Using Pre-Release Test Failures to Build Early Post-Release Defect Prediction Models

Kim Herzig
Microsoft Research
Cambridge, United Kingdom
kimh@microsoft.com

***Abstract*—Software quality is one of the most pressing concerns for nearly all software developing companies. At the same time, software companies also seek to shorten their release cycles to meet market demands while maintaining their product quality. Identifying problematic code areas becomes more and more important. Defect prediction models became popular in recent years and many different code and process metrics have been studied. There has been minimal effort relating test executions during development with defect likelihood. This is surprising as test executions capture the stability and quality of a program during the development process. This paper presents an exploratory study investigating whether test execution metrics, e.g. test failure bursts, can be used as software quality indicators and used to build pre- and post-release defects prediction models. We show that test metrics collected during Windows 8 development can be used to build pre- and post-release defect prediction models early in the development process of a software product. Test metrics outperform pre-release defect counts when predicting post-release defects.**

Keywords—measurement, software testing, development process, defect prediction

I. INTRODUCTION

Although product quality may still be the most important concern for most software companies, there is increasing pressure on development teams to deliver more features in shorter periods of time. The competition to gain or defend market share impacts development processes: developers are expected to work more efficiently while maintaining or increasing product quality. Thus, it becomes increasingly important to identify product issues early in the development process to ensure product quality without delaying product release. This situation is not new and there exists a large number of studies that investigate a wide variety of code and process metrics to estimate code quality before release [1,2,3,4,5,6,7]. A prominent intention of prediction models is to support and guide software testing: which code areas should be tested more thoroughly? However, releasing products in shorter periods of time also drives many product teams towards continuous integration and agile methods. Instead of dedicated testing milestones, code quality is ensured continuously. As a consequence, quality models should be usable early in the development process and should be capable of adapting themselves to code quality changes. However, static code properties do not significantly change their values when applying bug fixes. Both approaches may be too static to be continuously used during development. Constantly monitoring program behavior and health is needed: how does the program perform and how frequently does it fail to meet the specified functionality? Test cases check for such scenarios and report wrong, missing, or ambiguous program behavior. Collecting test execution results provides a history of program behavior during individual development periods.

The experiments described in this paper are designed to answer the following research questions:

- ***RQ1***) Do test execution metrics correlate with pre- or post-release defect counts for binaries and source code files?
- ***RQ2***) How effective are post-release defect prediction models based on test execution behavior metrics?
- ***RQ3***) Can test execution behavior metrics collected during a pre-release development milestone be used to predict pre-release defects for the later development milestones?
- ***RQ4)*** What test execution metric seem to have the highest defect prediction potential?
- ***RQ5)*** Do test execution metrics perform as proxy for post-release defects or do these metrics add additional value?

Existing effort to relate test behavior to code quality mainly focus on software reliability growth models (SRGMs), e.g. Musa et al. [7]. These models track and stochastically model the evolution of software quality. However, there exist indications that SGRMs may not fit in cases in which testing-effort is not constant [8] or if the software is constantly evolving [7]. In this paper, we do not investigate a stochastic model of possible quality evolution but rather explore the relationship (correlations) between long-term development test case execution behavior and code quality for evolving software systems. For this purpose, we mine and interpret system and integration test results collected during Windows 8 development. We further present a series of test execution metrics that summarize results and diversity of system and integration tests used to verify individual code changes. Mapping these code changes to source files and binaries, we are able to measure aggregated test execution behavior for individual source files and binaries. The results of this study show that test execution metrics can be used to derive pre- and post-release defect prediction models showing promising precision and recall values. Our results suggest that test metrics seem to outperform pre-release defect count measures when

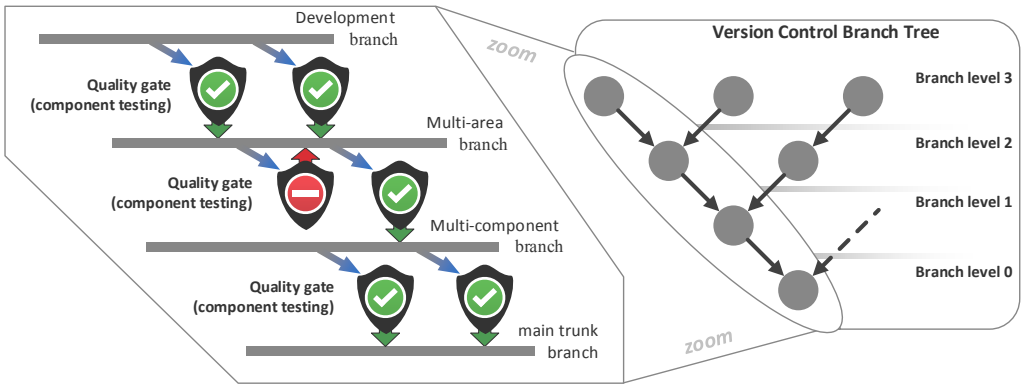

Fig. 1. Windows branching tree. Code changes have to pass quality gates to get integrated into lower level branches.

predicting post-release defects and that test bursts as well as the diversity of test failure scenarios seem to be important code quality indicators.

II. BACKGROUND

A. Branches

Development processes and branching structures for large software systems, such as Microsoft's Windows operating system, are complex and tend to be unique. For the sake of brevity, we provide a high-level description that contains enough information to make this paper self-contained. For details, we refer to Bird and Zimmermann [9]. Source code branches in Windows are organized in a tree structure with the stable main trunk branch as the root node (see Fig. 1). Branches are grouped by their branch level representing the distance of the branch to the main trunk branch (branch level zero). Branches at branch level one directly integrate into trunk while branches at branch level two integrate into branches at branch level one. Engineers commit code changes to development branches represented by leave nodes of branching tree allowing them to work in isolation. To integrate these code changes into trunk, changes must be pushed down the tree progressively merging code changes of different sub branches implemented in parallel. Once a code change is integrated into trunk, it is accepted and included in the next release of Windows.

B. Windows Quality Gates

Each edge in the Windows branching tree represents an integration path from one branch to a branch of the next lower level branch level. Each of these integration paths are guarded by quality gates, which are large system and integration test suites verifying that code changes meet the quality standards before they get further integrated into the Windows code base (see Fig. 1). If any of the tests executed by a quality gate fails, the scheduled integration will be cancelled and the failure reported for manual triage. Each quality gate executes multiple test cases and each test case can either pass or fail—an exception are test cases that timeout or are cancelled. In theory, every failing gate task hints to a code issue in the code base currently under test. However, the complexity of Windows and the complexity of the individual test cases themselves may cause test cases to fail due to test and infrastructure issues (we discuss this issue in more detail in Section III).

