Test Coverage and Post-Verification Defects: A Multiple Case Study

Audris Mockus
Avaya Labs Research
233 Mt Airy Rd
Basking Ridge, NJ
audris@avaya.com

Nachiappan Nagappan
Microsoft Research
One Microsoft Way
Redmond, WA
nachin@microsoft.com

Trung T. Dinh-Trong
Avaya Labs Research
233 Mt Airy Rd
Basking Ridge, NJ
ttdinhtrong@avaya.com

Abstract

Test coverage is a promising measure of test effectiveness and development organizations are interested in cost-effective levels of coverage that provide sufficient fault removal with contained testing effort. We have conducted a multiple-case study on two dissimilar industrial software projects to investigate if test coverage reflects test effectiveness and to find the relationship between test effort and the level of test coverage. We find that in both projects the increase in test coverage is associated with decrease in field reported problems when adjusted for the number of pre-release changes. A qualitative investigation revealed several potential explanations, including code complexity, developer experience, the type of functionality, and remote development teams. All these factors were related to the level of coverage and quality, with coverage having an effect even after these adjustments. We also find that the test effort increases exponentially with test coverage, but the reduction in field problems increases linearly with test coverage. This suggests that for most projects the optimal levels of coverage are likely to be well short of 100%.

1. Introduction

Among software quality improvement activities testing is arguably the most important [9]. It is, therefore, of particular interest to evaluate and understand how good is a particular set of tests with respect to its ability to detect the most disruptive (post-release) defects.

Clearly, skilled testers are more likely to produce more effective tests, but it is preferable to assess the test effectiveness using quantitative and easy-to-obtain measures of test performance that are applicable in software development practice. Because testing is a pre-release activity, the measures of test performance can not be based on the most relevant post-release observations of quality. Apart from a simple measure of test count or testing effort that does not distinguish individual tests (or hours spent testing) according to their ability to detect defects, a widely used measure of test effectiveness is test coverage (subsequently we will simply use the term "Coverage"). There is a variety of coverage metrics from simpler class, function, and statement coverage and to more sophisticated branch and even path coverage (see, e.g., [2]). The principal assumption behind the coverage metric is that if a branch or a statement contains a flaw, it can not be detected unless at least one test exercises that statement or branch. Therefore, it is argued, higher Coverage should lead to detection of more flaws in the code and, if they are fixed, to better release quality.

There are a number of potential flaws with this argument. First, the coverage measure reflects the percent of statements covered but does not consider if these statements are likely to contain a flaw. Therefore, it may be possible to create two test sets with the same coverage measure but markedly distinct ability to detect post-release defects. Second, the fact that a statement or a branch was executed does not imply that all possible data values have been exercised by the test [17]. This is of particular concern for systems with a simple control flow that can be easily covered with a few test cases. Third, even if a statement is executed and a failure has occurred, the test may not be able to detect it if the appropriate assert statements were not inserted or if the output was not recognized as being erroneous.

Despite these flaws, Coverage appears to be a promising measure due to its wide use in practice and prevalence of practical recommendations or requirements that we gathered through informal survey of various software practitioners in our organizations and professional contacts. They could be referred to as software "folklore":

1. The 70% coverage is needed prior to acceptance for a system test or a release.
2. It is often not effective to get Coverage above 70% because it requires a lot more work than achieving Coverage of 70% or less.
3. Going above 70% coverage leads into exception handling code.

978-1-4244-4841-8/09/$25.00 ©2009 IEEE

4. By introducing Coverage we have observed that developers who did not write unit tests have started doing so.

The 70% threshold is used for illustration purpose only. In different contexts the actual number might be higher or lower. Nevertheless, these requirements and recommendations beg the following questions:

1. What is defect detection effectiveness curve for X% coverage.
2. What is the effort needed to obtain X% coverage.
3. What is the cost effective coverage measure for a particular set of quality requirements?

Unfortunately, there appear to be no empirical studies of industrial software (see Section 6 for more details) that shed light on these questions. Therefore, we set out to conduct an empirical study to observe the relationship between the test coverage measure, the number of post-system verification (post-SV) defects, and the effort to produce tests. We have looked at industrial systems in order to observe common practices of testing and coverage, and, therefore, we could not observe what would happen for different level of coverage for the same code. Instead we looked at the variations in coverage among components of the system and related them to the probability that a defect affecting a component would be observed after system verification (SV).

While it would be also interesting to know which tests have detected defects, a significant part of the correction activity often occurs at the unit test level where the developer corrects issues prior to code submission and no defects are reported in the problem tracking system.

