Using Static Analysis to Determine Where to Focus Dynamic Testing Effort

Thomas J. Ostrand
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
ostrand@research.att.com

Elaine J. Weyuker
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
weyuker@research.att.com

Robert M. Bell
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
rbell@research.att.com

Abstract

We perform static analysis and develop a negative binomial regression model to predict which files in a large software system are most likely to contain the largest numbers of faults that manifest as failures in the next release, using information from all previous releases. This is then used to guide the dynamic testing process for software systems by suggesting that files identified as being likely to contain the largest numbers of faults be subjected to particular scrutiny during dynamic testing. In previous studies of a large inventory tracking system, we identified characteristics of the files containing the largest numbers of faults and those with the highest fault densities. In those studies, we observed that faults were highly concentrated in a relatively small percentage of the files, and that for every release, new files and old files that had been changed during the previous release generally had substantially higher average fault densities than old files that had not been changed. Other characteristics were observed to play a less central role. We now investigate additional potentially-important characteristics and use them, along with the previously-identified characteristics as the basis for the regression model of the current study. We found that the top 20% of files predicted by the statistical model contain between 71% and 85% of the observed faults found during dynamic testing of the twelve releases of the system that were available.

Keywords: Software Faults, Fault-prone, Prediction, Regression Model, Empirical Study, Software Testing.

1. Introduction and Earlier Work

Much of today's industry relies on software systems, and requires that they behave correctly, perform efficiently, and can be produced economically. For these reasons, it is important that we dynamically test systems to identify faults residing in the code. For large systems, this can be a very expensive and difficult process. Therefore, we want to determine which files in the system are most likely to contain the largest numbers of faults that lead to failures and prioritize our testing effort accordingly. In that way we minimize the cost of testing and maximize the effectiveness of the process. In order to do this, we have been investigating how to use data residing in a combined version control and change management system used during all stages of development, testing, and field release, to improve dynamic testing.

Preliminary work was reported in an earlier paper [10] which described a case study involving an industrial inventory tracking system, developed over a three year period, covering twelve quarterly releases. The goal of that research was to do static analysis to identify structural characteristics that are associated with files that contain particularly large numbers of faults as determined by reported failures. The data used for the static analysis resides in the combined version control/change management system with some of the data determined by statically analyzing the code while other data were identified during the dynamic testing phase.

Data in this repository were collected during each of nine development phases including requirements, design, development, unit testing, integration testing, system testing, beta release, controlled release, and general release. In this paper we will describe the use of this information to develop a statistical model to predict where faults are most likely to reside in the code, which in turn can be used as an integral part of the dynamic testing process. Thus our process relies on a complex interplay between static and dynamic analysis, and data associated with both of these types of analysis.

Our earlier studies considered the extent to which faults clustered in a small proportion of files, and looked at file characteristics such as size, age, whether the file is new to the current release, and if not, whether it was changed during the prior release, the number and magnitude of changes made to a file, the number of observed faults during early releases, and the number of faults observed during early development stages.

Most of the previous research in this area, including our earlier work [10], and that by other authors described in [1, 2, 3, 5, 8, 9], was aimed at examining software systems to establish *characteristics* that may be associated with high incidences of faults. In this paper, we go beyond merely identifying characteristics and successfully build a statistical model that can *predict* the incidence of faults in future versions of a system. Specifically, this model is used to predict the number of faults that will occur in each file during the next release, based on current characteristics of the file and its behavior in earlier releases. By selecting the set of files that are predicted to account for a large percentage of the faults in the next release, we can encourage testers to use that information to prioritize and focus their (dynamic) testing efforts.

Thus our goal is to accurately identify a relatively small percentage of the files that contain a large percentage of the faults. Of course, there is no guarantee that all faults, or even the most dangerous faults, will be located by this approach. However, if the prediction allows a large majority of all outstanding faults to be identified more rapidly than they would otherwise be found, then more resources will be available for additional testing to ferret out the remaining ones, or the process can be completed more quickly, and hence cheaply, with equal success.