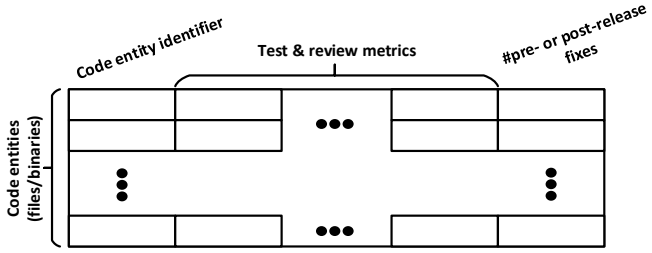

Fig. 2. Data collection used to predict post-release fixes for Windows 8 source files and binaries.

Quality gates execute test cases in multiple execution contexts: on different branches, different architectures (e.g. x86), different languages (e.g. en-us), and different build types (with and without debug symbols). The combination of these four factors (branch, architecture, build type, language) define the execution context of a test case. Test cases executed in different execution contexts must be handled separately.

Each code change submitted in one of the branch tree's leaf nodes has to pass through multiple integration branches before being integrated into the trunk branch. On its way down the branch tree, each code changes has to passed multiple quality gates—at least one at each branch level—and undergoes continuous system and integration testing.

III. DATA COLLECTION

The goal of this study is to investigate whether pre-release test metrics can be used to assess code quality during development as well as for released binaries (e.g. dll or exe files) and source files. The set of test execution metrics we collected can be grouped into five main groups: test case count metrics, quality gate count metrics, test case property metrics, test failure burst metrics, and code review metrics.

Before discussing these test metric groups in more detail, it is important to understand how the desired final data set will be organized. Fig. 2 illustrates the main structure of the data set that we will use to train and test various prediction models. Each row in this data matrix corresponds to one code entity (source file or code binary) identified by a primary key (e.g. file or binary name). The remaining columns of the data matrix correspond to individual metrics described in this section.

TABLE I. contains a short description of all test metrics collected and used during this study. A more detailed description of these metrics is given in the following subsections. Note that

TABLE I. TEST METRICS USED IN THIS STUDY. FOR EACH METRIC WE COLLECT ABSOLUTE AS WELL AS RELATIVE NUMBERS.

Name	Description
Test metrics counting individual test cases	
FPFailures	Number of individual test failures due to test and infrastructure issue; over all branches.
TPFailures	Number of individual test failures that reported at least one code issues; over all branches.
Test metrics counting quality gates	
FPGates	Number of quality gates that reported at least one false positive test case failure.
TPGates	Number of individual test failures that reported at least one code issues.
Test metrics counting failed test execution contexts	
TestSuites	Relative number of distinct test suites that reported at least one code issues; over all branches.
TestCases	Relative number of distinct test cases that reported at least one code issue.
Branches	Relative number of distinct branches on which at least one code issue was detected.
Architectures	Relative number of distinct architectures (e.g. *x86*, *amd64*) on which at least one test case reported at least one code issue.
BuildTypes	Relative number of distinct build types (*release* or *debug*) on which at least one test case detected at least one code issue.
Languages	Relative number of distinct languages (e.g. en-us) of test cases that reported at least one code issue.
Test failure burst metrics	
NumTPBursts	The number of code issue test failure bursts on the integration path into the main branch. Computed for all combinations of gap sizes and burst sizes between 1 and 3.
MaxTPBurst	The size of the largest test failure burst occurred on the integration path into the main branch. Computed for all combinations of gap sizes and burst sizes between 1 and 3.
NumFPBursts	Same as *NumTPBursts* but for false test alarms.
MaxFPBurst	Same as *MaxTPBurst* but for false test alarms.
Code review metrics	
Reviews	Number of distinct code changes that were code reviewed prior to check-in and quality gate testing.
Test metrics counting code changes	
BugChanges	Number of distinct code changes that modified the corresponding code entity and for which at least one test case reported at least one code issue. (no aggregation)

metrics described in this section are collected on different levels of granularity and aggregated to match the corresponding code entity granularity: source code files and code binaries. To investigate the predictive power of test execution behavior over time, we collect the metrics described in this section over all milestone periods of Windows 8.

A. Test Execution Metrics

Usually, testing an already tested code base is only necessary when new code changes have been applied to the code base. The goal is to investigate whether the applied code changes may have compromised code quality. Thus, test results can be associated with sets of code changes applied to the code base tested by the corresponding quality gates. To model these associations, we performed the following steps:

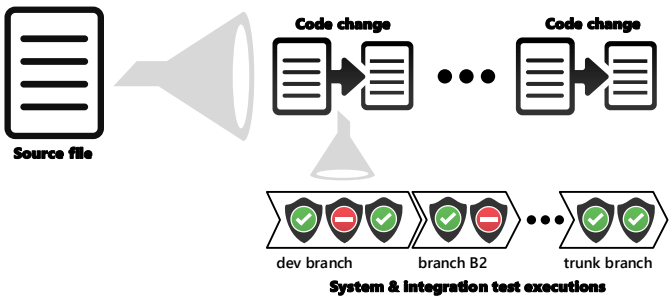

Fig. 3. Mapping test case execution results to code changes and their corresponding source code files. Source files can then be associated with code binaries.

Step 1: For each code change submitted to any code branch, we trace the code change's integration path through the branching tree—from the development branch to the main trunk branch. For each code change, we get a sequence of branches and timestamps that identify when and where the code change was applied. Using this sequence, we can identify quality gates that tested code bases modified by the code change across all branches (see Fig. 1).

Step 2: We then decompose quality gates into their individual test cases and associate execution context properties of the individual test case (e.g. architecture the test was executed on) and the result of each executed test case to the corresponding code change.

Step 3: Having test results associated with code changes, we decompose the code changes themselves to identify the code files they altered. Each file/binary modified by a code change will be associated with all corresponding test metrics. We aggregate test metrics of multiple code changes to single code entities using the following aggregation functions: sum, mean, median, max.

Fig. 3 visualizes this mapping process for source files. We used build information to identify source files that contribute to binaries. In the remainder of this section, we discuss individual test metrics we collected and the rationale behind them.

False Test Failures

Nearly all proposed test metrics count the number of failing test cases on different levels of granularity. However, as Fenton et al. point out: "[…] the results of the testing are likely only to be as trustworthy as the quality of the testing done" [10]. In order to assess the quality and reliability of test cases, we categorize test results into three different categories: (1) passing tests, (2) tests that fail but report a false alarm (false positive: FP), or (3) tests that fail and report code issues (true positive: TP). To separate false alarms from test failures due to code issues, we make use of bug reports linked against test failures and trace development activities triggered. If a test failure led to a bug report that was later fixed by applying a code change we mark the failure as a true positive (TP). Otherwise, the result of the failing test case is classified as false positive (FP). The main reason for FPs are test and infrastructure issues, e.g. a test requires a remote server to fetch test input, but the remote server could not be reached. FPs can have severe consequences as the failure requires manual inspection. Test cases reporting too

many FPs are considered as unreliable and are often treated as useless, although some failures might have unveiled real code issues. Thus, we explicitly modelled FP test failures in the below described set of metrics.

