Predicting the Location and Number of Faults in Large Software Systems

Thomas J. Ostrand, *Member*, *IEEE*, Elaine J. Weyuker, *Fellow*, *IEEE*, and Robert M. Bell

Abstract—Advance knowledge of which files in the next release of a large software system are most likely to contain the largest numbers of faults can be a very valuable asset. To accomplish this, a negative binomial regression model has been developed and used to predict the expected number of faults in each file of the next release of a system. The predictions are based on the code of the file in the current release, and fault and modification history of the file from previous releases. The model has been applied to two large industrial systems, one with a history of 17 consecutive quarterly releases over 4 years, and the other with nine releases over 2 years. The predictions were quite accurate: For each release of the two systems, the 20 percent of the files with the highest predicted number of faults contained between 71 percent and 92 percent of the faults that were actually detected, with the overall average being 83 percent. The same model was also used to predict which files of the first system were likely to have the highest fault densities (faults per KLOC). In this case, the 20 percent of the files with the highest predicted fault densities contained an average of 62 percent of the system's detected faults. However, the identified files contained a much smaller percentage of the code mass than the files selected to maximize the numbers of faults. The model was also used to make predictions from a much smaller input set that only contained fault data from integration testing and later. The prediction was again very accurate, identifying files that contained from 71 percent to 93 percent of the faults, with the average being 84 percent. Finally, a highly simplified version of the predictor selected files containing, on average, 73 percent and 74 percent of the faults for the two systems.

Index Terms—Software faults, fault-prone, prediction, regression model, empirical study, software testing.

1 Introduction and Previous Work

Software testing activities play a critical role in the production of dependable systems, and consume a significant amount of resources including time, money, and personnel. To help raise the effectiveness and efficiency of testing activities, we have developed a model to predict which files are likely to have the largest concentrations of faults in the next release of a system. The predictions allow testers to target their efforts on those files, enabling them to identify faults more quickly and providing additional time to test the remainder of the system. The net result should be systems that are of higher quality, containing fewer faults, and projects that stay more closely on schedule than would otherwise be possible.

Our statistical model assigns a predicted fault count to each file of a new software release, based on the structure of the file and its fault and change history in previous releases. The higher the predicted fault count, the more important it is to test the file early and carefully. The model can also be used to predict which files of a software system will have the highest *fault densities*, which we define in terms of faults per thousand lines of code (KLOC). Once these fault-prone files have been identified, testers can then apply any test case selection or generation algorithms normally used to particularly target the identified files.

We underscore that the goal of this work is fundamentally different from that of much other software testing research. While other testing research is aimed at devising and evaluating testing strategies, our research identifies files that will likely contain many faults and we therefore propose should be particularly scrutinized. Even work that investigates the need to allocate testing resources, such as *test prioritization* techniques [16] or regression testing, has different goals than ours. That research aims at sorting a given set of tests into an expected most-productive order of execution or proposes which tests of a previously selected test suite are most advantageous to rerun; our current work does not consider test case selection at all.

Our research provides indications of *where* it is likely to be beneficial to test, rather than *how* to select test cases. Therefore, the work presented here is intended to be used in conjunction with test case selection methods. For reasons explained in Sections 3 and 10, we and the testers who were doing the actual testing of the systems that we used in our case studies did not have any reliable severity data available to use to incorporate into our model or for test case selection. Faults can differ significantly in their impact on the reliability of a software system. Some have catastrophic consequences, while others are little more than cosmetic issues. Of course, it would be extremely helpful to know which of the identified faults were of high severity, or occur frequently, and which are of low consequence or occur rarely. But, for these real industrial projects and many others that we have encountered, that information is not available and so cannot be used as part of the model. Therefore, using predictions of files with the largest numbers of faults as the sole means of guiding testing is certainly not a prudent approach to testing a large system.

• *The authors are with AT&T Labs—Research, 180 Park Ave., Florham Park, NJ 07932. E-mail: {ostrand, weyuker, rbell}@research.att.com.*

Manuscript received 26 Oct. 2004; revised 6 Apr. 2005; accepted 12 Apr. 2005; published online 26 May 2005.
Recommended for acceptance by G. Rothermel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0246-1004.

However, we are merely providing this information as an extra tool for the tester to use in conjunction with whatever test case selection method they choose to use. This might include testing based on operational distributions that indicate the frequency of occurrence of elements of the input domain, or risk analysis-based tests which give particular weight to test cases that would have high consequences should they fail.

The basic unit of our analysis is files, in contrast to some other related work described in Section 9 that looked at collections of files or modules. We were able to work at the file level because the change tracking system from which we extracted our data provides information in terms of the individual files. A benefit of predicting fault-proneness at the file, rather than module, level is that a tester or developer who is trying to locate faults will have more precise information than would be provided by only module identification. For the systems that we studied, the average file size ranged between 250 and 300 commented lines of code.

Although we believe the application of our model could be useful during any phase of testing of new releases of a system, including unit, integration, system, and regression testing, our expectation is that it will be used primarily during system testing because that is such an expensive and resource-intensive phase of development. It might also be particularly valuable for regression testing if information is maintained indicating which test cases were designed to test specific files. The model could then be extremely helpful to regression testers by providing a means of deciding which test cases are most important to rerun.

In our initial work [13], we investigated which file characteristics were most closely associated with files having high fault densities. We based that work on 12 successive releases of an industrial inventory system that had been in the field in continuous use for about three years. However, discussions with software testers convinced us that it is often more valuable for testers to know which files had the largest numbers of faults rather than the largest fault densities since that would allow them to better identify most of the faults quickly. We therefore reconsidered a number of the characteristics in terms of absolute numbers of faults and built a model that is designed to predict which files will contain the largest numbers of faults in the next release.

However, one could reasonably argue that if the effort required for fault identification and removal is directly proportional to file size, then knowing which files will have the highest fault densities, rather than the highest fault counts, would also be valuable information for testers. For this reason, we also consider the use of our model to predict fault density and examine whether that prediction is useful and the advantages of each perspective. We will use the term *fault-prone* to mean that a file is likely to have either a large number of faults in the next release, or a high fault density, depending on which characteristic is being discussed.

For the work described in this paper, we followed the inventory system for five additional releases, and the overall prediction results are presented for 17 successive releases covering four years of field usage. We also collected data from nine releases, covering two years in the field, of a provisioning system that is used to initialize service for customers. For the inventory system, our data include faults identified at all stages of development, from the requirements phase through field release. For the provisioning system, faults were included beginning with integration testing. By examining such a large number of releases, we have been able to assess whether or not relevant characteristics change as the system matures and stabilizes, with new files comprising an increasingly smaller percentage of the system.

The predictions are done by a negative binomial regression model that predicts an expected number of faults for each file of a release, based on characteristics such as the file size, whether the file was new to the release, or changed or unchanged from the previous release, the age of the file, the number of faults in the previous release, and the programming language.

We first applied the model prospectively to predict the number of faults associated with each file in each of Releases 3 through 12 of the inventory system. For each release, we based the predictions on a model fit to data from prior releases only. When the files of each release were ranked in descending order of the predicted number of faults, the first 20 percent of the files contained from 71 percent to 85 percent of the release's faults, with the overall average being 80 percent. For the additional five releases, the model was even more successful at accurately identifying the files with the most faults. For these releases, the top 20 percent of the files contained between 82 percent and 92 percent of the faults, with the average for Releases 13-17 being 89 percent. Overall for all the releases the average was 83 percent.

For the provisioning system, the 20 percent of the files predicted to contain the largest numbers of faults also contained, on average, 83 percent of the faults. This system is discussed in Section 8.

Of course there is nothing magical about the 20 percent figure. When we looked at the graphs corresponding to many of the releases, we saw that the "knee" of the curve, i.e., the place where the curve flattened out, often occurred around the 20 percent mark. This can be seen in Fig. 1 in which we include curves that show the percentage of the files plotted against the percentage of faults included in the files, for each release from 3 through 17.

We are *not* suggesting that a project should test *only* the files that are highly ranked by the prediction model. What we are suggesting is that we are likely to get the greatest payoff by testing these files and, therefore, a good strategy is to test these files first and with greatest emphasis.