Table 1. Summary of project context

Area	Avaya	Microsoft
Language	Java	C/C++
Size	1 MLOC	40+ MLOC
Domain	Application	System
Team size	≈ 100	1000+
Users	≈ 1000	$\approx 100M$
Testing phase	Unit	System
Tools	JUnit	In-house
Types of coverage	Statement, branch	Block, Arc
Granularity	Files	Binaries

The primary hypothesis of our study is that test coverage affects post-SV defect rate. Drawing general conclusions from empirical studies is difficult because there are many context variables that are often difficult to quantify in software engineering. For this reason, we can not assume that the results of a study generalize beyond the specific environment where it was conducted. Finding similar results in a different context provides more confidence that the theory applies more generally [3]. Therefore, to strengthen the external validity of the results we decided to conduct a multiple case study in two radically different industrial contexts compared in Table 1. Not only the projects come from two unrelated companies, the application domain, the programming language, the size of software product, development team, and user base were vastly different. One of the initial challenges involved defining and operationalizing measures that were comparable across projects. In particular, the coverage could be measured at the class (file) and method level in the first project, but it was only available at the level of binaries (executables and libraries) in the second project. Furthermore, due to different technical and confidentiality requirements we cannot report all the results in exactly the same fashion.

A significant contribution of this paper is the analysis of empirical data from a multiple case study on two different industrial contexts. The results indicate that the increase in test coverage is correlated with the decrease in field reported problems (when controlling for number of edits), supporting the use of code coverage measurement as a quality control criteria. The data, however, indicate that the test effort increases exponentially with test coverage, suggesting that for many projects, 100% code coverage is not always the most cost-effective criteria. Further, our analysis suggested that factors such as code complexity, application domain, developer expertise, and remote development location also affect both fault potential and coverage, implying that user interface functionality, less experience developers, and remote development locations might benefit from utilizing increased coverage.

2. Method and Measures

First, we take into account business needs and published work to pose precise hypotheses about the relationship between test coverage and the software quality. Our study then uses semi-structured interviews and data from software repositories of change, defect, and coverage information to evaluate the plausibility of each proposition. Data collected from version control, problem tracking, and test coverage were used to fit regression models that quantify these observed relationships. Interviews were used to validate repository data attributes and analysis results.

We reformulate our initial hypotheses based on the findings of this multiple case study to refine the theory of how coverage affects software quality and to conduct further replication studies on the subject.

2.1. Avaya project context

The Avaya project under study is a call center reporting system. The project uses SQL, C, CPP in addition to the bulk of code in Java language that is studied here. It is a

978-1-4244-4841-8/09/$25.00 ©2009 IEEE

completely new release (unlike prior reporting systems that were written entirely in C language) that has been released for approximately one year at the time of the writing.

The project under study used ClearCase to track changes to the code and a proprietary system later replaced by ClearQuest to track software defects. Code coverage was measured using the Cobertura tool. We have obtained the change history of the code using the "lsh" command of the ClearCase. Except for changes done on the private branches, all code commits had modification request (MR) identifier that we automatically extracted from the change comments. That MR identifier was used to obtain more information about the change. In particular, we used information from ClearQuest and a proprietary tracking system to identify the phase an MR was reported. In particular we identified post-SV MRs that were found during alpha and beta trials and from customers of the final release. We also identified MRs found in system integration and testing. Finally we counted the number of MRs associated with each file (we had to map MR IDs transferred from an older system to MR IDs in the new system to avoid double-counting of these MRs. To track code coverage we have used daily (obtained over the span of two years) and final snapshots of Cobertura coverage reports. We used two numbers from these reports: the number of statements $n_s(f,t)$ and the number of covered statements $n_c(f,t)$, where f represents the file and t represents the snapshot in time when the coverage was obtained. The coverage numbers we use in our models are obtained by dividing the maxima of the ratio of covered and total statements: $C(f) = \max_t \frac{n_c(f)}{n_s(f)}$. Obviously, other measures, including cyclomatic complexity, non-commentary lines of code (NCSL), and FanOut varied over time. For each one we have selected the maximum over the observation period. We have selected the changes, MRs, and coverage for the interval prior to the release date of the software as the observation period. The only exception were customer reported MRs: the entire interval including eight months after the release date was used to identify files with field defects. We looked at the alternative measures of coverage, cyclomatic complexity, FanOut, and NCSL by calculating at averages or minima instead of maxima over time, but the regression results were similar to the ones obtained using $C(f)$.

2.2. Microsoft project context

We investigated the Microsoft Windows Vista system. The size of the code base analyzed was 40+ Million LOC. Code coverage in Windows takes places at the level of binaries (.lib, .exe, .dll etc.). We also collect and map post-release failures obtained for Windows Vista that were found in the field six months after the public release of Windows Vista. Code size and complexity metrics were selected to be similar to the ones in the Avaya case study and were collected on a per-binary basis at the release point of the system to the field. Code coverage measures were slightly different and are described below.