Counting individual test failures

The first group of test metrics (first section in TABLE I.) simply counts the number of failed test case executions. Note that one test case might be executed multiple times even within one quality gate. As discussed above, we count TP and FP test failure separately.

Counting failed quality gates

Counting individual test case failures does not reflect the number of distinct quality gates that caused this number of test cases to fail. Cases in which only few quality gates failed might indicate specific code issues detected only by the corresponding tests. Causing a broad list of quality gates to fail might indicate more severe and broad scale issues. Thus, we count the number of individual quality gates that failed at least once for code changes applied to the code entity. Note that this metric correlates with churn measures; e.g. number of applied code changes. The more frequently a file or binary is altered, the more test will be executed and the more test results can be associated with the code entities.

Failed execution context

As discussed in Section II.B, test cases are executed in different execution contexts (e.g. architecture or language) and we suspect the context in which a test case fails to be important. Thus, the third group of test metrics counts distinct execution contexts tests cases failed in. For example, we count the relative number of architectures a test case fails on—relative to the total number of architectures all test cases associated with the code artifact were executed on. The rationale behind counting the relative number of execution contexts (e.g. architectures) is to estimate the severity of test failures. Code issues detected only a small fraction of architectures are likely to be architecture specific and thus harder to replicate and detect when compared to code issues affecting all architectures. A value of one for these metrics refers to cases in which test suites for all possible execution contexts (e.g. architecture) failed at least once. See TABLE I. for all possible execution context measurements and a more detailed descriptions of these metrics.

Test Failure Bursts

So far, we counted the number of failures on different levels of granularity and across different execution context properties. However, none of these count metrics reflects possible dependencies between the observed test failures. Nagappan et al. [11] defined the concept of change bursts that captures the number of consecutive changes applied to code artifacts. The authors showed that change bursts are excellent indicators of code quality issues. Change bursts identify complex and hard code changes that had to be frequently revised. We adapt this basic concept of change bursts to testing and define the concept of test failure bursts analogue.

As discussed in the beginning of this section, we associate code changes to sequences of test executions performed on the integration path of the code change from its development branch into trunk. For each of these sequences, we can now count the number of subsequent test failures on this integration path. Since quality gate failures bring the integration process for the affected branch to a halt, these failures must be resolved immediately by applying a bug fix. Thus, a series of quality gate failures indicates incomplete bug fixes or a series of severe merge conflicts. For each code change we determine its test failure bursts as sequences of consecutive true positive quality gate failures. These test failure bursts are determined by two parameters:

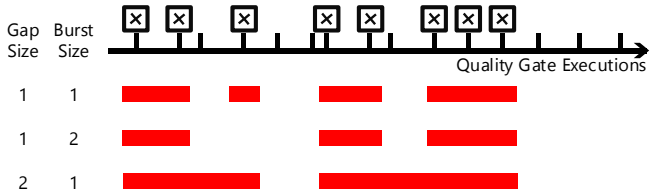

Fig. 4. How gap size and burst size determine test failure burst detection from a sequence of quality gate executions. Take from [11] and slightly modified.

Gap size. Shorter gaps between sequences of test failures are likely. Nagappan et al. [11] introduced the concept of gap size *G*, which determines the minimum distance between two true positive test failures. If two true positive test failures have a distance shorter than *G*, they will be part of the same burst. If we set the gap size to 1, then all directly consecutive test failures will be merged to bursts. Increasing gap size to two allows one passing test execution to "interrupt" failing test executions without causing a new burst (see last line in Fig. 4).

Burst size. The burst size *B* determines the minimal number of consecutive (with respect to the specified gap size) true positive test failures required in order to be counted as burst. If the number of true positive test failures in a burst is small than *B*, it will not be considered.

As an example consider Fig. 4. Assume we are investigating one code change and know the sequence of quality gate results for that particular code change (x-axis). Gate executions marked by a failure symbol ☒ reported at least one test failure. The number of test bursts for that code change depends on the two parameters gap size and burst size. In Fig. 4, red bars correspond to test failure bursts and depend on burst and gap size. Again, we count the number of test bursts for true positive test failures and false test alarms separately (see fourth section in TABLE I.). Further, we compute test bursts for all combinations of gap and burst sizes between one and three.

B. *Code Review Metrics*

In addition to test metrics described in Section III.A, we also include two measurements indicating whether code changes applied to code entities got manually reviewed or not. As a pre-check-in verification process, code reviews have shown to improve code quality and thus might influence the estimation of the severity of test failures encountered [6]. Code issues elapsing the code review but caught by test cases either determine the effectiveness of code reviews for these corresponding code changes, or simply hint to the fact that these defects were hard

to spot in code reviews and thus indicate tricky and complex code changes that might hold more still undetected code issues. For each source file or code binary, we report the absolute and relative number of code changes that were reviewed by at least one additional developer.

C. Pre- and Post-Release Defects

As quality measurement, we are using the number of pre- and post-release defects fixed in the corresponding source files and code binaries respectively. A post-release issue is an issue detected after releasing the corresponding software product to the public, but it does not state whether the code issue has been found by a customer or internally. Analogously pre-release defects are code issues reported and fixed during the regular development period. Post-release defects are of particular interest. These issues escaped quality assurance efforts and got exposed to customers. Nevertheless, pre-release defects can be of interest to fine tune development processes and to draw attention on code areas that seem to become or already are struggling with respect to code quality.

To identify post-release defect fixes, we counted the number of code changes applied in Windows 8 service pack branches. These branches serve as sink of defect fixes that will eventually be shipped to customers as part of a service pack or hot-fix. No feature development is permitted on these branches.

Identify pre-release defects is slightly more difficult. We used the CODEMINE [1] approach and tool developed at Microsoft and performed the following steps to identify pre-release defects and their affected source files and code binaries:

Step 1: We extract all bug reports filed against the product release under investigation. Only bug reports whose creation timestamps map to the development period of the corresponding product release and that were marked as closed and resolved or fixed were selected. We further associate bug reports with individual development milestones and will use this information for the experimental setup described in Section IV.C.

Step2: For all pre-filtered bug reports from step 1, we search for code change identifiers in bug reports and bug report references in code change commit messages. The result are pairs of bug reports and code changes for which at least one of the two pair elements references the respective other pair member.

Step 3: We remove all pairs for which we find no indication that the applied code change applies a fix for the code issue described in the corresponding bug report. For this purpose, we use key words (e.g. fix or patch) and common commit message templates. For more details, we refer to the detailed description of CODEMINE [1].

Thus, each pre-release defect is associated with a bug report filed against Windows 8 that got resolved as *fixed* and can be associated with at least one code change applied to the Windows 8 code base and that claims to fix the defect.