The work by Graves et al. [4] is most similar to ours, as they also construct models to predict fault-proneness. In contrast to Graves et al., however, our model makes predictions for individual files of the system, rather than for modules that are collections of files as was done in [4]. The fact that the granularity of the entities we use in our static analysis is significantly finer than that used by Graves et al. is important since it should facilitate the identification of faults in a much more localized portion of the code, thereby making debugging easier as well.

Other differences between our work and that done by the Graves et al. group include the fact that they attempted only a single prediction while our case study makes predictions for each release beginning with Release 3, and continuing through Release 12, allowing us to validate the effectiveness of our model over a sustained period of time, with the system at different levels of maturity. Also, their models use the fault history of a single two-year period to predict faults in the following two-year period, while our model uses data from much shorter 3-month intervals to predict faults in the following quarterly releases. This shorter interval provides much more timely information to testers, who can use the prediction from the current and prior releases to help focus their testing efforts. In fact the goal of our work is to design a process that can be used as a standard part of the development process in an industrial environment to improve and streamline the testing of systems requiring very high reliability.

Our earlier study considered a file's *fault density*, computed in terms of faults per thousand lines of code (KLOCs). In Section 3 of this paper we will describe our findings related to several new questions regarding the number of faults in a file. Among the new factors we consider is whether there was a relationship between the complexity of the file and the number of faults in a file, where complexity is measured by the cyclomatic number [6] rather than the number of lines of code. We also investigate the role of the choice of programming language, the fault history in the file during the previous release, and the amount of change during the previous release.

As mentioned above, our ultimate goal is to be able to identify a particular set of files in a new release that are determined by our statistical model to be the most likely ones to account for the largest numbers of faults. Since we have determined in our earlier study that faults typically have a highly skewed distribution, this should be possible to accomplish.

The remainder of the paper is organized as follows: In Section 2, we describe the software system that is the subject of our case study and present some basic information about file characteristics and the faults identified during testing. Section 3 illustrates associations between selected file characteristics and the number of faults identified during a particular release. In Section 4 we present findings from a negative binomial regression model to predict the number of faults, in order to analyze relationships while controlling for other characteristics. Finally, Section 5 presents conclusions and describes plans for extending this work.

2. The System Under Study

The system used in this study is the same inventory tracking system as was used during the preliminary study [10]. As a standard part of the operating procedure for most development projects at AT&T, whenever any change is to be made to a software system, a Modification Request (MR) is entered in the combined version control and change management system. Each MR includes information describing the file(s) to be changed, the nature of the change (for example, is this a new file being added, or a modification of an existing one), the details of the change including specific lines of code to be added, deleted, or changed, a description of the change, and a severity indicating the importance of the proposed change. These data are collected as part of the normal development process and were therefore available for every release of the system. It is these data that we will statically analyze in order to use it to streamline dynamic testing.

Some parts of the MR, such as the severity rating, are highly subjective, and therefore may not be particularly useful. Unfortunately, the standard MR format does not require the person initiating the request to indicate whether

Rel	Number of Files	Lines of Code	Mean LOC	Faults Detected	Fault Density	System Test and Later Fault Density
1	584	145,967	250	990	6.78	1.49
2	567	154,381	272	201	1.30	0.16
3	706	190,596	270	487	2.56	0.45
4	743	203,233	274	328	1.61	0.17
5	804	231,968	289	340	1.47	0.19
6	867	253,870	293	339	1.34	0.18
7	993	291,719	294	207	0.71	0.10
8	1197	338,774	283	490	1.45	0.25
9	1321	377,198	286	436	1.16	0.16
10	1372	396,209	289	246	0.62	0.09
11	1607	426,878	266	281	0.66	0.21
12	1740	476,215	274	273	0.57	0.15

Table 1. System Information

the change is due to a fault correction or to some other reason such as performance improvement, cleaning up the code, or changed functionality. We have now succeeded in getting the MR form changed to include a field that explicitly states whether the MR was due to the identification of a fault, but this was not available at the time that the data described here were entered or collected, and so we needed a way of making that determination.

Since our study included a total of roughly 5,000 faults, and many more MRs that were categorized as being other sorts of changes, it was impossible to read through every MR to make that determination. We therefore needed a heuristic and used a rule of thumb suggested by the testing group that an MR likely represents a fault correction if either exactly one or two files were modified. In an informal attempt to validate this hypothesis, we sampled a small number of MRs by carefully reading the text description of the change. In the small sample space, nearly every MR that modified one or two files was indeed a fault fix, and every MR that modified a larger number of files (sometimes as many as 60 files) was not a fault correction, but rather a modification made for some other reason. For example, if a new parameter was added to a file, every file that called it had to be modified accordingly.