For the code coverage measures in the binaries we use arc and block coverage analogous to statement and branch coverage. Figure 1 represents a simple example where the arcs are shaded lightly (yellow) and blocks are shaded darkly (grey). The code coverage tools are based on the Vulcan binary analysis framework at Microsoft [19].

```
#include <ctype.h>
#include <stdio.h>
main(int argc, char **argv)
{
   int flag;
   if (argc < 2 || !isdigit(argv[1][0])) {
       printf("Bad argument(s)\n");
       exit(1);
   }
   switch(atoi(argv[1])) {
   case 1: case 2: case 3:
       flag = 1;
   break;
   case 4:
   case 5:
       flag = 2;
   break;
   default:
       flag = 0;
       break;
   }
   if (flag > 0) flag = 1; else flag = 0;
   printf("Flag is %s\n", flag ? "1" : "0");
   exit (0);
}
```

Figure 1. Example of block measurement

Block coverage: A (basic) block is a set of contiguous instructions (code) in the physical layout of a binary that has exactly one entry point and one exit point. Calls, jumps, and branches mark the end of a block. A block typically consists of multiple machine-code instructions. The percentage of blocks covered during testing constitutes the block coverage measure. It is most similar to the "Coverage" measure in the Avaya project shown in Table 2.

Arc coverage: Arcs between blocks represent the transfer of control between basic blocks (due to conditional and unconditional jumps, as well as control falling through from one block to another). Similar to block coverage the proportion of arcs covered in a binary constitute the arc coverage.

Code complexity: McCabe's Cyclomatic Complexity metric [12] measures the number of linearly-independent paths through a program module. It is used to measure code

978-1-4244-4841-8/09/$25.00 ©2009 IEEE

complexity in a binary and is similar to the code complexity measure in the Avaya project shown in Table 2.

Frequency of Churn: is defined as the number of times a binary is worked upon (check-ins) during the development process. Unstable binaries often churn repeatedly due to multiple fixes thereby causing a "tail of check-ins" to be observed. It is identical to the "Delta" and similar to "MR" predictor in the Avaya project.

3. Theory

Our theoretical premise is based on the intuition that defects localized in the parts of the code that are not covered by the test execution can not be discovered by that test set. The fundamental weaknesses of this assumption is that it is not clear to what extent the defects are localized. In other words, it may be the case that most important defects can be discovered by test sets that do not achieve a complete or even high levels of coverage. A practical weakness of the approach is that even if a particular part of the code is covered by a test set, there is no guarantee that all or any flaws embedded within that part of the code would be exposed by the test. Despite these weaknesses, it is reasonable to expect that, given all other conditions being similar, increase in code coverage should reduce the defects observed post-SV.

Given that the individual files/binaries have different functionality, are written and tested by different developers and testers, are of different size, and have different history it is unreasonable to expect that "all other conditions are equal". To deal with these variations we have attempted to adjust for the factors that are known to affect post-SV defects. There are a number of studies in this area that use a variety of factors to predict post-release defects (see, e.g., [4, 7, 16, 20]). However, it appears that a single predictor of the total number of changes made to the file is the most important predictor that is almost impossible to improve upon (see, for example, [7]) Furthermore, predictors such as modified lines of code and number of changes to the module are often highly correlated. Therefore, our primary hypothesis is that an increase in module coverage adjusted for the changes made to the module should reduce post-SV defects.

Our fundamental hypothesis is as follows:

Hypothesis 1 Increasing the level of coverage would decrease the defect rate.

To respond to more practical questions from development organizations we also pose a second hypothesis:

Hypothesis 2 Progressively more effort is needed to increase the coverage by the same amount for higher levels of coverage.

4. Results

All measures for the Avaya study are presented for Java code only at the granularity of a file (class). Because our response measure is post-SV defects, we have excluded all the code that is not executed by customers and, therefore, can not be related to such defects. In particular, all test cases and stubs as well as build, test, and other development support tool code is excluded from our reports.

The notation in the subsequent tables is described below.

- GMR — number of post-SV MRs.
- MR — number of MRs excluding post-SV MRs.
- NCSL — non-commentary lines of code in all methods of a class.
- CC — cyclomatic complexity added over all methods in a class.
- FanOut — number methods/functions called added over all methods in a class.
- Delta — number of changes to a file in the version control system (added over all branches).
- Coverage — percent of statements that are covered.
- Files — number of files in a group with the coverage level shown in the table header.

4.1. Coverage and post-SV defects

First we investigate if the level of coverage is correlated with the defect rate. Figure 2 shows the number of field MRs (GMR) in a file normalized by the variables MR, NCSL, CC, and FanOut for files with different levels of coverage[1]. Each measure is scaled to fit on the same display. Both averages and standard errors are shown to indicate variability of the estimates. There appears to be a consistent decrease in the file field MR rate with increase in coverage that is independent of normalization (we used a variety of adjustment measures ranging from MR to FanOut to check if the result is sensitive to such choice). While the standard errors are fairly large, there is a statistically significant difference between the rates for the largest and smallest coverage. As in Table 2, such level of detail could be published only for the Avaya project.

Table 3 shows Spearman correlations [2] among the variables for the Avaya project. Coverage is negatively correlated only to post-SV defects (GMR), but the correlation is very small (though significant at $p - val < 0.0001$ as are all other correlations).