To associate defects (pre- and post-release) with source code entities, we identify all code entities altered by the defect fixing code change and count the number of distinct bug reports per code entity—we treat bug reports marked as duplicates as one bug report, thus counting only one bug report per group of bug report duplicates. When rolling metrics up to binary level, we excluded library files—files that get compiled into more than 100 binaries—and duplicated binaries—binaries with different names but with more than 90% equal content.

IV. Experimental Setup

In all experimental setups, we explicitly distinguish between two levels of code entity granularity: binaries and source code files. The reason is that many defect prediction models previously reported on Microsoft Windows were conducted on binary level. However, prediction models on that rather low level of granularity seem to be not actionable for development and product teams. Binaries combine hundreds of source code files and thousands of lines of code. Predicting defects for these large entities raises the question for more specific location as testing or reviewing code that is compiled into that binary is a massive challenge. Predicting defects for source files level is more challenging though. The defect density on this fine level of granularity is much lower and it is harder to identify those properties that actually correlate with defects.

A. Correlations between Test Metrics and Defects (RQ1)

To show basic relations between test behavior and defect counts, we computed spearman rank correlations between test execution metrics (Section III.A) and pre- and post-release defect counts (Section III.C). Correlation values lie between -1 and 1 and describes how well the dependency between two metrics can be described using a monotonic function. A correlation value of 1 or -1 occurs when one metrics is a perfect monotone function of the respectively other measurement. To conduct this experiment, we use metrics collected during the entire development period and correlated these metrics against the entire set of observed pre- and post-release bugs respectively.

B. Predicting Post-Release Defects (RQ2)

Rank correlations are good indicators of whether a metric might be a good predictor for a dependent variable. However, rank correlations do not allow to draw precise conclusions on how well a predictor based on multiple of these metrics will eventually perform. To investigate how well prediction models based test execution behavior metrics can predict post-release defects, we trained and tested actual prediction models. For both levels of granularity, binaries and source code files, we built classification models trained to separate entities that contained post-release defects from entities for which no post-release defects were recorded. We consider an entity defective if at least one post-release code fix altered the file.

To train individual prediction models, we use a data collection described in Section III that contains all described test behavior metrics collected over the entire development period of Windows 8 (independent variables) and the number of fixed post-release defects as dependent variable. We then split the overall collected data into two subsets: one used for training the other for testing purposes. We split the overall dataset into 2/3 training and 1/3 testing instances using stratified sampling—the ratio of code entities associated with post-release defects from the original dataset is preserved in both training and testing

TABLE II. **LIST OF MACHINE LEARNING MODELS TO TRAIN AND TEST POST-RELEASE DEFECT PREDICTION MODELS. TAKEN FROM [14].**

Model*	Description
Multinomial Logistic Regression (MLR)	This is a generalized linear model using a logic function and hence suited for binomial regression, i.e. where the outcome class is dichotomous.
Recursive Partitioning (RP)	A variant of decision trees, this model can be represented as a binomial tree and popularly used for classification tasks.
Support Vector Machine (SVM)	This model classifies data by determining a separator that distinguishes the data with the largest margin. We used the radial kernel for our experiments.
Tree Bagging (TB)	Another variant of decision trees, this model uses bootstrapping to stabilize the decision trees.
Random Forest (RF)	An ensemble of decision tree classifiers. Random forests grow multiple decision trees each "voting" for the class on an instance to be classified.
Naïve Bayes (NB)	Applying Bayes' theorem, this is a simple probabilistic classifier assuming strong independence.

*** For better understanding, we advise the reader to refer to specialized machine learning texts such as by Wittig and Frank [15].**

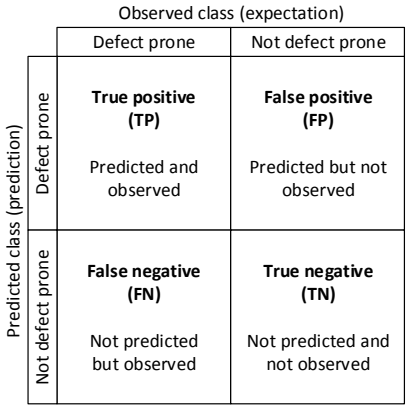

Fig. 5. Compraing observed and predictioned classes for code artifacts in a confusion matrix. Used to compute precision and recall values to measure accuracy of prediction algorithm.

subsets. Further, we repeatedly sampled the original data sets 100 times (100-fold cross-validation) using our splitting scheme in order to generate 100 independent training and testing subsets. We conducted the experiments using the R statistical software [12] (version 3.10) and more precisely Max Kuhn's R package caret [13] to train, tune, and test a set of six different machine learning models described in more detail in TABLE II. Each model is optimized by the caret package optimizing various tune parameters (please see caret manual for more details). The level of performed optimization can be set using the *tuneLength* parameter, which is set to five for all experiments reported in this paper.

We further removed constant metric and highly inter-correlated metrics columns, centered and rescaled the data values, before applying principal component analysis (PCA). Using PCA, we selected principal components that accounted for 95% of variance.

To compare the actual observed and predicted classes for individual code artifacts categorized each predicted value into four individual categories as shown in Fig. 5. As evaluation measure we report precision and recall where precision and recall are defined as:

$$precision = TP \div (TP + FP)$$
$$recall = TP \div (TP + FN).$$

Each of these measures is a value between zero and one. A precision of one indicated that the classification model does not report a single false positive—that is classifying a non-defective code entity as defective. A recall of one would imply that the classification model does not report any false negatives.

C. *Predicting Pre-Release Defects for Development Milestones (RQ3)*

To answer research question RQ3, we use the basic experimental setup as described in Section IV.B, but choose different time frames for dependent and independent variables. The goal is to investigate whether we can use test execution to predict code issues for the respectively next development milestone. Windows 8 was developed in three development milestones: M1, M2, and M3. In this experimental setup we will use test execution behavior collects during one of the three development milestones to predict code issues for the respectively consecutive milestone. More precisely, we used the following combinations of dependent and independent variables:

- *M1 predicts M2*: Test metrics collected during M1 to predict code issues reported in M2.
- *M2 predicts M3*: Test metrics collected during M1 to predict code issues reported in M2.
- *M1+M2 predicts M3*: Test metrics collected during the two consecutive milestones M1 and M2 to predict code issues reported in M3.

Each of these three scenarios is an independent experiment. Thus, we built prediction models for binary and file level and perform a 100-cross fold prediction using repeated stratified sampling (as described in Section IV.B). In total we trained and tested 600 independent prediction models.