This paper makes several contributions. The first is the definition of the fault prediction model. The second contribution is the application of the model to the two different industrial software systems mentioned above. Using large industrial systems as the subjects of case studies lets us see the extent to which the model is likely to be usable in practice and having many releases lets us assess the accuracy of the model as the system matures and stabilizes. The third contribution evaluates the prediction

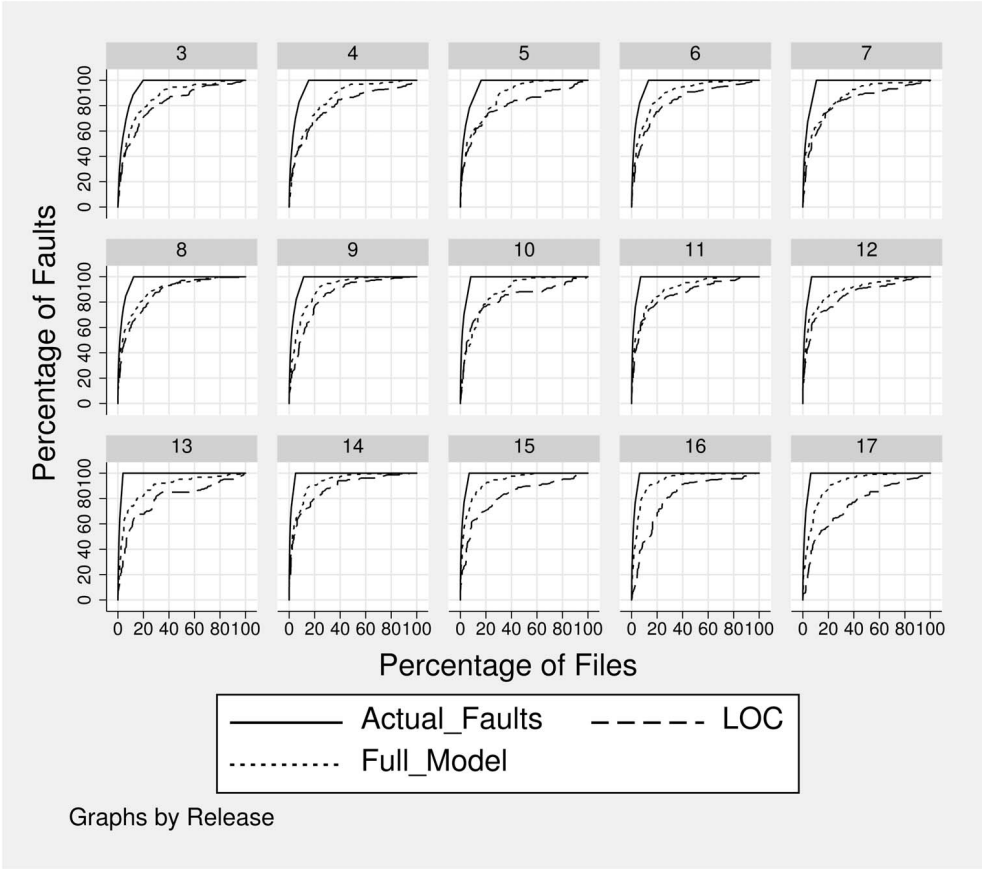


Fig. 1. Prospective predictions at Releases 3-17.

model when data are only available from testing stages later than unit testing. In many production environments, data collection for a system does not begin until after it leaves development, and moves on to integration or system testing. Since the model was developed using data from the inventory system that included faults detected at all stages of development, it was important to see whether the prediction based only on post-unit testing data would be comparably accurate.

The fourth contribution provides additional evidence of the prediction's effectiveness by applying the model to a different system, written by different personnel, using different primary programming languages, and performing different functions. In spite of these differences, a similar percentage of faults was identified by the model. The fifth contribution proposes a highly simplified prediction model and assesses its application to the two systems to which we applied the full model. We discuss the trade-offs between the decreased effort and expertise needed to apply the simplified model versus its decreased accuracy.

The remainder of the paper is organized as follows: Section 2 briefly describes the inventory system that is the primary subject of the case study. Section 3 describes the combined version control/change tracking system that was used by the projects to keep track of all requests to change the system for any reason, including faults. In Section 4, we describe the negative binomial regression model. Section 5.1 presents the results of applying the predictor to Releases 3 to 17 of the inventory system and compares its success at predicting which files have the highest numbers of faults for early releases with the results on the later, more mature and stable releases. Section 5.2 discusses a simplification of the predictor and provides information about how much predictive power is lost by this simplification. Section 6 considers the use of the model to predict the files that will have the highest fault densities and presents results for the inventory system. In Section 7, we compare the effectiveness of the predictor when applied to the fault data collected from all stages of development, including unit testing, to its effectiveness when restricted to faults detected only from integration testing and beyond for all 17 releases. Section 8 examines the model's applicability to the second system, to determine whether the key file characteristics for the inventory system are also significant for code written by a different group, for an entirely different purpose. In Section 9, we contrast our results with the work done by other research groups. This includes a detailed description of the prediction models and research done by three other groups. Section 10 discusses issues that might impede the widespread application of our model to production software projects, and proposes some solutions for dealing with those issues. Conclusions and plans for extending this work are discussed in Section 11.

2 The Inventory System

While our earlier studies [13], [14] used the first 12 releases of the inventory system, there are now a total of

TABLE 1
Inventory System Information

Rel	No of Files	Lines of Code	Mean LOC	Dev Unit Faults	PUT Faults	Total Faults	Fault Density	Files With Any Faults	Pct With Any Faults
1	584	145,967	250	768	220	988	6.78	233	39.9
2	567	154,381	272	172	28	200	1.30	88	15.5
3	706	190,596	270	400	85	485	2.56	140	19.8
4	743	203,233	274	292	35	327	1.61	114	15.3
5	804	231,968	289	281	56	337	1.47	131	16.3
6	867	253,870	293	288	51	339	1.34	115	13.3
7	993	291,719	294	170	37	207	0.71	106	10.7
8	1197	338,774	283	375	113	488	1.45	148	12.4
9	1321	377,198	286	346	88	434	1.16	151	11.4
10	1372	396,209	289	202	43	245	0.62	112	8.2
11	1607	426,878	266	174	106	280	0.66	114	7.1
12	1740	476,215	274	192	81	273	0.57	120	6.9
13	1772	460,437	260	88	39	127	0.28	71	4.0
14	1877	482,435	257	164	71	235	0.49	95	5.1
15	1728	479,818	278	251	54	305	0.64	120	6.9
16	1847	510,561	276	181	93	274	0.54	116	6.3
17	1950	538,487	276	188	65	253	0.47	122	6.3

17 successive releases which are the subject of the current study. Table 1 shows basic facts about the system's files and the detected faults over the 17 releases. The files we included in our study were written in languages that are normally produced and maintained by programmers (i.e., source files), and that are executable. They were selected on the basis of their extension. Thus, for example, .java, .xml, and .mk files are included. In contrast, .jpg files are not included because they are not produced by a programmer, and .doc files are not included because they are not executable. We count faults that occur in the included files.

The column in Table 1 labeled "Dev Unit Faults" contains a count of all faults identified during development and unit testing, the earliest stages of the system. The column labeled "PUT Faults" includes all faults identified during post unit testing, including integration testing, system testing, beta trial, controlled release and general release. In general, there were very few faults found from beta trial and later; for the 17 releases studied, barely 3 percent of the faults were found during any of these three stages. In contrast, 15.5 percent of the faults were identified during system test and almost 70 percent were found during unit testing.

Not surprisingly, there is a general increase in the size of the system as new functionality is added at most releases, particularly the earlier ones. From Release 1 to Release 12, the system roughly tripled in size from 584 files containing almost 146,000 lines of code to 1,740 files, containing a total of more than 476,000 lines of code. Although the system continued to grow steadily over Releases 13 to 17, the rate of growth definitely began to slow. This is not surprising, as new files frequently represent new features, while modified files often represent fault repairs, and as the system matures one would expect fewer new features to be added. The average number of lines of code per file remained fairly constant for all releases, while with a few exceptions, the fault density tended to decrease as the system matured.

The next to last column in the table shows the number of files that contained at least one fault that was detected at any stage of development, while the final column shows the percentage of files that were found to contain at least one fault in the release. At each release after the first, faults occurred in fewer than 20 percent of the files, with those files typically averaging two to three faults apiece. This concentration of faults suggested that testing effort could be reduced greatly if most of the faulty files could be identified prospectively. Of course it is possible, and even likely, that other files in the system contain faults that have gone undetected. However, the low number of field faults identified during the four year period that the inventory system was tracked indicates that undetected faults was not likely a significant issue.

The system is primarily written in Java (approximately 77 percent of the files); other languages used include shell script, make, xml, html, perl, c, sql, awk, plus a number of specialized languages.