Changes can be initiated during any stage from requirements through general release. For most development environments, change recording begins with integration or system test, when control leaves the development team and moves to an independent testing organization. For this system, however, MRs were written consistently from requirements on. Almost three quarters of the faults included in this study were identified during unit testing done by developers.

The final version of the system used in these studies (Release 12) included more than 1,700 separate files, with a total of more than 476,000 lines of code. Roughly 70% of these files were written in java, but there were also small numbers of shell scripts, makefiles, xml, html, perl, c, sql, awk, and other specialized languages. Non-executable files such as MS Word, gif, jpg, and readme files were not included in the study.

Over the three year period that we tracked this system, there was a roughly three-fold increase in both the number of files and lines of code. At the same time, there was a significant concentration of identified faults in files, going from appearing in 40% of the files in Release 1 to only 7% of the files by Release 12. One might hypothesize that the increased fault concentration was simply a reflection of the fact that the system was three times larger. However, when the absolute numbers of files containing faults was considered, this fault concentration was also apparent. For example, in Release 1, a total of 233 (of 584) files contained any identified faults, by Release 8 only 148 (of 1197) files contained any identified faults, and by Release 12, only 120 (of 1740) files contained any identified faults at all.

One important decision that had to be made involved exactly how to count the number of faults in a file. If n files were modified as the result of a failure, then this was counted as being n distinct faults. This is consistent with the convention used in References [8] and [3]. This implies that each fault was associated with exactly one file.

Table 1 provides summary information about the first twelve releases of the system, including lines of code and faults. New files typically represent new functionality, while changed files generally represent fault fixes. As the system matured and grew in size, the number of faults tended to fall, with the largest decrease occurring from Release 1 to Release 2. As one might expect, there is also a general downward trend in the fault density as the system matured, with some exceptions including Release 2.

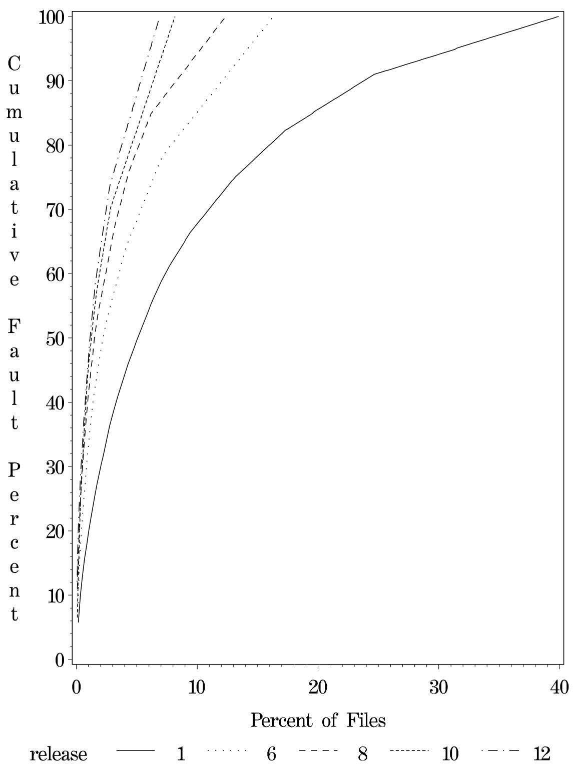

Figure 1. Fault Distribution for Releases 1, 6, 8, 10, 12

The large dip at Release 2 likely occurred because it was an interim release. While other releases all occurred at roughly three month intervals, Release 2 occurred between Releases 1 and 3, which were themselves done three months apart. This likely led to a decreased number of faults identified during Release 2, and hence a decreased fault density.

The last column of the table restricts attention to those faults identified during system test or later. As mentioned above, it is uncommon for faults identified during earlier stages of development to be included in a fault-reporting system. Therefore, the system test fault densities are likely to be more comparable to fault densities reported in other empirical studies. Recall too, that for this system, generally one quarter or fewer of the faults at any release were identified during system test or later.