D. *Metric Importance (RQ4)*

We investigated which of our test execution behavior metrics seems to be most suited to predict pre- and post-release code defects respectively. We used the *filterVarImpl* function of the caret package [13]. This function uses so-called ROC curves to determine the importance of a metric. Receiver operating characteristic (ROC) curves are graphical plots of illustrating the performance of a binary classification model. The curve is

TABLE III. **TOP 10 SPEARMAN RANK CORRELATIONS BETWEEN TEST METRICS AND POST-RELEASE DEFECT COUNTS FOR CODE BINARIES.**

Metric	Aggregation	Correlation value
FPBurstsB1G2	Max	0.44
FPGates	Sum	0.44
TestSuites	Sum	0.44
Architectures	Sum	0.37
TPBurstsB1G2	Max	0.37
TestCases	Sum	0.37
TPGates	Sum	0.36
TestCases	Max	0.33
Architectures	Max	0.29
Branches	Max	0.26

TABLE IV. **TOP 10 SPEARMAN RANK CORRELATIONS BETWEEN TEST METRICS AND POST-RELEASE DEFECT COUNTS FOR SOURCE FILES.**

Metric	Aggregation	Correlation value
FPGates	Sum	0.16
FPBurstsB1G2	Max	0.16
TestSuites	Sum	0.16
Branches	Max	0.16
TPBGurstsB1G2	Max	0.16
TestCases	Sum	0.16
Architectures	Sum	0.16
Languages	Sum	0.15
Architectures	Max	0.15
FPGates	Max	0.13

created by plotting recall against the FP rate, where FP rate is defined as $FP \div (FP + TN)$. The area under the ROC curve equals the probability that the classifier ranks a randomly chosen positive instance higher than a randomly chosen negative one. The larger the area under an ROC curve, the more accurate the corresponding binary classifier is considered. The function *filterVarImpl* conducts a series of ROC curve analysis for each metric: "a series of cutoffs is applied to the predictor data to predict the class. The sensitivity and specificity are computed for each cutoff and the ROC curve is computed. The trapezoidal rule is used to compute the area under the ROC curve. This area is used as the measure of variable importance." [13]. Please note that the metric importance for classification models may not match the spearman rank correlation results. While the rank correlation considers the exact order of entities, classification models separate entities into two categories. The suitability of a metric to solve either problem may be different.

E. *Are Test Execution Metrics Proxies for Pre-Release Defects (RQ5)*

Finally, we want to find out whether test execution metrics add any value when compared to pre-release defects. The key issue is that test execution failures tend to cause pre-release bug reports, except for false alarms. Pre-release defects can be interpreted as a set of test failures subject to human judgment whether the failure is a false alarm or not and thus might suffer less from data noise. On the other hand, test execution failures are available earlier as reliable number of pre-release defect counts and thus might be more valuable than simple defect counts. To answer RQ5, we compare prediction models based on test execution metrics with a base-line prediction model using pre-release defects only. The baseline model uses simply the number of pre-release defects collected during all development milestones as predicted value of post-release defects. We compare the area under the ROC curve (see Section IV.D) from this baseline regression model with the area under ROC for predicted values using test execution metrics. For this experiment we used the R-package *pROC* [16] and provide the absolute values of the area under the ROC using the recursive partitioning (*rpart)* model as well as the result of a statistical significance test (DeLong's test for two ROC curves) provided by the *pROC* package. The reason to use the recursive partitioning model (*rpart*) in this experiment stems from the observation that this model performed particularly well in all previous experiments.

V. EXPERIMENTAL RESULTS

In this section, we discuss the results of all experimental setups described in Section IV.

A. *Correlations between Test Metrics and Defects (RQ1)*

As described in Section IV.A we use Spearman rank correlations to show basic dependencies between post-release bug counts and our test metrics. TABLE III. contains these rank correlation values on the binary level of granularity. Naturally, many of these test metrics are highly correlated with churn, e.g. not altering a file implies that no code change altering this file could have encountered any test failure. For the sake of brevity, TABLE III. contains the top 10 highest correlation values for metric groups as described in TABLE I. and the aggregation function that achieved that highest correlation value.

False test alarms seem to correlate most strongly with post-release defect counts. This trend confirms that false test alarms hurt the quality assurance process as they decrease the confidence in test results and the overall testing process in general—a concern that we confirmed with the Windows product teams. In fact false test alarms seem to be one of the most prominent testing issues in many development processes. We expect that results from test cases that have a low reliability reputation will be inspected less thoroughly, which leads to the paradox effect that bugs might escape the testing process although they got caught and reported by test failures. Furthermore, the results show that false test alarms seem to dominate also the number of test failures reporting real code issues. Interestingly, the number of architectures on which test failures occur is among the 10 most influential metrics. We suspect that the number of architectures reflects code issue severity. If a code issue is caught on multiple architectures, these changes may be more severe than code issues detected for ARM processors only. As expected, correlation values between test metrics in our test metrics suite and post-release defect counts are much lower (see TABLE IV.). This is usually given by the fact that there exist only few source files that have at least one post-release. Nevertheless, we see on file level the same trends

TABLE V. **OVERALL DEFECT PREDICTION MODEL ACCURACY USING DIFFERENT SOFTWARE MEASURES ON WINDOWS VISTA. CONTENT TAKEN FROM [17] AND [11].**

Model	Precision	Recall
Change bursts [11]	0.91	0.92
Organizational structure [5]	0.86	0.84
Code churn [18]	0.79	0.80
Code complexity [19]	0.79	0.66
Social network measures [20]	0.77	0.71
Code dependencies [21]	0.75	0.69
Test coverage [22]	0.84	0.54
Pre-release defects	0.74	0.63
Pre-release test failures (this study)	0.81	0.70

TABLE VI. **PRECISION AND RECALL VALUES FOR PREDICTION MODELS TRAINED ON ALL DEVELOPMENT PERIODS PREDICTING POST-RELEASE DEFECTS, BOTH ON BINARY AND SOURCE FILE LEVEL.**

Model	Binary level		File level	
	Precision	Recall	Precision	Recall
MLR	0.79	0.30	0.62	0.16
NB	0.38	0.51	0.29	0.19
RF	0.81	0.70	0.65	0.24
RP	0.72	0.33	0.57	0.19
SVM	0.74	0.38	0.64	0.12
TP	0.75	0.51	0.63	0.21

as discussed on binary level. False alarms seem to be an issue, as well as the diversity of execution contexts failures occur on, e.g. the more distinct test suites detected the issue, the higher the number of post-release defects.

All reported metric values are statistically significant. We used the *cor.test* function in R to conduct a Spearman's rho statistic to estimate a rank-based measure of association. For all reported metrics, the test reported p-values below 1.00E-05.

Test metrics are correlated with each other. Especially test failure burst metrics show high correlation values often above 0.7. This is also true for absolute numbers of test case, quality gate failures and TP and FP metrics values. This was expected as more quality gates imply more test cases, as test burst measurements are by definition strongly correlated, and since TP failures can cause FP failures. In contrast, the relative numbers of test failures and execution contexts are only moderately correlated among each other: correlation values between 0.2 and 0.4. As described in the experimental setup, we performed principal component analysis for our prediction models. Thus, inter-metric correlation does not influence our prediction model results.