3 The Fault Database

Many AT&T software projects use a combined version control/change tracking system throughout their life cycles. The basic entity of this system is a *modification request* (MR) that is entered by a developer or tester when a change is deemed necessary. The change could involve either the software or the documentation of the project. An MR contains a written description of the reason for the proposed change and a severity rating of 1 through 4 that characterizes the importance of the proposed change. If the request results in an actual change, the version control system automatically records into the MR the file(s) that are changed or added, the specific lines of code that are added, deleted, or modified, and a timestamp of the change.

Problems in the software, which are manifested by failures such as incorrect output or crashes when the system

is run, are caused by faults in the code. MRs are the basis for counting faults. The problem described in an MR may be repaired by making one or more changes to one or more files. We count one fault for each file affected by an MR, regardless of the number of lines in the file that are actually modified, added, or deleted. Of course, a given file can require changes to fix more than one problem, each of which would be described in a separate MR. If k different MR's in a release describe changes to a given file, then that represents k distinct faults in that file. This fault counting convention is consistent with the ones used in [10] and [3], and in our earlier studies [13], [14].

The template for creating an MR provides a great many fields that must or can be completed manually by the MR creator. For our study, the most important of these are the Release ID, the MR category, the Abstract and Description, and the Bug Identification field described below. The Release ID identifies the release in which the modification is being requested. For MRs that represent faults, this is usually the release that was running when the problem was observed. The MR category field identifies how the problem was first identified. Possible values for this field include "developer_found," "system_test_found," and "customer_found," among others. The Abstract field gives a brief summary of the MR, and the Description provides space for detailed information. The system provides a default of "developer_found" for the MR category, while the Release ID, Abstract, Description, and Bug Identification fields are initially unfilled.

Although we had hoped to use the MRs' severity ratings to identify faults of critical importance, we learned that these ratings are highly subjective and also sometimes inaccurate because of political considerations not related to the importance of the change to be made. We also learned that they could be inaccurate in inconsistent ways. For example, a change could be rated as a Severity 2 modification, when in fact the person writing the MR actually judged it as either a Severity 1 or Severity 3 fault. It might be downplayed so that friends or colleagues in the development organization "looked better," provided they agreed to fix it with the speed and effort normally reserved for Severity 1 faults, or it might be "upgraded" so that developers who actually make the change focus on it quickly.

We also discovered that relatively few faults were listed as high severity faults. For the first 13 releases of the inventory system, more than 81 percent of the faults were ranked as Severity 3, a rating intended for faults that do not have severe consequences and can be worked around until they have been fixed, while little more than 1 percent of the faults were listed as being Severity 1, the most critical category of faults. These are faults that need to be corrected immediately so that the system can resume proper operation. For these reasons, we did not include fault severity in any way to influence or weight test case selection.

The inventory system used in the earlier case studies contained changes found at each of nine distinct stages of development: requirements, design, development, unit test, integration test, system test, beta release, controlled release, and general release. Faults were reported for each of these stages and, surprisingly, more than two-thirds of the faults (3,987 out of 5,797), were reported to have been identified during unit testing. When the other pre unit testing phases were added, nearly 80 percent of the faults reported in the database were found during these very early stages. Each fault is associated with the release that is active at the time of submitting the MR that reports the fault, which is not necessarily the same as the release when the fault was created. Faults discovered during unit or integration testing are most likely to be reported during the same release that created them, while system testing and field (i.e., beta, controlled, and general release) faults might be discovered only several releases later. We did not attempt to identify the release where faults originally entered into the code. In many cases that would be impossible to deduce since the code may have become problematic as a side effect of earlier changes or might have been faulty when initially written.

One of the problems we had to address at the outset of our studies was how to identify faults. For both systems, there was generally no identification in the MR database of whether a change was initiated because of a fault, an enhancement, or some other reason such as a change in the specifications. The inventory system recorded roughly 10,000 MRs over the 17 releases, far too many to read all of the reports and make a determination of exactly which were faults and which were not. The project's test team suggested that if only one or two files were changed by an MR, then it was likely a fault, while if more than two files were affected, it was likely not a fault. The rationale was that if many files were touched by the change, it was generally a change in the interface caused by a change in the specifications. For the inventory system we used that rule of thumb to identify fault-caused changes.

In order to determine whether or not this was a reasonable approximation, we did a small, informal study in which we selected roughly 50 MRs and read them carefully. To be sure that we were not likely to be missing a significant number of faults by using this approximation, we deliberately included at least 10 MRs that changed three or more files, and several that changed more than 20. Every MR we examined that caused changes in more than two files was not a fault correction. These many-file MRs were written for a variety of reasons, but had in common that the change was uniform across all the files affected by the MR. One typical MR of this type changed the method of reporting exceptions in 28 different files. In the old versions of these files, an error message was printed; in their new versions, the message was printed, and in addition, error information was logged. Another MR was used to redefine and shorten certain function names in a set of 16 different files. A third example is an MR that modified six files by replacing in each a function call with a call to a newly defined function that implements a different interface to a database.

Of course, by using this approximation, there will be some faults that are not identified as such, and there will be nonfaults that are incorrectly identified as faults. However, since the informal examination of MRs indicated that the rule of thumb almost always worked and the rule was used uniformly, we do not believe that it introduced any particular bias.

In Section 8, we apply our prediction model to a second system. In this case, substantially fewer changes were included in the database since data was not entered until system test began. We were therefore able to read every MR and determine whether or not the change was due to a fault. For most entries, it was apparent from the write-up whether or not the change was a fault. For a few entries, the description was unclear, and we asked the test team to make the determination.

4 Multivariate Analysis to Predict the Number of Faults in Each File

We now describe the negative binomial regression models we designed to predict the number of faults in a file during a release. Similar to multiple linear regression, the negative binomial regression model relates the outcome to a linear combination of file characteristics. Unlike linear regression, this model accounts for special circumstances involved in modeling an outcome that is a count.

Modeling serves two primary purposes:

- It provides information regarding the association between the number of faults and individual file characteristics while holding other file characteristics constant. That is, regression coefficients estimate the unique contribution of individual characteristics.
- The model makes predictions of which files will contain the largest numbers of faults in a release, allowing testing resources to be targeted more effectively.

The second of these objectives is the primary goal of the research described in this paper. We briefly outline the model in Section 4.1, while in Section 4.2, we describe the specific explanatory variables used. Section 4.3 presents results for Releases 1 to 12, which were the initial predictions we made, and lists other variables that we found did not enhance the predictive abilities of the model.

4.1 The Negative Binomial Regression Model

Negative binomial regression extends linear regression in order to handle outcomes like the number of faults [9]. It explicitly models outcomes that are nonnegative integers. For such outcomes, it is unrealistic to assume that the expected value of the outcome is an additive function of the explanatory variables. Instead, it is assumed that the expected number of faults varies as a function of file characteristics in a multiplicative relationship.

Let y_i equal the number of faults observed in file i and x_i be a vector of characteristics for that file. The negative binomial regression model specifies that y_i, given x_i, has a Poisson distribution with mean λ_i. Unlike the related Poisson regression model, the negative binomial model allows for the type of concentration of faults we observed in the inventory system by explicitly incorporating a source of additional dispersion (variation in the number of faults) for each file. Specifically, the conditional mean of y_i is given by $\lambda_i = \gamma_i e^{\beta' x_i}$, where β is a column vector[1] of regression coefficients conforming to the x_i, and 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 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.

The regression coefficients β and the dispersion parameter σ^2 are estimated by maximum likelihood, as is standard [9]. Standard errors for the estimated regression coefficients are adjusted to account for the additional uncertainty associated with the unobserved γ_i.

4.2 Explanatory Variables

We present the results of a negative binomial regression model fit to files from Releases 1 to 12. The unit of analysis is a file-release combination, yielding a total of 12,501 observations. The outcome is the number of faults associated with the file at the given release. The procedure Genmod in SAS/STAT Release 8.01 [17] was used to fit all models.

Predictor variables for the model included: the logarithm of the number of lines of code; whether the file was new, changed or unchanged, which we refer to as the file's *change status*; the file's age as measured by the number of previous releases in which it appeared; the square root of the number of faults identified during the previous release; the programming language used; and the release number.

The log of the number of lines of code, file age, and the square root of the number of prior faults were treated as continuous variables. We included lines of code in terms of its logarithm because initial analysis suggested that the expected number of faults for a file was roughly proportional to the number of lines of code (LOC). Since the logarithm of the expected number of faults is modeled as a linear combination of the explanatory variables, our initial findings would imply that code mass should be specified as log(LOC), with an anticipated coefficient near 1.0.

