNewCallees**	69.04	3.0	3.0
C3	C2+ **SQLC_File**	70.77	2.5	-
C4	C3 + **Cohesion**	71.73	1.4	1.4
C5	C4 + **PriorNewFile**	72.82	1.5	-
C6	C5 + **PriorNewCallers**	73.33	0.7	0.7
C	C6 + **NewFile**	74.19	1.2	-
	Total Improvement	7.19	10.7	5.1

The prediction accuracy of the final C model is 10.7% better than the base KLOC-only model. The improvement made purely by calling structure attributes was 5.1%. Model C, using code and calling structure attributes, provided 2.8% better results than the history-based model using the two most effective predictors, PriorChanges and KLOC (Model H2).

However, the prediction accuracy of the final history-based model without calling structure (Model H) is still 4.5% better than the prediction accuracy of Model C. Overall, the model with all the code and history attributes plus the calling structure attributes (Model HC) was 5.5% better than Model C. Despite the overall 5.5% difference, the release to release comparison in Figure 3 shows that these two models track each other fairly closely. For 10 of the 33 releases the two models achieved identical results. For 5 releases, C's results exceeded those of HC, and for the remaining 18, HC exceeded C. The differences are usually minor, with the exception of the 43 point difference for release 13.

7. Threats to validity

Our study has the following limitations. First, because our database identifies only the files that are faulty, not the specific methods, our fault analysis and prediction is done for files. Even though most faults are concentrated in a small percentage of files (in a Pareto distribution), knowing that a particular file is likely to have a fault does not mean that all methods in that file are faulty. Obviously method-level analysis and prediction would be more precise and more helpful for users. In principle, the analysis done here at the file level would be easily extendible to methods, whenever fault identification at the method level is available. Second, the results hold only for the attributes, models, data set, and manner of assessment used in this study. For example, if the goal of prediction is classification of fault-prone files instead of predicting files that contain the largest number of faults, the effectiveness of attributes including the calling structure attributes might be different from what we observed. To find more general evidence, we must replicate the study using a wider range of projects and modeling techniques. In addition, we only considered files written in C and two variants of C++. Files written in other languages might have very different results.

Third, the code and history attributes we used in this study are not the only ones that could be used. Therefore, the comparison of attributes should be interpreted only within the attribute set used in this study. However, including too many variables in a model could result in overfitting, which makes the model less effective when the model is applied to other data sets. We tried to use representative code and history attributes that have been used in the past and keep the model simple to avoid overfitting as well as to improve understanding of the effects of each attribute.

Fourth, the calling structure information was analyzed only from the accessible source code. Method invocations to the system libraries and to third party libraries were not included in our analysis, since we have fault information available only from the source code. Therefore, the prediction results can only be accurate within the scope of information currently obtainable. Finally, calling structure measures might not be precise when the method binding is performed dynamically at runtime because we use only statically collectable information.

8. Conclusions

In this paper, we described an empirical study to predict faults using calling structure information abstracted at file level. This study not only focused on the overall improvement in accuracy of fault prediction using calling structure information, but also investigated the effectiveness of each attribute of code and history collected in the context of multivariate models to see which attributes are still effective when other, possibly correlated, attributes are used together. We performed negative binomial regression using models with code and history attributes with and without calling structure information. We then compared the effectiveness of the models by showing the improvement in the prediction accuracy when calling structure information was added.

When averaged over Releases 3-35, the addition of calling structure information provided only marginal improvement of 0.6% in prediction accuracy over models based only on non-calling structure code attributes and history attributes. Many of the calling structure attributes were effective fault predictors individually, but when used in combination the results improved only very slightly or not at all. These findings indicate that using calling structure information for fault prediction might not be worthwhile when history information is available.

The original history model, without any calling structure attributes, provided a 4.5% better prediction accuracy than the best model based only on source code attributes including calling structure. These results indicate that history information should be used for fault prediction whenever possible.

If history information is not available, then calling structure information provides added value to a model based only on the non-calling structure code attributes.

This paper's results, which are based on data from only one system, are not guaranteed to hold universally. We expect to repeat the study of calling structure attributes on additional systems of various types, and attempt to determine which system characteristics lead to similar results.

9. Acknowledgment

This work is supported in part by the National Science Foundation Grant No. 0716176. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

10. References

[1] "The Economic Impacts of Inadequate Infrastructure for Software Testing", National Institute of Standards & Technology, May, 2002.

[2] Arisholm, E., and Briand, L.C., "Predicting Fault-prone Components in a Java Legacy System", *Proc. the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06)*, Rio de Janeiro, Brazil, Sep. 21-22 2006, pp. 8 - 17.

[3] Basili, V.R., Briand, L.C., and Melo, W.L., "A Validation of Object-Oriented Design Metrics as Quality Indicators", *IEEE Trans. Software Eng.*, 22(10), 1996, pp. 751 - 761.

[4] Basili, V.R., and Perricone, B.R., "Software Errors and Complexity: An Empirical Investigation", *Comm. ACM*, 27, 1984, pp. 42-52.