B. Predicting Post-Release Defects (RQ2)

In this section, we discuss the experimental results showing how well the test metrics suite performs with respect to predicting post-release defects. In the first experiment, we used test metrics collected during all three development milestones M1, M2, and M3 to predict post-release defects counts. Highest precision and recall values for the best machine learning model is shown in TABLE V. , along with previous prediction models

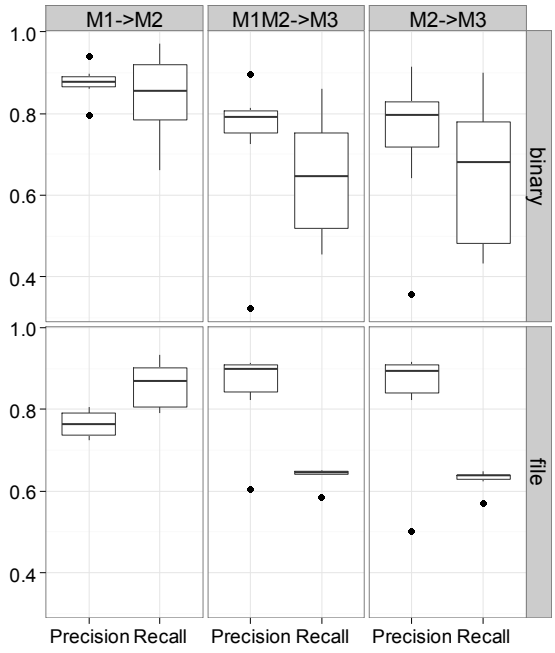

Fig. 6. Precision and recall values for pre-release models on binary and file level.

reported on earlier versions of Windows. The best model in our experimental setup showed a precision value around 0.81 and a recall value of 0.7. All precision and recall values for this experiment are shown in TABLE VI. Median precision over all models lies at 0.75, the median recall lies at 0.44. Compared to prediction models based on other code and process metrics, out test metrics show the third highest precision value (out of nine models) and the fourth highest recall value—only organizational structure and change bursts show stronger combinations of precision and recall values. Importantly, test metrics perform significantly better than models trained and tested on pre-release defects or churn. As expected, prediction models on file level perform worse than models on binary level. The median precision for these models lies at 0.63 and their median recall at 0.19. Recall values on file level are too low to allow trustable applications. Nevertheless, source files are the preferable level of granularity as prediction models for binaries are hardly actionable.

C. Predicting Pre-Release Defects for Development Milestones (RQ3)

In RQ3 we want to investigate whether we can predict pre-release issues for individual milestones using test metrics collected during the respectively previous milestone(s), e.g. predicting defects for M2 using test metrics collected during M1 (M1→M2). Fig. 6 shows precision and recall values for models predicting pre-release defect counts for M2 and M3 on binary and file level. On binary level, precision values are high and lie around 0.8. Recall lies between 0.5 and 0.8, depending on the milestone predicted. Results for the same experiment on source file level of granularity shown strong precision values above 0.7 for models trained on milestone M1 predicting M2. However, precision increases to 0.9 for models predicting pre-release defects found in M3. For these models, we also observed very

TABLE VII. **TOP 10 MOST IMPORTANCE METRICS FOR MODELS PREDICTING *POST*-RELEASE DEFECTS ORDERED BY IMPORTANCE.**

Binary level		File level	
Metric	Area under ROC	Metric	Area under ROC
FPBurstsB1G1	0.86	**FPGates**	0.73
FPGates	0.85	**FPBurstsB1G1**	0.73
TestSuites	0.85	**TestCases**	0.73
Architectures	0.79	**FPBurstsB2G3**	0.72
TPBurstsB1G3	0.78	**Branches**	0.72
TPGates	0.78	**TPBurstsB1G2**	0.72
BuildTypes	0.77	**Architectures**	0.72
Branches	0.77	**BuildTypes**	0.70
Languages	0.76	**Languages**	0.69
FPBurstsB2G3	0.75	**FPBurstsB2G1**	0.67

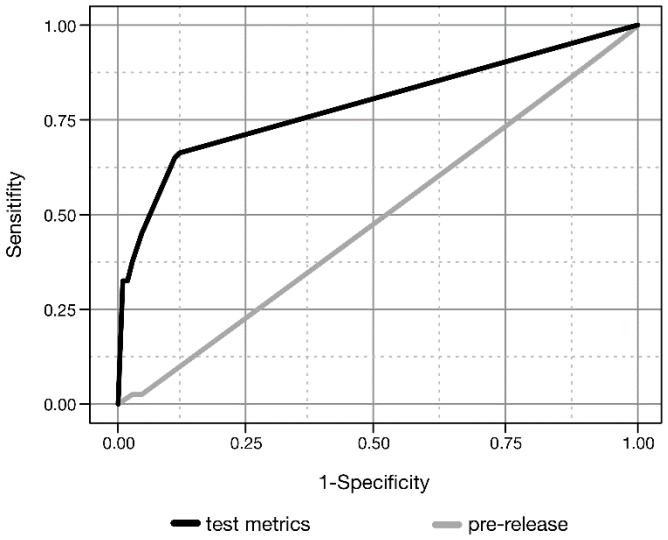

Fig. 7. ROC curves for regression models using pre-release defects and test execution metrics to predict post-release defects.

stable recall values with minimal variations between cross-folds and learners. In general, recall values on file level are low (around 0.65). The overall fraction of source files having post- and pre-release defects is small. This affect the accuracy of many machine learning algorithms. Furthermore, it seems clear that test effort and test execution are not the only factors influencing code issues. Other influencing factors are security issues (mostly found by other test cases or manual inspections), specification issues, or compatibility issues—none these factors is adequately covered by our test suite metrics and thus cannot be explained by models using only these metrics. Comparing file and binary models, it emerges that precision value trends for both levels of granularity have opposite trends. While M1→M2 models show the strongest precision values on binary level models for the same experiment on file level show lowest precision values. Another interesting observation is that models predicting milestone M3 trained on M2 seem to perform as good or even better when compared to models predicting M3 but trained on M1 and M2. Our interpretation is that milestones M2 and M3 are similar wheras M1 contains many activities with respect to code consolidation, refactorings, and new core features. This trend is consitent over binary and file level.

D. Metric importance (RQ4)

Although models on file and binary level show different prediction performance, they show similar metric importance. The number of quality gates that raise false alarms and the number of false alarm bursts with burst and gap size of one are the two most important metrics, as shown in TABLE VII. Interestingly, the number of architectures, build types, and languages are among the top 10 most important metrics for both levels of granularity. Overall, test bursts and test execution context signals, such as branches and architectures, seem to show great value for predicting post-release defects.