Because the number of faults at the previous release had a very long tail (the maximum value was 57), we tried a variety of transformations to reduce the influence of extreme values. The square root transformation produced a better fit to the data, in terms of the log likelihood of the model, than either no transformation or more extreme transformations.

Change status, programming language type, and release number were treated as categorical variables, each fit by a series of dummy (0-1) variables, with one omitted category that served as the reference. For change status, the reference category was unchanged files. In this way, the new and changed coefficients represented contrasts with existing, unchanged files. For program type, the reference category was java files, the most commonly occurring type for this system. We arbitrarily set Release 12 as the reference release. Each factor included in the model was easily statistically significant at the 0.001 level.

1. β' is the transpose of β.

TABLE 2
Results from Full Negative Binomial Regression Model

Predictor	Coef	Standard Error	t	95 Percent Conf Interval
Intercept	-1.636	.149	-10.95	(-1.929, -1.343)
log(KLOC)	1.047	.031	34.32	(.987, 1.107)
Sqrt Prior Faults	.425	.041	10.47	(.346, .505)
Age	-.050	.014	-3.60	(-.077, -.023)
File Status				
New	1.861	.114	16.33	(1.638, 2.085)
Changed	1.066	.087	12.21	(.894, 1.237)
Unchanged	.000	NA	NA	NA
Program Type				
makefile	2.548	.149	17.09	(2.256, 2.840)
sql	1.747	.217	8.06	(1.322, 2.172)
Other	1.714	.231	7.41	(1.261, 2.168)
sh	1.011	.147	6.87	(.723, 1.300)
html	.576	.177	3.26	(.230, .923)
xml	.439	.324	1.35	(-.196, 1.074)
perl	.168	.189	.89	(-.202, .539)
java	.000	NA	NA	NA
c	-1.082	.327	-3.30	(-1.723, -.440)
Release				
1	1.453	.139	10.48	(1.181, 1.724)
2	.141	.175	.81	(-.202, .485)
3	1.115	.143	7.79	(.835, 1.396)
4	.803	.151	5.31	(.507, 1.100)
5	.671	.149	4.51	(.380, .963)
6	.654	.147	4.46	(.367, .941)
7	.087	.153	.57	(-.213, .387)
8	.613	.130	4.73	(.359, .867)
9	.550	.132	4.17	(.292, .809)
10	.385	.141	2.73	(.109, .660)
11	.211	.135	1.56	(-.054, .476)
12	.000	NA	NA	NA

4.3 Results for Releases 1 to 12

Table 2 shows estimated coefficients, their standard errors, t-statistics associated with the null hypothesis that the true coefficient equals zero, and 95 percent confidence intervals for the true coefficients.

For continuous predictor variables, positive coefficients indicate characteristics that are positively associated with the number of faults, while controlling for other characteristics. For example, the model estimates that a unit change in the square root of the number of faults in the prior release (e.g., a change from 0 to 1 fault or from 1 to 4 faults) is associated with an increase of 0.425 in the logarithm of the expected number of faults. This translates into a multiplicative factor of $e^{0.425} = 1.53$. In contrast, the model estimates that the expected number of faults in a file decreases by 5 percent ($e^{-0.050} = 0.95$) for each additional release it has existed, holding all else equal (in particular, the file's change status and number of prior faults).

We found that the number of lines of code was the strongest individual predictor in the model, provided the goal was to identify files likely to contain the largest numbers of faults. The estimated coefficient for the logarithm of lines of code was 1.047. Because the 95 percent confidence interval included 1.00, this result is consistent with a finding that after controlling for all other factors, the number of faults is proportional to the number of lines of code—in other words, that fault density does not change with LOC.

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, the coefficient of 1.066 for changed files indicates that changed files have about $e^{1.066} = 2.90$ times more faults than existing, unchanged files with otherwise similar characteristics. Of course, the changed files are also more likely to have other characteristics such as prior faults that indicate a propensity for faults at the current release, so that the average number of faults per changed file is far greater than 2.90 times that for unchanged files. The estimated coefficient for the new file variable was 1.861. This implies that new files had about $e^{1.861} = 6.4$ times as many faults as existing, unchanged files that were similar in all other respects.

Makefiles, sql, shell, html, and "other" files all had significantly higher fault rates than java files when holding all else constant, while c files had significantly lower rates.

TABLE 3
Percentage of Faults Included in 20 Percent of the Files Using the Full Model

Release	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	Avg
% Faults Identified	77	74	71	85	77	81	85	78	84	84	82	91	92	92	88	83

We do not interpret these findings as evidence that one language is inherently more, or less, fault-prone than any other, because the various languages may be providing very different functionality. In addition, the same functionality may require many more lines of code in one language than in another, or may simply be very poorly suited for a task. Nonetheless, it is important to include programming language when predicting numbers of faults.

As shown in Table 1, fault density declines by an order of magnitude, though not monotonically, from Release 1 to Release 12. Much of that decline is due to decreasing proportions of new or changed files, the increased age of most files, and fewer faults at the prior release for existing files at later releases. After controlling for those other characteristics, the coefficients for release number show a gradual decline in fault rates. We note that the coefficients for release do not directly affect the rank ordering of predictions for files within a single future release.

The full model, which combined these and the other factors displayed in Table 2, produced more accurate predictions than models based on any of the individual factors (see Section 5.2 for additional details).

In Section 5, we will apply the model to an additional five releases. The results for these additional releases provide a completely independent validation of the model's predictive accuracy for this system. The form of the model and the variables used were selected before looking at the data for Releases 13-17. In addition, the estimated coefficients used to order files in Releases 13 through 17 were those resulting from fitting the model to data from Releases 1 through 12. Consequently, all aspects of the model were independent of the five additional releases.

Other possible predictor variables were tried, but not included in the model, as they did little to improve the predictive power as measured by log likelihood. Among the variables that we decided not to include were the number of changes made to a file for those files that were changed since the previous release, whether a file had been changed prior to the previous release, and the logarithm of the cyclomatic number for java files. Since the cyclomatic number [8] has been found to be very highly correlated with the number of lines of code, it is not surprising that this did not enhance the predictive power of the model.

5 Predictions for Releases 3 through 17 of the Inventory System

5.1 Prediction Results

The negative binomial regression model assigns a predicted fault count to each file of a release. In each case, the predictions use data from Releases 1 through (n-1) to make the predictions for Release n. Sorting the files in descending order of their predicted fault counts creates an ordered list of the files from most to least likely to be problematic. Although the individual fault counts predicted for each file generally do not exactly match their actual fault counts, the great majority of the actual faults occur in the set of files at the top of the listing. Table 3 shows the percentage of actual faults that occurred in the first 20 percent of files predicted by the model for each release. The results for Releases 3 through 12 were described in [14]. For these releases, the model's first 20 percent of files contained between 71 percent and 85 percent of the faults actually detected in the system, with an average of 80 percent over all releases through Release 12.

The accuracy of these predictions was very encouraging, but raised the question of whether the accuracy would diminish for later releases, as the system stabilized and matured. This is particularly important since, as can be seen in Table 1, by Release 10, all of the identified faults are concentrated in less than 10 percent of the files, making the potential payoff of accurate prediction extremely valuable.

We see by comparing the results for Releases 13 through 17 to those for the earlier releases that the accuracy of the prediction actually improved as the system matured. The average percentage of faults contained in the 20 percent of the files selected by the model was 89 percent for the five most recent releases, bringing the overall average for all of the releases through Release 17 to almost 83 percent.

The accuracy of the overall fault predictions can be evaluated by comparing the predicted ordering to the ordering according to the actual number of faults discovered. The graphs of Fig. 1 show this comparison, both for the full prediction model based on all the significant variables, and for a simplified model that is based only on the lines of code (LOC) in each file. This simplified LOC model and our assessment of its predictive ability will be discussed in Section 5.2 for both the inventory system and the provisioning system described in Section 8.

For each release, the curves plot the cumulative percentage of faults (on the vertical axis) found in a given percentage of the files (on the horizontal axis). On the Actual Faults curve represented by the heavy solid line, the files are sorted in decreasing order of the number of actual faults found in each file. This is the optimal ordering given the goal of finding all the faults in as few files as possible; it represents perfect prediction. On the Full Model curve represented by the short dashed line segments, the files are sorted according to the full model's prediction of the number of faults contained in each file. The "quality" of the model predictions can be measured in terms of how close the prediction curve comes to the Actual Faults curve. The LOC curve, represented by the long dashed line segments,

TABLE 4
Percentage of Faults Included in the 20 Percent of the Files Selected by LOC

Release	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	Avg
% Flts	71	68	71	75	76	76	77	77	80	75	68	80	71	67	58	73

sorts the files solely in decreasing order of their size. This model will be discussed in the next section.