[1] The values in Figure 2 can not be obtained from the values in Table 2 because Figure 2 uses ratios averaged over files

[2] insensitive to the skewed distributions of these quantities

978-1-4244-4841-8/09/$25.00 ©2009 IEEE

Table 2. Counts added over all files for the five levels of coverage for the Avaya project

	None	$(0\%, 30\%]$	$(30\%, 60\%]$	$(60\%, 80\%]$	$> 80\%$
GMR	273	128	23	29	30
MR	6551	4188	915	1389	2754
NCSL	278071	127823	32136	41061	78487
CC	11641	12956	3620	4517	8241
FanOut	4470	4193	949	1425	3378
Files	994	773	115	160	430

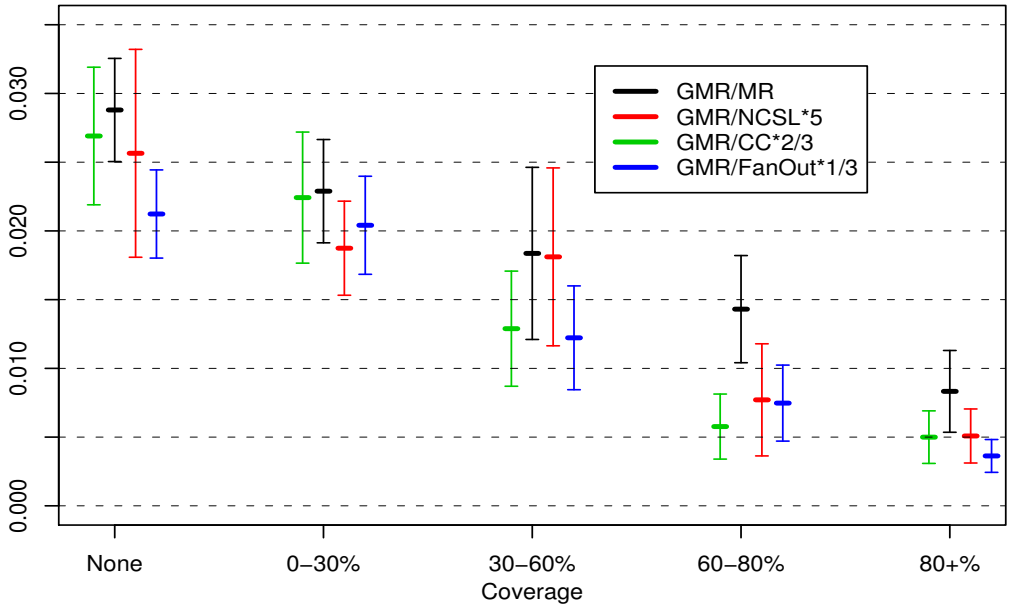

Figure 2. The averages and standard errors of field MR rates for the Avaya project

For comparison, Table 4 shows Spearman correlations among the variables in the Microsoft project. Note that here the increase in coverage is positively correlated with the number of failures (again, all correlations are small and highly significant). This may be due to the fact that frequently changed (and, therefore, more likely to have a failure) binaries are also more likely to be more extensively tested. This phenomena may explain such low negative correlation in the Avaya study as well. It is, therefore, important to adjust for the differences in the propensity for failure among files or binaries.

To accomplish that and to have better understanding of the factors affecting the failures we fit a regression models to Avaya and Microsoft data. Because of the small user base, very few files have more than one post-SV defect in the Avaya project. Therefore, a logistic regression with the response being an indicator of the file containing such fault is a suitable model. To adjust for the different propensity of the files to have post-SV defects we include the number of pre-release MRs (MR) as a predictor in addition to the extent of coverage. We included MRs as a base predictor because they have been shown to be the best predictor of faultiness in, for example, [7]. The number of deltas was strongly correlated to MRs and was not added to avoid collinearity problems. Of the remaining predictors FanOut was the least correlated to the MR predictor and after adding it to the model the remaining predictors were strongly correlated to one of the predictors in the model and, therefore, were not included. Results in Table 5 show that increased coverage is associated with a lower probability of a post-SV defect.

In the Microsoft data the system has a sufficient number of users to provide ample counts of failure reports for many binaries so that a simpler liner regression approach may be used. However, the response variable is highly skewed and needs to be transformed using logarithms. Because the MR

978-1-4244-4841-8/09/$25.00 ©2009 IEEE

Table 3. Correlations among the variables in the Avaya study.

	$GMR > 0$	MR	NCSL	CC	FanOut	Delta	Coverage
$GMR > 0$	1.00	0.35	0.14	0.18	0.16	0.15	−0.07
MR	0.35	1.00	0.48	0.25	0.25	0.77	0.13
NCSL	0.14	0.48	1.00	0.53	0.54	0.59	0.09
CC	0.18	0.25	0.53	1.00	0.82	0.26	0.21
FanOut	0.16	0.25	0.54	0.82	1.00	0.29	0.18
Delta	0.15	0.77	0.59	0.26	0.29	1.00	0.24
Coverage	−0.07	0.13	0.09	0.21	0.18	0.24	1.00

Table 4. Correlations among the variables in the Microsoft study.