[5] Bell, R.M., Ostrand, T.J., and Weyuker:, E.J., "Looking for Bugs in All the Right Places", *Proc. the 2006 International Symposium on Software Testing and Analysis (ISSTA'06)*, Portland, Maine, USA, July 17-20, 2006, pp. 61 - 72.

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

[7] Briand, L.C., Wust, J., Ikonomovski, S.V., and Lounis, H., "Investigating Quality Factors in Object-Oriented Designs: an Industrial Case Study", *Proc. the 1999 International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, USA, 16-22 May, 1999, pp. 345-354.

[8] Fenton, N.E., and Ohlsson, N., "Quantitative Analysis of Faults and Failures in a Complex Software System", *IEEE Trans. Software Eng.*, 26(8), 2000, pp. 797 - 814.

[9] Graves, T.L., Karr, A.F., Marron, J.S., and Siy, H., "Predicting Fault Incidence Using Software Change History", *IEEE Trans. Software Eng.*, 26(7), 2000, pp. 653-661.

[10] van Heesch, D., Doxygen, http://www.stack.nl/~dimitri/doxygen/

[11] Henry, S., and Kafura, D., "Software Structure Metrics Based on Information Flow", *IEEE Trans. Software Eng.*, 7(5), 1981, pp. 510-518.

[12] Jiang, Y., Cukic, B., Menzies, T., and Bartlow, N., "Comparing Design and Code Metrics for Software Quality Prediction", *Proc. the 4th International Workshop on Predictor Models in Software Engineering (PROMISE'08)*, Leipzig, Germany, May 12–13, 2008, pp. 11-18.

[13] Khoshgoftaar, T.M., Allen, E.B., Kalaichelvan, K.S., and Goel, N., "Early Quality Prediction: A Case Study in Telecommunications", *IEEE Software*, 13, (1), Jan., 1996, pp. 65 - 71.

[14] Menzies, T., Greenwald, J., and Frank, A., "Data Mining Static Code Attributes to Learn Defect Predictors", *IEEE Trans. Software Eng.*, 33(1), 2007, pp. 2 - 13.

[15] Nagappan, N., and Ball, T., "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study", *Proc. First International Symposium on Empirical Software Engineering and Measurement*, Madrid, Spain, 20-21 Sept., 2007, pp. 364 - 373.

[16] Nagappan, N., Ball, T., and Murphy, B., "Using Historical In-Process and Product Metrics for Early Estimation of Software Failures", *Proc. the 17th International Symposium on Software Reliability Engineering*, Raleigh, NC, U.S.A., November 7-10, 2006, pp. 62-74.

[17] Nagappan, N., Ball, T., and Zeller, A., "Mining Metrics to Predict Component Failures", *Proc. the 28th International Conference on Software Engineering (ICSE'06)*, Shanghai, China, May 20-28, 2006, pp. 452-461.

[18] Ohlsson, N., and Alberg, H., "Predicting Fault-Prone Software Modules in Telephone Switches", *IEEE Trans. Software Eng.*, 22(12), 1996, pp. 886-894.

[19] Ostrand, T.J., and Weyuker, E.J., "The Distribution of Faults in a Large Industrial Software System", *Proc. the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'02)*, Roma, Italy, July 22-24, 2002, pp. 55 - 64.

[20] Ostrand, T.J., Weyuker, E.J., and Bell, R.M., "Predicting the Location and Number of Faults in Large Software Systems", *IEEE Trans. Software Eng.*, 31(4), 2005, pp. 340 - 355.

[21] Stevens, W.P., Myers, G.J., and Constantine, L.L., "Structured Design", *IBM Systems Journal*, 13(2), 1974, pp. 115-139.

[22] Subramanyam, R., and Krishnan, M.S., "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects", *IEEE Trans. Software Eng.*, 29(4), 2003, pp. 297-310.

[23] Weyuker, E., Ostrand, T., and Bell, R., "Comparing Negative Binomial and Recursive Partitioning Models for Fault Prediction", *Proc. the 4th International Workshop on Predictor Models in Software Engineering (PROMISE'08)*, Leipzig, Germany, May 12-13, 2008, pp. 3-10.

[24] Weyuker, E.J., and Ostrand, T.J., "Comparing Methods to Identify Defect Reports in a Change Management Database", *Proc. International Workshop on Defects in Large Software Systems (DEFFECTS'08)*, Seattle, WA, July 20, 2008

[25] Weyuker, E.J., Ostrand, T.J., and Bell, R.M., "Using Developer Information as a Factor for Fault Prediction ", *Proc. International Workshop on Predictor Models in Software Engineering (PROMISE '07)*, Minneapolis, MN, 20 May, 2007, pp. 8.

[26] Zimmermann, T., and Nagappan, N., "Predicting Defects Using Network Analysis on Dependency Graphs", *Proc. the 13th International Conference on Software Engineering (ICSE'08)*, Leipzig, Germany, 10 - 18 May, 2008, pp. 531-540.