5.2 Simplifying the Model

We have now provided evidence that our proposed model is useful for predicting which files are likely to contain the highest numbers of faults. However, it does require relatively sophisticated use of statistical modeling and, so, the next question we consider is whether the model can be significantly simplified so that a development team can make similar sorts of predictions without needing to rely on someone with a strong background in statistics.

Since file size was by far the most potent predictor of the number of faults in a file, we considered using file size alone to predict which files would contain the largest numbers of faults. Table 4 shows the percentage of faults contained in the 20 percent of the files that were the largest in terms of the number of lines of code for the inventory system. While the average percentage of faults contained in the 20 percent of the files selected by the full model over the 17 releases is 83 percent, for the simplified model that value is 73 percent.

Looking again at Fig. 1, we see that while the LOC model approaches the full model in some of the early releases (especially Releases 6-8), in the later more mature releases, its performance is significantly poorer. In particular, note that for the last four releases, the LOC model requires nearly 100 percent of the files to capture all the release's faults, while the full model reaches 100 percent of the faults with 40 to 50 percent of the files. This reflects the fact that some very short files contain faults, and characteristics other than size play a role and, therefore, raise their ranking in the full model.

These results provide evidence of a trade-off between the model's complexity and the expertise required to apply the model, versus the model's accuracy. For testers, this implies a certain utility to each model. The simplified model provides a "quick and dirty" ranking of the files that is probably most useful at early stages of system testing, when there is no lack of faults to be found and removed.

The added complexity of the full model seems capable of narrowing down the location of nearly all the faults to far fewer files than the simplified model. This occurs because even though size is the most important factor in the full model, several other factors also affect the ordering of files. The full model might assign a high fault prediction to small or medium-sized files that have many faults because, for example, they are new or changed in the previous release. In the simplified model, however, those files will not place high in the ranking and, hence, not be singled out for particular scrutiny.

Fig. 2 compares the effectiveness of the full model with that of the LOC model when different percentages of files are selected. The files selected by the full model consistently contain between 10 and 13 percentage points more faults than the LOC model for the entire range from 5 percent through 30 percent of the files. For the 35 percent to 50 percent of the files range, the difference is still seven or eight percentage points. It is difficult to imagine using the model to recommend such a large percentage of the files, however, especially given that, for as few as 20 percent of the files, the average percentage of faults included in the files selected by the full model is

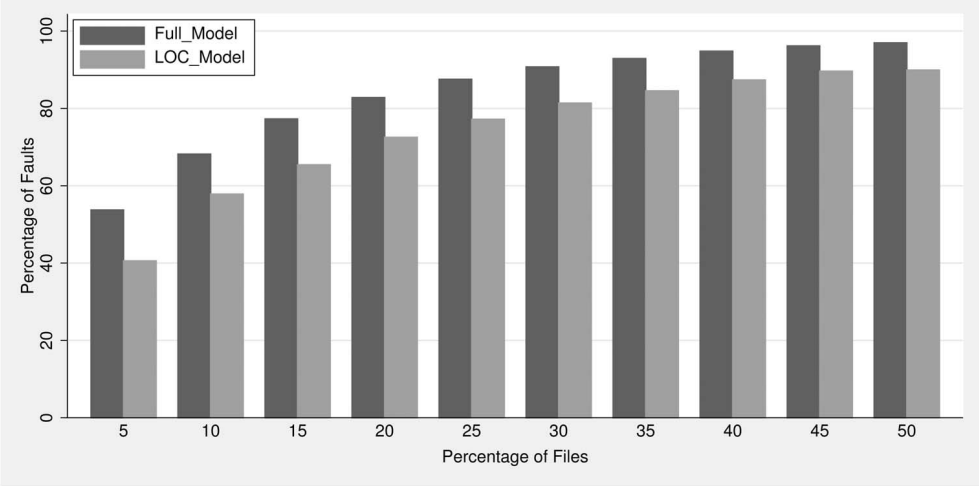


Fig. 2. Average percentage of faults by percentage of files included.

TABLE 5
Percentage of Faults Found in 20 Percent of Files Using the Full Model to Predict Fault Density or Number of Faults

20% of Files with the Highest Predicted Fault Densities																
Release	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	Avg
% of Faults	49	63	55	55	46	38	71	54	36	60	86	62	89	88	84	62
% of LOC	25	37	28	28	17	11	29	29	12	26	33	26	26	23	23	25
20% of Files with the Highest Predicted Numbers of Faults																
Release	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	Avg
% of Faults	77	74	71	85	77	81	85	78	84	84	82	91	92	92	88	83
% of LOC	61	60	59	59	56	64	59	59	57	60	63	61	59	58	57	59

83 percent. We would therefore expect to see differences above 10 percent in practice as testers select small percentages of files to focus on.

6 Predicting Fault Density

We have provided evidence that our model is quite accurate at identifying those files that will contain the largest numbers of faults. Although we believe that is the information that will be most useful to testing practitioners, we still wondered how the model would fare at predicting which files were likely to contain the largest numbers of faults relative to their size. As argued in Section 1, if the cost to identify and remove faults is proportional to the size of the file, then knowing which files have high fault densities might also be very valuable.

Instead of sorting the files in decreasing order of the predicted number of faults per file, we can use the model's predicted number of faults and the size of each file to compute a predicted fault density, and sort the files based on that result.

The top section of Table 5 shows the percent of faults found in the top 20 percent of the files ordered by predicted fault density, while the bottom section shows the results using predicted fault counts, for comparison.

Not surprisingly, sorting files by predicted fault density is not as effective at finding large numbers of faults, although it always correctly identifies a far larger percentage of the faults than would be expected in a random selection of 20 percent of the files. On average, the top 20 percent of files in the fault density ordering contained 62 percent of the faults, compared to 83 percent in the fault count ordering. For Releases 13-17, the fault density ordering's average of 82 percent was significantly higher than the average for the earlier releases, and only somewhat lower than the average of 89 percent for Releases 13-17 in the fault count ordering. It is also interesting to compare the percentage of the lines of code included in the identified files under the two different orderings. Because lines of code was determined to be the most important characteristic associated with high-fault files even though other characteristics do play a role, the model generally predicts a large number of faults for large files. The top of the fault count ordering therefore contains many large files. In the fault density ordering, however, it is possible for very short files containing a small number of predicted faults to have a high fault density and occur in the top part of the list. As a result, the top 20 percent of the fault density ordering generally contains much less code than the top 20 percent of the fault count ordering. For the inventory system, the top 20 percent of files ordered by fault count contained, on average, 59 percent of the lines of code and 83 percent of the faults. In the fault density ordering, the top 20 percent of the files contained, on average, only 25 percent of the lines of code and 62 percent of the faults. Therefore, if a fault isolation method is used that is proportional to the size of the file, then the fault density ordering might be particularly advantageous.

This prediction data might be considered one other way. Suppose instead of selecting the 20 percent (or any other percentage) of the files expected to contain the largest numbers of faults or the highest fault densities, we instead fix the percentage of the lines of code to include and select the files in the initial segment of each fault ordering that contain that amount of code. Table 6 illustrates this approach using 50 percent of the lines of code. The entries show the percent of faults found and the percent of the files selected to include 50 percent of the LOC when files are sorted in order of fault count and fault density. Using the fault count ordering, these files contain, on average, 76 percent of the faults, and only 14 percent of the files. In contrast, the fault density ordering does better at identifying the files with large numbers of faults, averaging 85 percent of the faults, but selects a significantly larger number of files, in this case about 39 percent.

Therefore, if it is believed that testing effort is proportional to the amount of code tested, then identifying the high fault density files is more effective than identifying the high fault count files. But, since system testers generally

TABLE 6
Percentage of Faults Found in Files Containing 50 Percent of LOC Using Model to Predict Fault Density or Number of Faults

Files Constituting 50% of LOC, with the Highest Predicted Fault Densities																
Release	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	Avg
% of Faults	79	75	80	80	79	82	90	75	81	89	92	88	95	99	98	85
% of Files	37	30	37	30	36	50	36	32	41	34	38	45	42	43	47	39
Files Constituting 50% of LOC, with the Highest Predicted Numbers of Faults																
Release	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	Avg
% of Faults	69	62	63	72	68	72	76	73	79	78	78	89	87	91	85	76
% of Files	13	14	14	13	15	12	14	16	15	14	13	14	15	15	17	14

base tests on functionality rather than code details, we hypothesize that this relationship does not generally hold for system testing. However, it might be useful to identify those files that are most likely to be problematic from both points of view to see whether they contain particularly high numbers of actual faults. We plan to investigate this question.