	Failures	LOC	Delta	FanOut	CC	Block Coverage	Arc Coverage
Failures	1	.699	.804	.529	.531	.100	.116
Size	.699	1	.848	.813	.820	.049*	.067
Delta	.804	.848	1	.663	.665	.128	.140
Fan Out	.529	.813	.663	1	.977	.121	.143
CC	.531	.820	.665	.977	1	.142	.155
Block Coverage	.100	.049*	.128	.121	.142	1	.990
Arc Coverage	.116	.067	.140	.143	.155	.990	1

Table 5. Logistic regression for Avaya project with response indicating post-SV MR. There are 2472 observations and the three predictors explain 20% of the deviance. AIC is 1249

	Estimate	Stderr	z val	Pr(>|z|)
(Intercept)	−4.72	0.205	−23	0.00
$\log(MR)$	1.18	0.084	14	0.00
$\log(FanOut)$	0.34	0.067	5	0.00
Coverage	−1.51	0.252	−6	0.00

Table 6. Linear regression for Microsoft project with the response of number of customer reported failures. The number of observations is confidential the adjusted $R^2 = 0.66$, i.e., the model explains 66% of the variance.

	Estimate	Stder	t-val	Pr(>|z|)
(Intercept)	NA	NA	12.3	<0.0005
$\log(Delta)$	NA	NA	58.5	<0.0005
$\log(FanOut)$	NA	NA	-5.8	<0.0005
Block Coverage	NA	NA	−3.5	0.001

variable was not available in this study, we used the Delta variable, which is most similar, to adjust for the differences among binaries. As in the Avaya study, the predictor least correlated to Delta and Block Coverage is FanOut, which we also include in the model to adjust for the possible differences in failures among binaries. Results in Table 6 show that increased coverage is associated with a lower number of customer reported failures. The only difference is that the FanOut has an opposite sign. We further investigate the validity of this result using structured and semi-structured interviews in section 5.

Both studies are consistent with Hypothesis 1 indicating that higher levels of coverage are associated with better quality. We further investigate the validity of this result in Section 5.

4.2. Practical Implications

We converted the regression coefficients of Table 5 into more interpretable terms of probability that a class will have a post-SV defect. The result, presented in Figure 2, shows the ratio of roughly three between the highest levels of coverage and no coverage. To refine these estimates we have fitted a model with predictors indicating five levels of coverage as in Figure 2. For illustration we present the predicted values of post-SV MR for classes that had three and six MRs prior to release date with the median value of FanOut in Table 7 (in this illustration MR and FanOut are fixed and, therefore, do not affect the post-SV MR). Surprisingly, the largest decrease in failure probability occurs with the high-

978-1-4244-4841-8/09/$25.00 ©2009 IEEE

Table 7. The predicted probability that a class will have a post-SV MR for classes with three and six pre-SV MRs.

	None	$[0, 30\%)$	$[30, 60\%)$	$[60 - 80\%)$	$[80, 100\%]$
$MR = 3$	0.07	0.056	0.050	0.040	0.018
$MR = 6$	0.13	0.108	0.097	0.077	0.035

est levels of coverage where the probability halves by going from $60-80\%$ coverage to $80-100\%$ coverage and changes less for the lower levels of coverage. This fact makes it difficult to determine a cost-effective level of coverage because it takes a lot of effort to reach the highest levels of coverage (see Section 4.3).

4.3. Coverage and Test Effort

Observing that higher levels of coverage are associated with better quality, raises an important practical question of how hard it may be to increase coverage. To find such a relationship we have selected JUnit test case classes and their corresponding classes in the executable code and looked at the relationship between the coverage and the amount of effort spent in creating the test case classes. We noticed that developers ussually name the test case classes by adding the prefix "Test" into the name of the classes under test (e.g., the test cases for a class, Utility, are ussually included in the class, TestUtility). Only classes that have such name matching mechanism with the test case class name are included in the analysis. We used the number of changes to the test case classes as the proxy of effort. The use of changes to measure effort has been successfully done before, see, e.g., [1, 13].

It is worth noting that a test case for a particular class may affect coverage for other classes as well. It appears that this effect is likely to be fairly small given the relatively low coupling among the classes that have test cases. In this project the estimated number of changes per month for the developers that were involved in modifying this particular set of test cases was 58 with a 95% confidence interval[3] between 48 and 71.

To obtain the relationship between the number of changes to the test cases and the level of coverage of the associated class, we fit a linear model with the level of coverage as the response. Apart from the number of changes to the test cases, only FanOut[4] has a coefficient that is significantly different from zero. Perhaps for classes that make a large number of calls it is more difficult to achieve high levels of coverage. Table 8 shows the results of the regression model. The response was transformed using a transformation commonly used to make a proportion distributed more like a Normal distribution. The predictors were transformed using logarithmic transformation. While R^2 is fairly low (the model does not explain a lot of variance in the coverage levels), the number of changes to the test case is a significant predictor of test coverage.

Table 8. Linear regression with response $\sqrt{\arcsin Coverage}$ and numbers of changes to test class and average fan-out of the tested class. The number of observations is 215 and adjusted R^2 is 0.13.

	Estimate	Stderr	t-val	p-val
(Intercept)	0.36	0.10	3.5	0.00
$\log(Changes + 1)$	0.21	0.04	5.6	0.00
$\log(FanOut + 1)$	-0.06	0.03	-1.7	0.08

Table 9. Predicted levels of coverage for different numbers of changes to the test class and median Fan-Out of 7.