3. Fault Concentration and Potential Explanatory Factors

In this section we discuss various potential additional factors not considered in our earlier work that might explain the differential fault concentration in files. Once these factors are understood, we will use them to build a statistical model that statically analyzes the software, to guide its dynamic testing.

3.1 Concentration of Faults

Ostrand and Weyuker [10] reported that faults for this system tended to concentrate in a relatively small proportion of files at every release. We repeat here Figure 1 which originally appeared in [10], showing the concentration of faults in Releases 1, 6, 8, 10, and 12. For clarity, we showed data for only a sampling of the releases. The selected releases are representative of the other releases that were not included. We found that when too many releases were shown on the same graph, it became impossible to distinguish among the lines and therefore the import of the data was lost.

The files in each release are sorted in decreasing order of the number of faults they contain. A point (x, y) on the Release R curve represents the fact that x% of the files in Release R contain y% of the faults. For example, at Release 1, the ten percent of files with the most faults (58 files) had 669 faults, representing 68% of the total for Release 1. The curves show that the proportion of faults tends to become increasingly concentrated in fewer files as the system matures.

3.2 File Size

In [10], we examined the relationship between file size and fault density and found that there was a tendency for

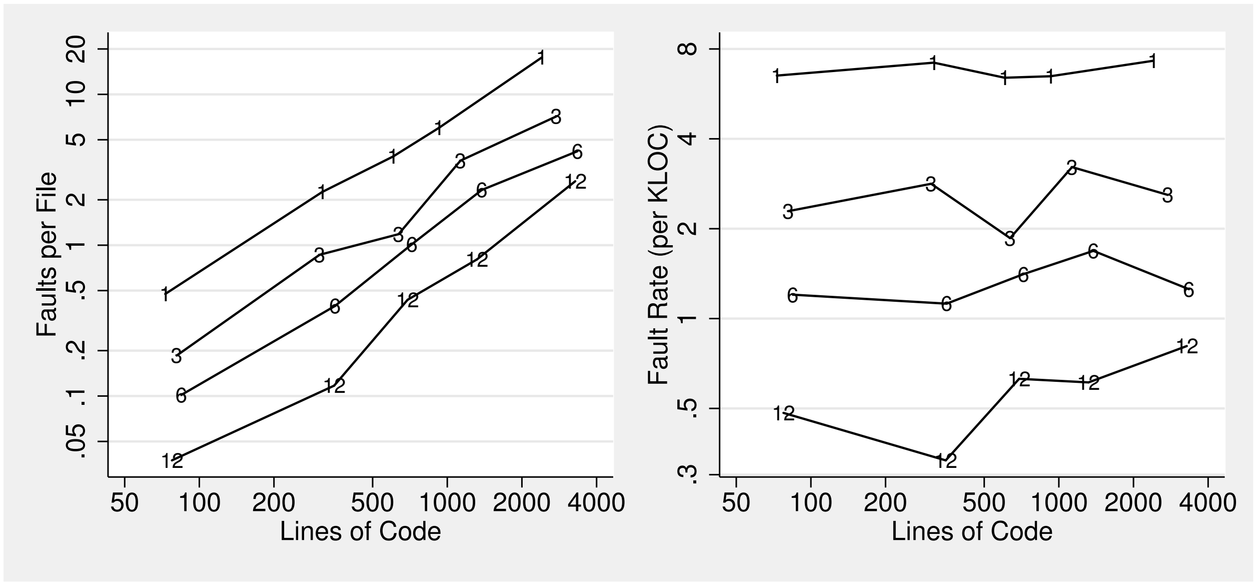

Figure 2. Faults and Files Grouped by Size

small files to have higher average fault densities than large files. We now consider the relationship between file size and the average number of faults per file, rather than considering the average fault density. This will be done by dividing the files for each release into bins. For a given release, we sort the files in increasing size order, divide all of these files into five bins with roughly equal numbers of total lines of code, and calculate the average faults per file for files in the bin.