We therefore see that there can be some value in using the current model to identify files that are likely to have high fault densities instead of only considering files with high fault counts. If the percentage of files is fixed, then the model does a better job at correctly predicting the files with a large number of faults at a cost of including a much larger percentage of the lines of code, as compared to using the model to predict fault density. On the other hand, if the percentage of lines of code is fixed, then using the model to predict the fault density identifies a larger percentage of the faults at a cost of requiring the inclusion of a larger percentage of the files.

7 Prediction for Faults Found in Integration Testing and Beyond

The success of the model in making accurate predictions for the first 17 releases of the inventory system encourages us to believe that the approach is potentially applicable to many production software systems. However, it is unusual for software projects to maintain records of faults detected prior to integration or system testing; in fact, out of more than 20 systems the authors have examined during this and other case studies, only the inventory system of the current study classifies the "phase identified" of a large proportion of the faults as unit testing.

If early faults are not included in the change management database for a system, then it will have a different population of faults, as well as having significantly less data. We are interested in determining whether the predictions change substantially if the fault universe is restricted to post unit testing faults and whether the model is robust with substantially smaller amounts of data. Out of 5,797 faults detected over all life-cycle phases of the 17 releases, 1,265, or 22 percent, were recorded as detected during integration testing or later.

We applied the negative binomial regression model to the inventory system using this reduced universe of faults, both as the dependent variable and as the measure of faults in the prior release. Table 7 shows the percent of post unit test (PUT) faults that were in the top 20 percent of files predicted by the model. To facilitate comparison, the upper row of the table shows the results obtained using the entire set of faults.

With an overall average of 84 percent versus 83 percent and a range of 71-93 percent versus 71-92 percent, the results are nearly identical, showing that the model appears to work equally well for a system for which significantly less data is available, and that does not include faults identified during unit testing or earlier. This encourages us to believe that the prediction approach will be applicable to

TABLE 7
Percentage of Post Unit Test Faults Included in 20 Percent of the Files

Release	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	Avg
% Flts - ALL	77	74	71	85	77	81	85	78	84	84	82	91	92	92	88	83
% Flts - PUT	80	71	73	90	81	83	81	81	83	90	90	93	85	88	86	84

TABLE 8
Provisioning System Information

Aggregated Rel Name	Rel No.	No. of Files	Lines of Code	Mean LOC	Faults Detected	Fault Density	Files With Any Faults	Pct With Any Faults
A	1	2008	381,973	190	24	0.06	19	0.9
B	2	2085	397,683	191	85	0.21	63	3.0
	3	2137	412,621	193	52	0.13	41	1.9
	4	2104	406,674	193	10	0.02	9	0.4
	5	2119	407,724	192	6	0.01	6	0.3
C	6	2213	423,895	192	15	0.04	14	0.6
	7	2250	434,772	193	74	0.17	64	2.8
	8	2230	434,781	195	34	0.08	30	1.3
	9	2241	437,578	195	7	0.02	6	0.3

far more projects than if fault data from all development phases were always required.

8 Applicability of the Model to Other Systems

To judge the potential of the model for helping a wide variety of software projects to order their testing efforts, we studied a second software system with a substantial number of releases and years of field usage. This system, developed by different personnel, using different design and development paradigms, is a *provisioning system*, used to establish or activate service for a customer by configuring hardware and software.

We collected data for nine releases of this system, spanning two years of field experience. While the inventory system is primarily written in java, the most-used language for the provisioning system is sql, accounting for roughly one quarter of the files. The next most common languages are java and shell script, which together account for another quarter of the files.

The two systems are similar in size. Release 9 of the provisioning system has 2,241 files containing a total of 437.6 KLOCs. Table 8 provides more details about the system. However, while the inventory system grew substantially over its lifetime, with many new files and features being added at each release, the provisioning system was already fairly mature at the point when our data collection began. While the number of files in the inventory system grew by 233 percent over 17 releases, the provisioning system files increased by only 11 percent during the two year period that was studied. Another difference between the systems is that systematic fault reporting for the provisioning system did not begin until unit testing was complete and integration testing began.

For this system, only 307 faults were identified over the nine releases, substantially fewer than were observed in the inventory system, even when only post unit testing faults were considered as in Section 7. The combination of relatively few new files and small number of faults per release conforms to our observation in [13] that files that existed in earlier releases generally have substantially fewer faults and lower fault densities than new files.

Because there were so few faults associated with this system, with four of the releases having 15 or fewer faults identified, there was insufficient data to apply the negative binomial regression model separately for the nine releases.

To see whether we could actually use the model to make predictions for this system in spite of the paucity of data, we treated the period from Release 2 through 5 as a single release, and the period from Release 6 through 9 as a single release. This gave us three releases with the first release (Release A) having 24 faults, the second release (Release B) having a total of 153 faults, and the third release (Release C) having a total of 130 faults.

Since Release A was our starting point, its change status and file fault history were unknown, and hence no attempt was made to model its faults. Data from Release A were used to determine values of the change status and fault history for the files in Release B. Data from Release B were then used to develop predictions of the number of faults for Release C. The prediction model was a slight revision of the one developed for the inventory system presented in Section 5.1. Because data for only Release B were used, the model excluded release number and age of the files, which were not known.

As with the inventory system, lines of code and whether a file was new, changed, or unchanged were strong, statistically significant factors for predicting faults in the provisioning system. The estimated coefficient for the logarithm of LOC was 0.73, significantly lower than 1.00, suggesting that in this system, fault density may decrease for longer files, holding all else equal. The estimated coefficient for new files was 1.92, very similar to the value for the other system (1.86). The estimated coefficient for changed files was 1.77, compared with 1.06. Programming language was statistically significant overall, but not strikingly so. That may simply be due to the large drop in the amount of information (few faults) in the new data. The number of prior faults was not significant, perhaps due to very few faults in Release A, but was retained for purposes of making predictions at Release C.

Using the model estimated with data from Release B to predict which files would likely contain the most faults in Release C, the percentage of faults contained in the 20 percent of the files selected by the model was 83 percent, identical to the overall predictive accuracy for the 17 releases of the inventory system.

We also examined the LOC model for Release C. As was the case for the inventory system, although the LOC model did not predict which files would contain the largest numbers of faults as accurately as the full model, there was still value in using the size-alone predictor. For this system, the full model's top 20 percent of files contain 83 percent of the faults, while the LOC model's top 20 percent of files contain close to 74 percent of the faults.

9 Comparison with Other Prediction Approaches

Several other research groups have done studies to identify characteristics associated with software entities that are particularly fault-prone. Results along these lines can be found in [1], [2], [3], [4], [5], [6], [7], [10], [11], [12], [15], among others. Research with goals similar to the research described in this paper appears in [7], [4], and [12]. These groups attempt to make predictions relating faults to specific parts of the software system.

Khoshgoftaar et al. [7] used discriminant analysis to develop two classification models that predict whether or not a module will be fault-prone, rather than attempting to predict the actual number of faults that will occur in the module. This study defines a module to be a collection of files and says that a module is fault-prone if it contains five or more faults. The models use data from a single release of a large telecommunications system, containing about 1.3 million LOC, with approximately 2,000 modules and 25,000 files. For the entire sample set of the study, approximately 12 percent of the modules were, by their definition, fault-prone.

The predictions are based on software design metrics and reuse information. The design metrics include call graph metrics such as the number of calls from one module to other modules and the number of distinct modules that are referenced, as well as control-flow graph metrics such as the number of loops, cyclomatic complexity, and nesting level. Reuse information classifies whether a module was new in the current release and, if new, whether it was changed from the previous release.

Model 1 was based solely on the software design metrics, while Model 2 used both the design metrics and reuse information. The discriminant analysis was performed on a randomly chosen set of two thirds of the system's modules. The two models were evaluated according to their ability to correctly predict the fault-proneness of the system modules.