E. Are Test Execution Metrics Proxies for Pre-Release Defects (RQ5)

ROC curves for both regression models, using pre-release defect counts and test metrics, are shown in Fig. 7. The base line model using pre-release defects shows an area under the ROC of 0.48 while test metric models show a statistical significant higher value of 0.79—DeLong's test for two ROC curves yields an p-value of under 2.2E-16. The difference between both ROC curves is higher than expected. In the past pre-release defects showed strong post-release prediction performances, but at least for Windows 8, test execution metrics outperform this baseline regression model.

VI. RELATED WORK

There have been a wide variety earlier studies investigating various code and process metrics for defect prediction purposes, including studies showing that testing and test effort are related to product quality. However, we are not aware of previous studies studying false test alarm and test failure bursts and their suitability to predict pre- and post-release bugs for evolving software products.

A. Defect Prediction

The number of related studies on defect prediction models is large. For the sake of brevity, we only refer to the most relevant related studies in this section. The first studies on predicting defects using code metrics emerged in the 1990s. In 1999, Fenton and Neil [23] provided a comprehensive overview of defect prediction models at that time. In their paper, the authors mention a couple of studies that used software test metrics to predict code defects—but until then, most studies concentrated on testability and the impact of testability measurements on code quality rather than using real test results (e.g. [10,24,25]). In recent years, more reviews on defect prediction models are emerging (e.g. [26,27]). These reviews show the wide variety of aspects and measurements used for defect prediction purposes. Ostrand et al. [28] used code metrics and prior faults to predict the number of faults. Zimmermann et al. [29] related code complexity to defects. Other studies used change-related metrics [30], developer related metrics [31], organizational metrics [5], process metrics [32], or change dependency metrics [33,34,2] to build defect prediction models, on various software systems and levels of granularity.

B. Impact of Tests on Code Quality

Nagappan et al. [35] and lately Rafique and Misic [36] showed that the effort invested into testing has a significant effect on code quality. Other studies used code coverage [37,38,39] and test count metrics such as the number of assertions [35] suggesting that test effort and test-driven development are likely to increase product quality [39,40].

C. Test Effort Measusres

Wu et al. [41] used testing effort measures (e.g., complexity of test cases) and testing effectiveness measures (e.g., number of test failures) to assess software reliability. Similar, Mende and Koschke [4] showed that effort aware prediction models perform significantly better than models based on count metrics. However, these effort metrics are measuring the human effort that was invested to write or test the system. In this paper, we investigate test execution results rather than effort estimations.

D. Test Coverage

Chen and Wong presented a technique "using both time and code coverage measures for the prediction of software failures" [38] and showed that test coverage and the time between "test cases, which neither increases code coverage nor causes a failure" [38] can be used to reduce overestimations of reliability. Later, Cai and Lyu [37] confirmed that test coverage impact code quality. However, none of these studies analyzed test execution results and their predictive power.

E. Test Execution Data & Reliability Growth Models

Musa et al. [7] were among the first to relate test execution data with software reliability using software reliability growth models (SRGMs). The idea behind these models when designed was to use a stochastic model to assess the evolution of software in its successive testing phases. Since then, SGRMs based on test execution data evolved showing high accuracies [8,42,43,44]. However, SGRMs assume that the program being executed is stable (not changing except for bug fixes) [7]. In this study, we do rely on the stability of the program nor do we assess or predict the evolution of the software project. The purpose of this paper is to determine the suitability of test metrics to predict pre- and post-release defects for an evolving software project.

Elberzhager et al. [45] used test failure inspection metrics to prediction defect prone areas. However, the authors used only pre-release defects as independent variable. In this study, we use more diverse independent variables including false test alarms and test failure bursts.

VII. Threats To Validity

A. Study Subject

Test metrics were not available for Windows releases before Windows 8 and Windows 8.1 was only recently released leaving us without a reliable estimation of post-release defects. Based on one release, we do not claim our results as general truth. Further studies are needed to confirm the suitability of our models.

B. Test Metrics

For the interpretation of test execution results, whether a failure is a true or false positive, is partially based on heuristics. Although, we ensured the correctness and rationales behind these heuristics with the Windows product teams, we and the Windows team estimate a test result interpretation error rate of 10%. This error rate may impact prediction results.

C. Number of Defects

To compute pre- and post-release defect counts we used datasets provided by CODEMINE [1]. Although CODEMINE is extensively used and its results frequently verified, bug counts for some files may remain approximations.

D. Predicting Pre-Release Defects

Models predicting pre-release defects tend to predict churned files (usual suspects), e.g. files that are heavily churned due to new functionality. We did not compare against a usual suspect model. The presented results remain valid. However, there might exist simpler models achieving similar results.

VIII. Conclusion

Although the primary goal of tests is to detect code issues, there has been surprisingly little work to use test results to predict pre- and post-release defects for evolving software systems. Software reliability growth models (SRGMs) make extensive use of test execution data but assume stable development stages [7]. In this study, we defined a set of test execution metrics, separating false test alarms and failures reporting code issues, and test failure burst metrics to predict pre- and post-release bugs on binary and source file level for Windows 8. The results presented in this paper are promising. Prediction models trained on test metrics showed relatively high precision and recall values when compared to previous prediction models reported for Windows. The fact that these models are also precise on the finer and actionable source file level is encouraging. We also showed that test metrics can be used to predict bugs between individual development milestones rather than between individual releases and that test metrics are more expressive than simple prior bug counts. An initial analysis on the importance of individual metrics showed that test failure burst, false test alarms, and metrics counting different execution context properties are among the most influential metrics. However, this study should be considered as an explorative study assessing the general suitability of test failure metrics for defect prediction models. More work is needed to confirm these findings and to consolidate the current set of correlated test metrics and to find a more condense and optimal set. It also remains to be demonstrated that test metrics are good defect predictors in general. Due to confidentiality reasons, we are not able to share the raw datasets used in this study. Replication studies need to re-implement metrics. However, we expect these test metrics to depend on the individual test processes and infrastructures of the corresponding projects. Adapting or at least confirming the metrics underlying assumptions will be indispensable.

Acknowledgment

We thank the Windows development and BVT quality team for their support and feedback. This work is based on data extracted from varies development repositories provided by the Microsoft TSE group. Our special thanks go to Michaela Greiler, Jacek Czerwonka, Brendan Murphy, Christopher Theisen, Jason Means, and Poornima Priyadarshini.

REFERENCES

[1] Czerwonka, J., Nagappan, N., Schulte, W., and Murphy, B. CODEMINE: Building a Software Development Data Analytics Platform at Microsoft. *Software, IEEE*, 30, 4 (2013), 64--71.

[2] Herzig, K. and Zeller, A. Mining cause-effect-chains from version histories. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on* (2011), 60--69.

[3] Herzig, K.S. Capturing the long-term impact of changes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2* (2010), 393--396.