For example, at Release 1, the first bin contains 398 files ranging in size from 5 to 202 lines of code, with an average size of 73 lines. Those files have a total of 189 faults, resulting in an average of 0.47 faults per file. Subsequent bins include progressively smaller numbers (94, 48, 32, and 12) of larger files, with increasingly more faults per file. This relationship is shown on a log-log scale for Releases 1, 3, 6, and 12. As noted earler, these releases were representative of all releases and were selected to show releases at various stages of maturity. The lefthand portion of Figure 2 shows that there is a strong relationship between file size and the average number of faults per file. We also look at the fault density to see whether there are a "disproportionate" number of faults that occur in larger files than smaller ones, and if there are, whether it might make sense to limit the permitted size of files.

The righthand portion of Figure 2 shows fault densities versus file size for the same sets of bins and releases. The figure shows that there is little or no relationship between fault density and file size. Graphs for the releases not shown in this figure tell similar stories. Although the fault densities for a given release tend to be higher for the two bins containing the largest files than for the two bins containing the smallest ones, the relationship is not monotonic for any of the twelve releases. Specifically, across the releases, the bin containing the largest files has the highest fault density for only five of the twelve releases, and the bin containing the smallest files has the lowest fault density for only three of the twelve releases. Moreover, when results are aggregated across releases, fault densities for largest files are only about 20% higher than for the shortest files. We therefore conclude that file size is not a strong predictor of fault density, but might be a good predictor of the absolute number of faults in a file.

Note that there are two points of difference from our earlier analysis of fault density and size. First, in the present study, we look at the fault density data aggregated over files in a given size range rather than considering each file individually. Second, we include all files: those for which faults were detected, and those for which there were no faults detected. In the earlier work the fault density was computed only for those files that contained faults.

3.3 Program Type

Table 2 compares fault densities for the most commonly-used program types in this system. Because fault densities are much lower for existing files, this table only includes results for a file at the time of its first entry into the system (new files). The observed fault densities vary by a factor of close to 30, with makefiles having the highest average density and xml files the lowest.

Type	Files	LOC	Faults	Fault Density
makefile	94	2509	58	23.12
sh	140	7756	69	8.90
sql	80	6875	60	8.73
html	52	5639	22	3.90
java	1492	413420	1424	3.44
perl	68	17619	52	2.95
c	21	5824	8	1.37
xml	95	5070	4	0.79

Table 2. Fault Densities for New Files, by Program Type

4. Multivariate Analysis of the Number of Faults

In this section we present results from negative binomial regression models that predict the number of faults in a file during a release, as a function of various file characteristics. This modeling process serves three major purposes. First, it provides information about the association between the number of faults and individual file characteristics while holding other file characteristics constant. Most of this information is determined by statically analyzing the code. Information about fault counts is, of course, determined by dynamic execution of the code, primarily on test cases, but also during field operation. Data provided in [10] showed that most faults were detected during either unit or system testing, with only 2% of the faults detected during field operation. The second purpose of the modeling process is to provide a measure of the concentration of faults beyond what is accounted for by file characteristics. This allows us to compare the effectiveness of alternative sets of factors. Third, the model produces predictions of the most fault-prone files in a release, so that testing resources can potentially be targeted more effectively. The third purpose is the ultimate goal of this research. In Section 4.1, we outline the model, while in Section 4.2, we describe our findings. In Section 4.3, we assess the efficacy of this strategy.

4.1 The Negative Binomial Regression Model

Negative binomial regression is an extension of linear regression designed to handle outcomes like the number of faults [7]. It explicitly models counts or outcomes that are nonnegative integers. The expected number of faults is assumed to vary in a multiplicative way as a function of file characteristics, rather than in an additive relationship. Unlike the related modeling approach, Poisson regression, the negative binomial model allows for the type of concentration of faults apparent in Figure 1, in which we see a relatively small percentage of files containing a large percentage of faults. This is done by adjusting inference for the additional uncertainty in the estimated regression coefficients caused by overdispersion.

Let y_i equal the observed number of faults and x_i be a vector of characteristics for file i. The negative binomial regression model specifies that y_i, given x_i, has a Poisson distribution with mean λ_i. This conditional mean is given by $\lambda_i = \gamma_i e^{\beta' x_i}$, where γ_i is itself a random variable drawn from a gamma distribution with mean 1 and unknown variance $\sigma^2 \geq 0$. The variance σ^2 is known as the dispersion parameter, and it allows for the type of concentration we observed for faults. The larger the dispersion parameter, the greater the unexplained concentration of faults. However, to the extent that this concentration is explained by file characteristics x_i that are included in the model, the dispersion parameter will decline.