Changes	Predicted Coverage	95% CI
1	0.15	[0.07, 0.26]
7	0.45	[0.38, 0.53]
14	0.62	[0.53, 0.72]
50	0.92	[0.76, 1.00]

Table 9 shows the predicted relationship between the amount of effort put in the test case (50 changes represent slightly less that one person-month of effort) and the level of coverage. We also include the prediction for test cases with seven (the median number of changes to a test case), 14, and 50 changes. Table 9 shows that it is increasingly difficult to attain higher levels of coverage (the maximum observed changes to a test case was 134 and the corresponding

[3] We used random effects model (with developers representing random effects) and observations being the logarithm of the number of changes per month to estimate the number of changes and the confidence interval.

[4] Cyclomatic complexity and even the number of lines did not have a significant effect after adjusting for the number of changes to the test case.

978-1-4244-4841-8/09/$25.00 ©2009 IEEE

class had 80% coverage, FanOut of 26, and NCSL of 729.

The results indicate that while it may be relatively easy to reach 50% coverage, getting levels above 90% may not be feasible except in special circumstances. Unlike with JUnit, the test cases in the Microsoft project could not be mapped and hence we could not investigate Hypothesis 2 in the second study. Consequently, the external validity of this result is not as high. Nevertheless, it is consistent with our Hypothesis 2 that reaching higher levels of coverage requires disproportionally more effort.

5. Validity

The observational nature of our study implies that we can not show evidence for the causal effects. In particular, it is likely that the variations we observed among classes and binaries were partially influenced by factors other than the amount of coverage.

We investigated the different groups of files (with and without coverage) via interviews with Avaya project members to verify that:

1. the presence or absence of coverage is, indeed, accurately established;
2. the post-SV defects are accurately recorded;
3. there are no latent factors (apart from presence of coverage) that explain the different defect rates.

To accomplish this we have selected a subset of files from the two groups and selected developers that made the most changes on these files for the interview. Our interviews were semi-structured with open-ended questions asking about the functionality of the set of files in question and the possible hypotheses developers might have about the differences in the defect rates. The more structured part involved verifying the absence of coverage and the existence of post-SV defects.

5.1. Interview procedure

We interviewed a set of 6 Avaya developers that made the most changes in the code base within the last year prior to the date of the interviews. For each developer we used the following documents during the interview session:

- four MRs that the developer had recently solved. These included two GMR and two test MR.
- four deltas that the developer had committed recently. The set included two deltas that associate with MRs and two that do not associate with any MR.
- three groups of Java files that the developers changed recently. These groups have high, medium, and low code coverage, respectively. Each group includes two files – one with low bug rate and another with high bug rate. Thus, we selected six files per developer.

We interviewed the developers using a set of predefined questions (as shown in Table 10). The first three are yes/no questions, while the rest are open-ended. The first four attempt to address Validity Concern 2 and the last four aim at addressing Validity Concern 3.

In order to address the validity concern number 1, we reviewed all official code coverage reports generated by the project team. We compared these reports with the coverage data that we gathered using the method described in Section 2.1. Because one of our interviewees is the person in charge of measuring code coverage in the project, we also asked him to verify the code coverage data on six Java files that we used during the interview (as described above).

5.2. Interview Results

The results of comparing the official code coverage report and the conversation with the person in charge of code coverage measurement indicated that the code coverage data that we gathered is accurate.

Although we tried to select the most recently changed MR, not all of them were easily recalled by the interviewees. However, every MR that the interviewees remembered correctly recorded the phase it was reported, i.e., MRs that were recorded as post-SV were indeed found after SV test and MRs that were recorded as test MR were indeed found during SV or development test.

When being asked Question 3, most developers noted cases when they deliver changes relating to more than one MR at the same times, but associate the delta to only one MR. However, no example of such a case was identified among the discussed deltas.

Answers to Question 4 reveal that all the questioned deltas were committed to personal repositories. Developers further indicated that their code versioning system forces them to associate code changes to MR whenever they deliver code to the common code based. As such, all deltas committed to the common repository were associated with a certain MR.

The following is the list of reasons given for the differences of coverage between Java files:

1. "When I develop files from scratch, I tend to write more test cases for it. If I just maintain the files, I would rarely write test cases for it."
2. "The files that are complex and are used by many other files tend to be tested more."
3. "The files that are easier to test tend to be tested more. When we first deployed code coverage measurement practice, we first write test cases for these files to quickly increase the overall coverage."
4. "The files that provide a service class tend to be called by many different classes. Hence it is executed more

978-1-4244-4841-8/09/$25.00 ©2009 IEEE

during testing even though we might not intend to test them."

5. "Some areas, such as UI, are harder to test and hence tend to be tested less than the other."

6. "Code that interacts with database might not be well-tested because it may take a lot of effort to write stubs representing database behavior."

7. "Remote development locations may not be using test coverage as extensively."