[4] Mende, T. and Koschke, R. Effort-Aware Defect Prediction Models. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on* (March 2010), 107-116.

[5] Nagappan, N., Murphy, B., and Basili, V. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Proceedings of the 30th International Conference on Software Engineering* (2008), ACM, 521--530.

[6] McIntosh, S., Kamei, Y., Adams, B., and Hassan, A.E. The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects. In *In Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)* (Hyderabad (India), 2014).

[7] Musa, J.D., Iannino, A., and Okumoto, K. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, Inc., 1987.

[8] Huang, C.-Y., Kuo, S.-Y., and Lyu, M.R. An Assessment of Testing-Effort Dependent Software Reliability Growth Models. *Reliability, IEEE Transactions on*, 56 (June 2007), 198-211.

[9] Bird, C. and Zimmermann, T. Assessing the Value of Branches with What-if Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina, 2012), ACM, 45:1--45:11.

[10] Fenton, N., Krause, P., and Neil, M. Software measurement: uncertainty and causal modeling. *Software, IEEE*, 19 (Jul 2002), 116-122.

[11] Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., and Murphy, B. Change Bursts as Defect Predictors. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering* (November 2010).

[12] Team, R.D.C. *R: A Language and Environment for Statistical Computing*. , 2010. R Foundation for Statistical Computing.

[13] Kuhn, M. *caret: Classification and Regression Training*. , 2011.

[14] Herzig, K. *Mining and Untangling Change Genealogies*. , 2012. Universität des Saarlandes.

[15] Witten, I.H. and Frank, E. Data mining: practical machine learning tools and techniques with Java implementations. *SIGMOD Rec.*, 31 (mar 2002), 76--77.

[16] Robin, X., Turck, N., Hainard, A., Tiberti, N., Lisacek, F., Sanchez, J.-C., and Muller, M. pROC: an open-source package for R and S+ to analyze and compare ROC curves. *BMC Bioinformatics*, 12 (2011), 77.

[17] Nagappan, N. and Ball, T. Evidence-Based Failure Prediction. O'Reilly Media, 2010.

[18] Nagappan, N. and Ball, T. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th International Conference on Software Engineering* (2005), ACM, 284--292.

[19] McCabe, T.J. A Complexity Measure. *IEEE Trans. Software Eng.*, 2 (1976), 308-320.

[20] Bird, C., Nagappan, N., Gall, H., Murphy, B., and Devanbu, P. Putting It All Together: Using Socio-technical Networks to Predict Failures. In *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering* (2009), IEEE Computer Society, 109--119.

[21] Zimmermann, T. and Nagappan, N. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering* (2008), ACM, 531--540.

[22] Mockus, A., Nagappan, N., and Dinh-Trong, T.T. Test coverage and post-verification defects: A multiple case study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement* (2009), IEEE Computer Society, 291--301.

[23] Fenton, N.E. and Neil, M. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25 (Sep 1999), 675-689.

[24] Voas, J.M. and Miller, K.W. Software testability: the new verification. *Software, IEEE*, 12 (May 1995), 17-28.

[25] Badri, M. and Toure, F. Evaluating the effect of control flow on the unit testing effort of classes: An empirical analysis. *Advances in Software Engineering*, 2012 (2012), 5.

[26] Catal, C. and Diri, B. Review: A Systematic Review of Software Fault Prediction Studies. *Expert Syst. Appl.*, 36 (may 2009), 7346--7354.

[27] Radjenoviç, D., Heričko, M., Torkar, R., and Živkovič, A. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55 (2013), 1397--1418.

[28] Ostrand, T.J., Weyuker, E.J., and Bell, R.M. Where the bugs are. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis* (2004), ACM, 86--96.

[29] Zimmermann, T., Premraj, R., and Zeller, A. Predicting Defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering* (2007), IEEE Computer Society, 9--.

[30] Moser, R., Pedrycz, W., and Succi, G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering* (2008), ACM, 181--190.

[31] Pinzger, M., Nagappan, N., and Murphy, B. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (2008), ACM, 2--12.

[32] Hassan, A.E. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering* (2009), IEEE Computer Society, 78--88.

[33] Zimmermann, T. and Nagappan, N. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering* (2008), ACM, 531--540.

[34] Herzig, K., Just, S., Rau, A., and Zeller, A. Predicting Defects Using Change Genealogies. In *Proceedings of the 2013 IEEE 24nd International Symposium on Software Reliability Engineering* (2013), IEEE Computer Society.

[35] Nagappan, N., Williams, L., Vouk, M., and Osborne, J. Early Estimation of Software Quality Using In-process Testing Metrics: A Controlled Case Study. *SIGSOFT Softw. Eng. Notes*, 30 (may 2005), 1--7.

[36] Rafique, Y. and Misic, V.B. The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis. *Software Engineering, IEEE Transactions on*, 39 (June 2013), 835-856.

[37] Cai, X. and Lyu, M.R. Software Reliability Modeling with Test Coverage: Experimentation and Measurement with A Fault-Tolerant Software Project. In *Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on* (Nov 2007), 17-26.

[38] Chen, M.-H., Lyu, M.R., and Wong, W.E. Effect of code coverage on software reliability measurement. *Reliability, IEEE Transactions on*, 50 (Jun 2001), 165-170.

[39] Nagappan, N., Maximilien, E.M., Bhat, T., and Williams, L. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, 13, 289-302.

[40] Strecker, J. and Memon, A.M. Accounting for Defect Characteristics in Evaluations of Testing Techniques. *ACM Trans. Softw. Eng. Methodol.*, 21 (jul 2012), 17:1--17:43.

[41] Wu, J., Ali, S., Yue, T., and Tian, J. Experience report: Assessing the reliability of an industrial avionics software: Results, insights and recommendations. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on* (Nov 2013), 218-227.

[42] Okumoto, K. Software defect prediction based on stability test data. In *Quality, Reliability, Risk, Maintenance, and Safety Engineering (ICQR2MSE), 2011 International Conference on* (June 2011), 385-387.

[43] Khatri, S.K., Kumar, D., Dwivedi, A., and Mrinal, N. Software Reliability Growth Model with testing effort using learning function. In *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on* (Sept 2012), 1-5.

[44] Rafi, S.M. and Akthar, S. Incorporating fault dependent correction delay in SRGM with testing effort and release policy analysis. In *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on* (Sept 2012), 1-6.

[45] Elberzhager, F., Kremer, S., Munch, J., and Assmann, D. Guiding Testing Activities by Predicting Defect-Prone Parts Using Product and Inspection Metrics. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on* (Sept 2012), 406-413.

[46] McIntosh, S., Kamei, Y., Adams, B., and Hassan, A.E. The Impact of Code Review Coverage and Code Review. In *11th Working Conference on Mining Software Repositories (MSR)* (Hyderabad, 2014), ACM.