4.2 Results

We used a negative binomial regression model fit to files from Releases 1 to 12 with the unit of analysis being a file-release combination. This yielded a total of 12,501 observations. The outcome is the number of faults predicted to be associated with the file at the given release. All models were fit by maximum likelihood using the procedure Genmod in SAS/STAT Release 8.01 [11].

Predictor variables for the model are: the logarithm of lines of code; whether the file is new, changed or unchanged (the file's change status); age (number of previous releases the file was in); the square root of the number of faults in the previous release (if any); program type; and release. Logged lines of code (LOC), file age, and the square root of prior faults are treated as continuous variables. File change status, program type, and release are treated as categorical variables, each fit by a series of dummy (0-1) variables, with one omitted category that serves as the reference. For file change status, the reference category is unchanged files, so that the new and changed coefficients represent contrasts with existing, unchanged files. For program type, the reference category is java files, the most commonly-occurring

Predictor Variables	Dispersion Parameter	Amount Explained	Percentage Explained
Null	13.38	NA	NA
LOC	5.61	7.77	58.0
Release	11.00	2.38	17.8
File Change Status	7.29	6.09	45.5
Program Type	12.88	.51	3.8
Prior Faults	9.86	3.53	26.3
LOC, Release	3.91	9.47	70.8
LOC, Release, File Change Status	3.03	10.35	77.4
LOC, Release, File Change Status, Program Type	2.52	10.87	81.2
Full Model	2.27	11.11	83.0

Table 3. Estimated Dispersion Parameters Associated with Selected Models

Release	3	4	5	6	7	8	9	10	11	12
% Faults Identified	77	74	71	85	77	81	85	78	84	84

Table 4. Percentage of Faults Included in the 20% of the Files Selected by the Model

type for this system. We arbitrarily set Release 12 as the reference release.

The strongest predictor in the model is the number of lines of code. Because the model uses the logarithm of lines of code, a coefficient of 1.00 would imply that the expected number of faults grows proportionally with lines of code (i.e., that fault density is unrelated to lines of code). The estimated coefficient was 1.047 which exceeds 1.00. This therefore provides some evidence that fault density grows with lines of code, holding all else equal. We note, however, that the 95 percent confidence interval does include 1.00.

For categorical predictors, each coefficient estimates the difference in the logarithm of the expected number of faults for the corresponding category versus the reference category. For example, for changed files, the coefficient was 1.066. This indicates that changed files have about exp(1.066) = 2.90 times more faults than existing, unchanged files with otherwise similar characteristics. Of course, the changed files are more likely to have other characteristics (such as prior faults) indicating a propensity for faults at the current release.

Table 3 displays estimates of the dispersion parameter for a series of alternative models, to help show the relative improvement associated with individual, or groups, of predictor variables. The estimated dispersion parameter for a null model, with no predictors, is 13.38. The best single predictors were lines of code and the file's change status. Lines of code reduced the dispersion to 5.61, a reduction of 58.0%, while file change status explained 45.5% of the dispersion. Use of the full model reduced the dispersion parameter to 2.27, a reduction of 83.0%.

Various other potential predictor variables were tested, but dropped from the model because they did little to improve the predictive power when added to the model. Some of the variables that we decided to exclude because they did not significantly improve the predictive capability of the model included: the number of changes for files that changed since the previous release, whether or not the files had changed prior to the previous release, and the logarithm of the cyclomatic number (which was computed for java files only). The *cyclomatic number* measures the complexity of a file by counting the number of decision statements in the file [6]. It has been found to be very highly correlated with the number of lines of code. Although the cyclomatic number did predict faults well in a bivariate context, it helped very little when used in conjunction with lines of code (both logged), especially at later releases. In contrast, lines of code remained important even in conjunction with the cyclomatic number.