To obtain similar qualitative opinions in the Microsoft study meetings were held with senior engineers with significant (> 10 years experience) at Microsoft. In summary, the main reason for the existences of low (but positive) direct correlation as shown in Table 4 was attributed to the following reasons:

1. Code covered is not correct code (as was pointed out in the introduction)

2. Code with high coverage might still have defects because a usage scenario leading to a bug could have been missed.

3. Developers inherently know that certain binaries are complex and will tend to get very high coverage in those binaries (see Item 2 in the Avaya summary). But these binaries may be the most used ones in the system that can lead to more failures being found in them, i.e. code coverage and usage profiles might not match

4. Complexity should tie into code coverage, i.e. obtaining a code coverage of 80% on a binary with cyclomatic complexity 1000 is more difficult than getting a coverage of 80% on a binary with cyclomatic complexity 10. The complexity values tied into the code coverage helps exploit the efficacy of the code coverage measures.

The above answers suggest that the following hypotheses about factors that might predict code coverage (and quality):

- Code functionality: Java files that provide different functionalities tend to have different code coverage during testing. Especially, we expect that UI code has less coverage than the average.

- Code ownership transition: Java files created by one developer and then significantly changed by the others tend to have less coverage than Java files that are created and changed by the same person. We expect that the off-shoring process might make the difference even more significant.

- Code that is easier to cover during testing tends to be tested more than code that is harder to cover.

From our experience, we have found that experienced developers who are trying to write high-quality, reliable code will include many error handling branches, attempting to account for different error cases. Generally, many of the error handling branches are very hard to cover during testing. Thus, we come up with additional hypothesis:

- Code with many error handling branches is hard to cover during testing, though it tends to have lower failure rate.

5.3. Investigating latent factors

In this section we investigate how to adjust for possible latent variables that may affect the level of coverage and the number of defects. In addition to the confounding factors that were obtained in interviews, we also used factors used to predict failures from the literature. In particular, as demonstrated in [14], developer expertise (measured in number of changes made to the code) decreases the failure. The best predictor of defects is the number of past changes [7]. Lines of code, FanOut, and cyclomatic complexity tend to have different effects in different studies but we included them here as well. The correlations between the lines of code and cyclomatic complexity was above 0.53, and between FanOut and cyclomatic complexity was above 0.82, therefore we selected only cyclomatic complexity in the validation model. The results were very similar if an alternative of FanOut or NCSL were chosen.

To adjust for the potential latent variables discovered during the interviews we have also included user interface subsystem indicator and an indicator of a subsystem that was developed in another (offshore location) as two predictors that may affect both coverage and defects. Table 11 shows the results. Cyclomatic complexity, UI subsystem, and remote development location all increase the faultiness as suggested by the interviews. Increased developer expertise (measured by the changes made to the system by a developer) is associated with the decrease in fault potential. The very high levels of coverage (above 80%) are associated with decreased fault potential.

As shown in Table 11, much of the variation in class fault potential can be explained by the differences in functionality and developer expertise. However, even after adjusting for the differences the impact of coverage remains, albeit at a smaller level. From a point of view of the code coverage effect on faultiness, the earlier model in Table 5 is perhaps more relevant. It may well be that increasing coverage for classes in UI subsystems and for less experienced developers may lead to similar (or even better) quality increases as observed in Table 5. However, the regression in

978-1-4244-4841-8/09/$25.00 ©2009 IEEE

Table 10. Interview questions.

#	Question	Accompanied items	Type of question
1	For the given MR, is it GMR?	All MRs	Yes/No
2	For the given MR, is it feature-request or bug?	All MRs	
3	Are the associated MRs recorded correctly in these deltas?	Deltas with MR	
4	Why do these deltas not associate with any MR?	Deltas without MR	Open ended
5	Why do these files have high coverage?	Files with high coverage	
6	Why do these files have low coverage?	Files with low coverage	
7	Why do these files have average coverage?	Files with medium coverage	
8	MR and code coverage data indicate that code with less than 30% coverage and code with more than 70% coverage are more likely to be buggy than code with 30% - 70% coverage. How would you explain it?	N/A	

Table 11. Logistic regression for class faultiness with 2472 observations explains 31% of the observed variance with AIC of 1086**.**

	Estimate	Std. Error	z value	Pr($>\|$z$\|$)
Intercept	3.2	1.7	2	0.053
$\log MR$	1.4	0.1	1E+01	1.8E$-$40
$\log CC$	0.15	0.055	3	0.0056
$\log(Experience)$	-4.1	0.79	-5	2E$-$07
UI Subsystem	1	0.22	5	3.9E$-$06
Remote Location	2.6	0.39	6	8.4E$-$11
Coverage > 0.8	-0.7	0.29	-2	0.015

Table 11 suggests that some of the reasons why the faultiness for some types of code and less experienced developers is higher, may be at least partially explained by the less effective test cases.