The authors point out that from a testing point of view, it is most important to avoid predicting that a module that is actually fault-prone is not fault-prone, a so-called Type 2 misclassification. To evaluate the efficacy of the models, they were applied to the remaining one third of the system's modules. Model 1 predicted 21.3 percent of the actual fault-prone modules to be not fault-prone, while Model 2 predicted 13.8 percent of the fault-prone modules to be not fault-prone. The obvious conclusion is that reuse information contributes significant predictive power. The study did not evaluate the effectiveness of a model based solely on the reuse information.

Among several significant differences between the research described in [7] and ours, the most important is the primary goals. Their predictions classify all modules into two categories: fault-prone or not fault-prone, where fault-proneness is defined based on an arbitrary and fixed number of faults in a module. In contrast, our model assigns a number of expected faults to each file and ranks the files from most to least fault-prone, without choosing an arbitrary cutoff point for fault-proneness. Another important difference is the level of granularity at which the predictions are being done. Khoshgoftaar et al. are working at the module level, which is much coarser than the file level at which we are working. Using this coarser-grained prediction will increase the difficulty of pinpointing where faults are likely to be and therefore make it less useful for testing and debugging.

Another difference is the extent of the studies. While Khoshgoftaar et al. applied their models to a portion of a single release of a system, we have applied ours to 17 successive releases of one system, as well as to a second unrelated production system.

In [4], Graves et al. report a study using the fault history for the modules of a large telecommunications system. The subject system contained roughly 1.5 million lines of code, divided into 80 modules, each a collection of files. They first considered different file characteristics, and found that module size and other standard software complexity metrics were generally poor predictors of fault likelihood. Their best predictors were based on combinations of a module's age, the changes made to the module, and the ages of the changes. Although they did use the models to predict faults in files, the primary focus of their work was the identification of the most relevant module characteristics and the comparison of models. They applied their model to make predictions using a single two year time interval, but did not discuss the extent to which the predictions were effective. At the conclusion of their study they described applying their model to a one year interval in the middle of the original two year interval and were

troubled to discover that certain parameter values differed by an order of magnitude between the two time periods.

There are also significant differences between this work and ours. Graves et al. focused on identifying the most relevant module characteristics or groups of characteristics and comparing the apparent effectiveness of the proposed models, rather than predicting faults and using them to guide testing. As with the work described in [7], they worked at the module rather than the file level and only looked at a single release, albeit one of long duration.

Ohlsson and Alberg [12] evaluate the ability of various design metrics to predict the fault-proneness of software modules before the coding has been carried out. For the Ericsson telephony software of their case study, 60 percent of the detected faults were located in 20 percent of the modules. The top 20 percent of modules identified by their best predictors contained from 43 percent to 47 percent of the faults in the implemented code. The key differences between their work and ours are that their predictions are based entirely on precoding characteristics of the system design and that they make no use of the software's prior history. The paper contains no information about the life history of the system analyzed for the case study, so we do not know whether the design metrics are being applied to a new design for a new system, or to designs that have been in existence and already used for earlier implementations. In either case, the study is apparently applied to only a single instance or version of the design. (Unfortunately, the published version of this paper contains a number of misprints that occasionally render its meaning obscure.)

10 Issues

The prediction model has been shown to be capable of successfully identifying the most fault-prone files for multiple releases of two different software systems. However, before testing practitioners can use this prediction method to help guide their testing efforts, several issues need to be addressed. These include:

- Software change databases are usually not organized in a way that supports the extraction of information needed to do fault prediction.
- It is frequently difficult to determine which change reports in a software project's database represent faults.
- The project's software developers and testers may supply inaccurate information to the database.
- Statistical expertise is needed to apply the method.
- The current form of our results, identifying the files of a system that are most likely to contain faults, is not ideal for testers. System testers, the most likely users of fault predictions, typically base their testing on functionality described in a specification or requirements document, rather than on files.

These issues can be addressed through education of software developers and testers, construction of a tool to automate much of the prediction process, and some redesign of both the internals of project databases and the external interface to the databases.

A major issue to overcome is the need to identify, extract, and analyze a large amount of information from a database that was not intended for the purpose of fault-proneness prediction. In our study, it was frequently difficult to identify the necessary data, and always cumbersome to do the extraction and analysis. On the positive side, since MR information was stored in a system that is integrated with the project's version control system, almost all of the necessary data actually resided within a common database. An automated tool should be able to extract all the needed information and supply it to the statistical engine that generates the fault predictions.

As reported in previous sections, one of the major problems we faced was how to determine which change reports represent faults. As discussed in Section 3, we used a heuristic to make this determination for the inventory system, while for the service provisioning system, there were few enough faults that we could individually read each MR description and make a determination.

Since neither solution is entirely satisfactory, in the first case for accuracy and in the second case for efficiency, we have convinced a third project with which we are working to modify the MR recording form to include a field that gives the person initiating the MR three options. It asks whether this MR is a fault ("Is this a bug?"). The person can then click on one of three choices: yes, no, and I don't know. In most cases of its use so far, the response on this field has been either yes or no with only a small percentage being the "I don't know" option. If we can make this a standard part of the MR form for all projects and if those initiating an MR conscientiously fill in this information, that would go a long way to solving this problem.

The above issue relates to the third problem concerning the database: data may be missing or inaccurately entered. If the person initiating the MR does not think a field is important or will be used, they may leave it blank, fill it in with a default value, or fill it in with an arbitrary value. Since the MR reporting form was not designed to be used to predict file fault-proneness, data that we now need may not have been reported or may have been recorded inaccurately. Some fields can be counted on to be accurate. For example, the textual description of the reason why an MR is being created should always be accurate. This is how the person making the change, who is generally not the person writing the MR, will know what to change and how to change it. However, since the reason for making a distinction between an initialization, a modification, and an enhancement is often unclear, this field may be treated as unimportant and left as the default value, providing misleading information. When using an entry as a surrogate for information that is not directly available, such as whether or not an MR is a fault, it is essential that all fields have been accurately and consistently entered. The primary solution to this issue is education of those writing MRs.

As mentioned in Section 3, each MR has a severity associated with it, and we had initially expected that this rating could be used to identify high severity faults that are of particular importance to projects. Ideally, this would allow us to develop models designed specifically to predict high severity faults or, perhaps, a fault count

where high severity faults receive extra weight. However, because we learned that these severity ratings were highly likely to be inaccurate, we decided not to use them in our analysis. Again, the education of personnel writing MRs is a major issue.

The actual statistical modeling and fault prediction clearly has to be packaged within an automated tool, as the necessary statistical expertise cannot generally be expected of a typical developer or tester. The tool's interface would allow the user to specify the percentage of files to be targeted. The tool will extract the relevant information about each file, properly weight each characteristic, and then return a list of files determined to be most fault-prone in decreasing order of expected number of faults. It will not be necessary for the tester to understand anything about how the prediction was done.

The final issue involves the form of the predictions produced by the model versus the type of information that is normally used by system testers. System testers generally are unaware of the system's file structure. The unit of discourse for them is a functionality unit—a description of a task the system is to perform. The prediction process, and an eventual prediction tool, does not know about functionality, only about files. To make the results most useful for testers, a second tool that maps files to functionality should be developed and integrated with the prediction tool. In general, such a tool would keep track of relations among all the software development artifacts, and would affect the way software development is done.

Several types of artifacts are associated with every software system and its components. The first is the code itself, a collection of files that together comprise the software system. The second type of artifact is the specification that documents and describes what the software is to do—the functionality that must be included in the system. Generally, between the specification and the implementation are design documents that describe which parts of the required functionality are to be included in each file. Although many development projects do not keep all of these documents current as the system is modified and augmented with new and changed functionality, it is essential that this be done systematically. In that way, when the prediction tool indicates that a file is particularly fault-prone, the mapping tool can point the tester to the functionality that the file is intended to implement.

11 Conclusions and Future Work

We have continued our investigation of software fault behavior, furthering the goal of predicting which particular files in a software system are most likely to contain faults in a new release. Using these predictions, testers can focus their efforts and obtain effective test results more quickly than would otherwise be possible. This implies either that a given amount of resources can be used to do more testing, or that a given amount of testing can be done with fewer resources. The result should be either more reliable and dependable systems at the same cost, or similar reliability and dependability at a lower cost.

The predictions are based on a negative binomial regression model whose variables were selected by using the characteristics we identified as being associated with high fault files. The predictions that we were able to make using this model were in fact very accurate whether they were based on faults found at all stages of development, or restricted to just those faults identified after unit testing was completed. Overall, in both cases, the average percentage of faults contained in the 20 percent of the files identified by the model as likely to be most problematic, was at least 83 percent for the inventory system. We also applied the model to another software system, and again the top 20 percent of the files contained 83 percent of the faults.