4.3 Targeting Fault-Prone Files for Testing

We now evaluate the potential of the regression model to improve testing productivity by prospectively identifying a subset of files that contain disproportionately many of the faults at the next release. At each release, beginning with Release 3, we created predictions based on fitting alternative models using data from only the previous releases (e.g., predictions for Release 3 used data from Releases 1 and 2). For each release, these predictions are used to order the files from most to least fault-prone, based on the predicted numbers of faults.

Table 4 shows the percentages of actual faults contained in the top 20 percent of files identified by the full model at each of Releases 3 to 12. The model prospectively identified between 71% and 85% of the faults in the system, with an average over all releases of 80%. Of course any percentage of the files could have been selected, but we determined that 20% was a good choice providing a large percentage of the faults while focusing on a relatively small percentage of the files.

5. Conclusions and Future Work

We have used static analysis to develop a negative binomial regression model as a way of predicting which files are most likely to contain the largest numbers of faults in a new release, and thereby prioritize effort during dynamic testing. This prediction was done for each release by using only data collected during earlier releases. Our initial model was quite successful in the sense that we were able to use it to accurately predict, on average, the 20% of the files that corresponded to 80% of the faults.

The factors that influenced our predictions include the size of the file, the file's change status, the number of faults in the previous release, the programming language, and the file's age. Unlike Graves et al. [4], we found that change history before the prior release was not needed in our models. This finding may be because our models are more specific in terms of content and time since we predict faults for individual files during a series of releases. Graves et al., in contrast, modeled faults for modules which are large groups of files, during a single two year period.

So far we have designed our model based on the characteristics identified as most relevant for the twelve releases of one software system. Although this is a substantial system that runs continuously, with quarterly new releases, there may be characteristics of this system that are atypical, and therefore the model may not be applicable to other systems without tuning. In addition, as the system ages, the most important factors may change somewhat. For this reason, it is important to apply our model to additional releases of the inventory tracking system, as well as to other systems with different characteristics, developed in different environments.

We have now collected data for an additional five releases of the current system, and identified two additional industrial software systems, each with multiple releases and years of field exposure, for which data collection and analysis have begun. Once this is complete, we will apply the current negative binomial regression model to the data collected from these systems and see whether the prediction is as successful as we observed for the first twelve releases of this system. If not, we may have to identify additional relevant characteristics or modify the role played by the factors by defining new weightings. We are also designing a tool to automate the application of our prediction model.

We consider our initial results extremely promising and look forward to the routine use of this sort of predictive modeling to focus software testing efforts, thereby improving both the efficiency and the effectiveness of our software testing process. We have found that using static analysis to guide and prioritize dynamic software testing is an excellent way of improving the testing process for this system.

References

[1] E.N. Adams. Optimizing Preventive Service of Software Products. *IBM J. Res. Develop.*, Vol28, No1, Jan 1984, pp.2-14.

[2] V.R. Basili and B.T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, Vol27, No1, Jan 1984, pp.42-52.

[3] N.E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Trans. on Software Engineering*, Vol26, No8, Aug 2000, pp.797-814.

[4] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting Fault Incidence Using Software Change History. *IEEE Trans. on Software Engineering*, Vol 26, No. 7, July 2000, pp. 653-661.

[5] L. Hatton. Reexamining the Fault Density - Component Size Connection. *IEEE Software*, March/April 1997, pp.89-97.

[6] T.J. McCabe. A Complexity Measure. *IEEE Trans. on Software Engineering*, Vol2, 1976, pp.308-320.

[7] P. McCullagh and J.A. Nelder. Generalized Linear Models, 2nd Edition, Chapman and Hall, London, 1989.

[8] K-H. Moller and D.J. Paulish. An Empirical Investigation of Software Fault Distribution. *Proc. IEEE First Internation Software Metrics Symposium*, Baltimore, Md., May 21-22, 1993, pp.82-90.

[9] J.C. Munson and T.M. Khoshgoftaar. The Detection of Fault-Prone Programs. *IEEE Trans. on Software Engineering*, Vol18, No5, May 1992, pp.423-433.

[10] T. Ostrand and E.J. Weyuker. The Distribution of Faults in a Large Industrial Software System. *Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA2002)*, Rome, Italy, July 2002, pp.55-64.

[11] SAS Institute Inc. SAS/STAT User's Guide, Version 8, SAS Institute, Cary, NC, 1999.