6. Related Work

Based on a literature survey of prior related work, it is surprising to note that the fundamental relationship between code coverage and quality (measured in terms of failures, defects etc.) has been rarely studied. For large projects done in industry there has been little empirical evidence presented on the relationship between coverage and quality. There does exist a significant body of work related to regression test coverage, coverage for test prioritization, fault detection, mutation testing etc. We highlight a few most closely related studies in this section. A variety of techniques attempt to prioritize test sets based on the specific changes to the code. Here we are assuming that all existing tests are executed because we look at the post-release failures.

All-edges and all-uses coverage criteria using an experiment with 130 fault seeded versions of seven programs is evaluated in [11]. The study observed that test sets achieving coverage levels over 90% usually showed significantly better fault detection than randomly chosen test sets of the same size. In addition, significant improvements in the effectiveness of coverage-based tests usually occurred as coverage increased from 90% to 100%. The size of the programs ranged from 141 LOC to 512 LOC and was many orders of magnitude lower than in our studies. All-edges and all-uses coverage using nine subject programs is investigated in [6]. Error-exposing ability was shown to be positive and strongly correlated to percentage of covered definition-use associations in four of the nine subjects. Error exposing ability was also shown to be positively correlated with the percentage of covered edges in four (different) subjects, but the relationship was weaker. The size of the subject programs used in this study were even smaller and ranged from 22 LOC to 78 LOC. In [5], five programs are analyzed, four of which were Unix utilities that ranged in size from 121 to 8857 LOC each of which were seeded with defects. Test cases were generated randomly based

978-1-4244-4841-8/09/$25.00 ©2009 IEEE

on the operational profile of the system [15] and the results indicated that with an increased coverage there was an increased reliability. However they observed that this increased coverage was independent of code complexity measures.

A point to clarify is that the related work to our study as noted above concerns the direct relationship between coverage and quality. There is a fair amount of work that have been done in the context of using coverage for regression testing (e.g., see [8, 10]). Test prioritization [18] is outside the context of our study and is hence not discussed here.

7. Summary

Despite dramatic differences between the two industrial projects under study we found that code coverage was associated with fewer field failures and a lower probability of field defects when adjusted for the number of pre-release changes. This strongly suggests that code coverage is a sensible and practical measure of test effectiveness.

A validation conducted via interviews of project participants validated the results and suggested code complexity, application domain, developer expertise, and remote development location as factors that may affect both the fault potential and coverage. The analysis confirmed these effects leading to a practical suggestion that user interface functionality, less experienced developers, and remote development locations might benefit from utilizing increased coverage.

The investigation of how much an increase in code coverage is related to a decrease in fault potential is shown in Figure 2 is compatible with the assumption that an increase in coverage leads to a proportional decrease in fault potential. Disappointingly, there is no indication of diminishing returns (when an additional increase in coverage brings smaller decrease in fault potential). What appears to be even more disappointing, is the finding that additional increases in coverage come with exponentially increasing effort. Therefore, for many projects it may be impractical to achieve complete coverage.

These are the first results that compare the direct effect of code coverage and post-release defects in two large systems drawing conclusions to build up an empirical body of knowledge. We hope other researchers will replicate this study and add to this body of knowledge.

References

[1] D. Atkins, T. Ball, T. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools: A case study of the version editor. *IEEE Transactions on Software Engineering*, 28(7):625–637, July 2002.

[2] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: The showdown. In *Symposium on Principles of Programming Languages*, 1998.

[3] V. Basili and F. Shull, Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[4] D. A. Christenson and S. T. Huang. Estimating the fault content of software using the fix-on-fix model. *Bell Labs Technical Journal*, 1(1):130–137, Summer 1996.

[5] F. Del Frate, P. Garg, A. Mathur, and A. Pasquini. On the correlation between code coverage and software reliability. In *International Conference on Software Reliability Engineering*, pages 124–132, 1995.

[6] P. Frankl and S. Weiss. An experimental comparison of the effectiveness of branch testing and data flow analysis. *IEEE TSE*, 19(8):774–787, 1993.

[7] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(2), 2000.

[8] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *International Conference on Software Engineering*, Portland, OR, USA, 2003.

[9] M. J. Harrold. Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, New York, NY, USA, 2000.

[10] M. J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Trans. on Software Engineering*, 27(3):248–263, March 2001.

[11] H. M., H. Foster, T. Goardia, and O. T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE'94*, pages 191–200, 1994.

[12] T. J. McCabe. A complexity measure. *IEEE Trans. on Software Engineering*, 2(4):308–320, Dec. 1976.

[13] A. Mockus and L. G. Votta. Identifying reasons for software change using historic databases. In *International Conference on Software Maintenance*, pages 120–130, San Jose, California, October 11-14 2000.

[14] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, April–June 2000.

[15] J. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32, 1993.

[16] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. on Software Engineering*, 22(12):886–894, December 1996.

[17] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, 1985.

[18] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. on Software Engineering*, 27(10):929–948, October 2001.

[19] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research Technical Report, 2001.

[20] T. J. Yu, V. Y. Shen, and H. E. Dunsmore. An analysis of several software defect models. *IEEE Trans. on Software Engineering*, 14(9):1261–1270, September 1988.

978-1-4244-4841-8/09/$25.00 ©2009 IEEE