We investigated the use of the model to predict which files are likely to have high fault densities. We saw that although using the model in this way identified files with fewer faults than when the files were ordered by the predicted fault count, they always contained significantly more faults than a randomly selected set of files would be expected to contain and significantly fewer lines of code than in the set of files selected by fault count ordering.

We expect to address several related issues in the future. Can the current model be used to predict the files that contain a given percentage of the faults? For example, the user might ask that the model identify a set of files associated with 90 percent of the faults. It would be interesting to see how close that prediction comes to the percentage of actually detected faults and, also, what percentage of the files have to be included in order to reach the predicted percentage of faults with high confidence.

Because using our negative binomial regression model requires a certain level of statistical expertise, we considered a simplified model that involved only sorting files by length and selecting the 20 percent of the files that are the largest. This was tried because in the case study we found that size was the most significant factor influencing the number of faults, and it is easy to do, requiring no particular statistics knowledge. We found that for both the inventory system and the provisioning system, this highly simplified model achieved a predictive accuracy roughly 10 percentage points lower than the full model.

Doing the predictions is time-consuming, difficult, and requires statistical expertise. Therefore, we are currently designing and building a tool to automate the prediction process. This tool will have to interface with the current MR/version control system in order to make the entire process seamless. We have also identified another very large software project with a significant number of releases, and have begun to prepare its data for analysis and application of our prediction model. Since this is a relatively new project that is under active development, we are optimistic that our prediction can help its testers use this information to positively impact the project's quality. Thus, we expect both to continue investigating the use of our model for targeting testing through prediction, and to help the test team apply the model to improve the efficiency and reliability of their software system.

If, in fact, our predictions are used during system development and testing, we may find that the results affect the faults identified. Such an effect might require the

prediction method to be tuned as development progresses. We plan to see if this is an issue in future work.

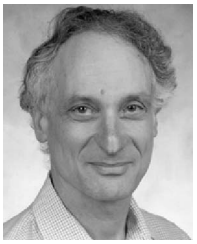
Overall, we are convinced that the sort of prediction that we have been able to do with our model is of potentially enormous use to testing practitioners, leading to software systems that are of higher quality at lower costs than we can currently attain with our current technology. We expect to continue refining the model as different potential uses are identified, and continue building the tool to make it fully automated so that it can be widely adopted by practitioners.

ACKNOWLEDGMENTS

This work could not have been carried out without the cooperation and assistance of developers and testers at AT&T. Jainag Vallabhaneni, Jim Laur, Joe Pisano, Chaya Schneider, Steve Prisco, and Henry Gurbisz were all generous with their time and willingness to answer our questions about their systems. Raju Pericherla helped greatly with data acquisition and converting raw data into structured formats, as well as finding and adapting a metrics tool for Java code. The authors' colleagues Ken Church, Dave Korn, and John Linderman provided helpful assistance with Unix and Microsoft utilities.

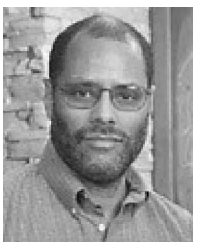
REFERENCES

[1] E.N. Adams, "Optimizing Preventive Service of Software Products," *IBM J. Research Development*, vol. 28, no. 1, pp. 2-14, Jan. 1984.
[2] V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM*, vol. 27, no. 1, pp. 42-52, Jan 1984.
[3] N.E. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Trans. Software Eng.*, vol. 26, no. 8, pp. 797-814, Aug. 2000.
[4] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Trans. Software Eng.*, vol. 26, no. 7, pp. 653-661, July 2000.
[5] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust Prediction of Fault-Proneness by Random Forests," *Proc. Int'l Symp. Software Reliability Eng.*, Nov. 2004.
[6] L. Hatton, "Reexamining the Fault Density—Component Size Connection," *IEEE Software*, pp. 89-97, Mar./Apr. 1997.
[7] T.M. Khoshgoftaar, E.B. Allen, K.S. Kalaichelvan, and N. Goel, "Early Quality Prediction: A Case Study in Telecommunications," *IEEE Software*, pp. 65-71, Jan. 1996.
[8] T.J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, pp. 308-320, 1976.
[9] P. McCullagh and J.A. Nelder, *Generalized Linear Models*, second ed. Chapman and Hall, 1989.
[10] K-H. Moller and D.J. Paulish, "An Empirical Investigation of Software Fault Distribution," *Proc. IEEE First Int'l Software Metrics Symp.*, pp. 82-90, May 1993.
[11] J.C. Munson and T.M. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Trans. Software Eng.*, vol. 18, no. 5, pp. 423-433, May 1992.
[12] N. Ohlsson and H. Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches," *IEEE Trans. Software Eng.*, vol. 22, no. 12, pp. 886-894, Dec. 1996.
[13] T. Ostrand and E.J. Weyuker, "The Distribution of Faults in a Large Industrial Software System," *Proc. ACM/Int'l Symp. Software Testing and Analysis (ISSTA 2002)*, pp. 55-64, July 2002.
[14] T. Ostrand, E.J. Weyuker, and R. Bell, "Using Static Analysis to Determine Where to Focus Dynamic Testing Effort," *Proc. IEE/ Workshop Dynamic Analysis (WODA 04)*, May 2004.
[15] M. Pighin and A. Marzona, "An Empirical Analysis of Fault Persistence through Software Releases," *Proc. IEEE/ACM Symp. Empirical Software Eng.*, pp. 206-212, 2003.
[16] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold, "Test Case Prioritization," *IEEE Trans. Software Eng.*, vol. 27, no. 10, pp. 929-948, Oct. 2001.
[17] SAS Institute Inc., "SAS/STAT User's Guide," Version 8, SAS Inst., Cary, N.C., 1999.



Thomas J. Ostrand is a research member of the Management Systems Division at AT&T Labs in Florham Park, New Jersey. He was previously with the Software Research Groups of Siemens Corporate Research and Sperry Univac, and was also a faculty member of the Rutgers University Computer Science Department. His current research interests are in software testing methods and the practical application of testing theory. He is especially interested in empirical assessment of testing methods. Dr. Ostrand has been an editor of the *IEEE Transactions on Software Engineering*, and a program committee member of various software engineering and testing conferences. He was the program chair of the 1994 ACM International Symposium on Software Testing and Analysis. He is a past Member-at-Large of the Executive Committee of ACM SIGSOFT, and is currently a member of the ACM and the IEEE.

Elaine J. Weyuker received the PhD degree in computer science from Rutgers University, and the MSE degree from the University of Pennsylvania. She is currently an AT&T Fellow, performing research in software testing and metrics and has published more than 125 papers in journals and refereed conference proceedings. She is also interested in the theory of computation and is the author of a book (with Martin Davis and Ron Sigal), *Computability, Complexity, and Languages* (second ed., Academic Press). She was on the faculty of the Courant Institute of Mathematical Sciences of New York University, was a faculty member at the City University of New York, a systems engineer at IBM and a programmer at Texaco, Inc. She was elected to the National Academy of Engineering, is an ACM Fellow, an IEEE Fellow, and an AT&T Fellow. She was the 2004 recipient of the IEEE Harlan D. Mills award, a recipient of the YWCA Woman of Achievement Award, and was named the Outstanding Alumni at the Rutgers University 50th Anniversary celebration. She was also the recipient of the AT&T Chairman's award for her mentoring activities and efforts to foster diversity. She is the cochair of the ACM-W committee, a member of the steering committee of the Coalition to Diversify Computing, a member of the Rutgers University Graduate School Advisory Board, and was a member of the Board of Directors of the Computing Research Association (CRA). She is a member of the editorial boards of *IEEE Transactions on Software Engineering*, *IEEE Transactions on Dependable and Secure Computing*, *IEEE Spectrum*, the *Empirical Software Engineering Journal*, and the *Journal of Systems and Software*, and was a founding editor of the *ACM Transactions of Software Engineering and Methodology*. She was the Secretary/ Treasurer of ACM SIGSOFT and was an ACM National Lecturer.



Robert M. Bell is a member of the Statistics Research Department at AT&T Labs-Research. His research interests include survey research methods, analysis of data from complex samples, and record linkage methods. He is a fellow of the American Statistical Association. He is currently a member of the Committee on National Statistics organized by the National Academies and a member of the board of the National Institute of Statistical Sciences.