Assessing the Cost Effectiveness of Fault Prediction in Acceptance Testing

Akito Monden, *Member*, *IEEE*, Takuma Hayashi, Shoji Shinoda, Kumiko Shirai, Junichi Yoshida, Mike Barker, *Member*, *IEEE*, and Kenichi Matsumoto, *Senior Member*, *IEEE*

Abstract—Until now, various techniques for predicting fault-prone modules have been proposed and evaluated in terms of their prediction performance; however, their actual contribution to business objectives such as quality improvement and cost reduction has rarely been assessed. This paper proposes using a simulation model of software testing to assess the cost effectiveness of test effort allocation strategies based on fault prediction results. The simulation model estimates the number of discoverable faults with respect to the given test resources, the resource allocation strategy, a set of modules to be tested, and the fault prediction results. In a case study applying fault prediction of a small system to acceptance testing in the telecommunication industry, results from our simulation model showed that the best strategy was to let the test effort be proportional to "the number of expected faults in a module $\times$ log(module size)." By using this strategy with our best fault prediction model, the test effort could be reduced by 25 percent while still detecting as many faults as were normally discovered in testing, although the company required about 6 percent of the test effort for metrics collection, data cleansing, and modeling. The simulation results also indicate that the lower bound of acceptable prediction accuracy is around 0.78 in terms of an effort-aware measure, $Norm(P_{opt})$. The results indicate that reduction of the test effort can be achieved by fault prediction only if the appropriate test strategy is employed with high enough fault prediction accuracy. Based on these preliminary results, we expect further research to assess their general validity with larger systems.

Index Terms—Complexity measures, fault prediction, quality assurance, resource allocation, simulation

1 Introduction

As recent software systems have grown in size and complexity, quality assurance activities such as testing and inspection have become increasingly important, not only for software developers, but also for software purchasers who are responsible for acceptance testing and/or software service deployment. Since resources are limited and scheduling is tight in most cases, quality assurance must be performed as efficiently as possible.

To prioritize quality assurance efforts, techniques for predicting fault-prone modules have been proposed to select software modules by their probability of having a fault [16], the number of expected faults [11], [17], or the fault density [14]. Based on the prediction results, practitioners can allocate limited testing (or inspection) efforts to fault-prone modules so as to find more faults with smaller effort.

However, while the prediction performances of those techniques have been evaluated in terms of recall/precision/F1-measure [12], [26], Alberg diagrams [23], and/or ROC curves [16], the final goal of reducing the test effort or increasing software quality has been rarely explored. To adopt fault prediction techniques in industry, one needs to be able to assess the cost effectiveness of the prediction because not only poor predictions but also a poor resource allocation strategy could even increase the test effort. Moreover, one needs to consider that metrics collection, data cleansing, and modeling themselves require a significant cost.

In this paper, from the point of view of the software purchaser-side organization, we set our primary goal to *estimate the reduction of acceptance test effort* that fault prediction can achieve. To achieve this goal, our study aimed to answer the following research questions:

(RQ1) What is the appropriate strategy to allocate test effort to each module after prediction?

This question is difficult to answer although several strategies can be easily developed. The simplest one is to let the test effort be proportional to the number of expected faults in a module [17]. The concern with this strategy is that it does not consider the size of a module, which may affect the ease of discovering a fault. Another strategy is to allocate test effort based on the fault density, but this may also not be the best choice because it concentrates on modules with high fault density no matter what their size, even though larger modules will contain more faults than smaller ones. Moreover, the best strategy may depend on the available test effort. For example, if we have plenty of resources, applying equal effort to all modules might be fine.

To answer this question, we need to be able to compute the expected number of discoverable faults with respect to the given test resources, the resource allocation strategy (with fault prediction result), and the set of modules to be tested. Then, we could estimate the test effort needed to discover a desired number of faults.

- *A. Monden, M. Barker, and K. Matsumoto are with the Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5 Takayama, Ikoma, Nara 630-0192, Japan. E-mail: {akito-m, mbarker, matumoto}@is.naist.jp.*
- *T. Hayashi, S. Shinoda, K. Shirai, and J. Yoshida are with the R&D Center, NTT West Corporation, 1-2-31 Sonezaki, Kita-ku, Osaka 530-0057, Japan. E-mail: {k.shirai, j.yoshida}@rdc.west.ntt.co.jp.*

Manuscript received 3 July 2012; revised 30 Jan. 2013; accepted 5 Apr. 2013; published online 11 Apr. 2013.
Recommended for acceptance by T. Menzies.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2012-07-0175.
Digital Object Identifier no. 10.1109/TSE.2013.21.

In this paper, we propose a fault discovery model that can represent the relationship among the prediction results, test efforts, module sizes, and the probability of fault discovery. By using this model, we simulated testing to compare the effectiveness of resource allocation strategies.

(RQ2) What is the required level of prediction accuracy?

If the prediction accuracy is very low, we cannot rely on any test effort allocation strategy that uses the prediction result. This paper tries to find the lower bound of the required accuracy to distinguish between resource allocation strategies.

(RQ3) How much is test effort reduced by the prediction and how much effort is needed to conduct the prediction?

This is the primary question companies want answered. To answer this question, we assume that the tests currently conducted by the company are not complete, i.e., there are still some faults remaining after testing because most software systems contain faults after release [7].

Therefore, we define a parameter called the remaining fault rate in Section 4.3, and compute a required test effort that potentially discovers as many faults as actual testing through the simulation.

We also need to measure the effort needed for metrics collection, data cleansing, and modeling to assess the cost effectiveness of prediction.

The next section introduces our project context, including motivation and related work. In Section 3, we explain how we built the fault prediction models. Section 4 proposes our assessment method, including effort allocation strategies, a fault discovery model, and the simulated testing. Section 5 describes the results and discusses the case study. We summarize this paper in Section 6.

2 Project Context

2.1 Motivation

The target organization is a software purchaser-side company that provides various types of telecommunication services using acquired software systems. In the software acquisition processes, the company is responsible for requirements analysis, architectural design, and acceptance testing, while developer-side companies are in charge of detailed design, programming unit/integration/system testing, and debugging.

As the services grow in the number of variations with shorter renewal cycles than ever before, the main motivation here is *optimization of acceptance testing* to provide high-quality services to customers. From this perspective, the primary goal of this paper is reduction of acceptance test effort using techniques for predicting fault-prone modules. Our study includes metrics collection, building predictor models, and assessing the reduction of test effort.

2.2 Target Software and Modules

The target software consists of workflow modules, basic function modules, library modules (of third parties), and COTS components, with links to external systems such as database and point-to-point systems. As shown in Fig. 1, each functionality is implemented as one or more workflow modules, and each workflow module uses basic function modules, library modules, and COTS components as needed.

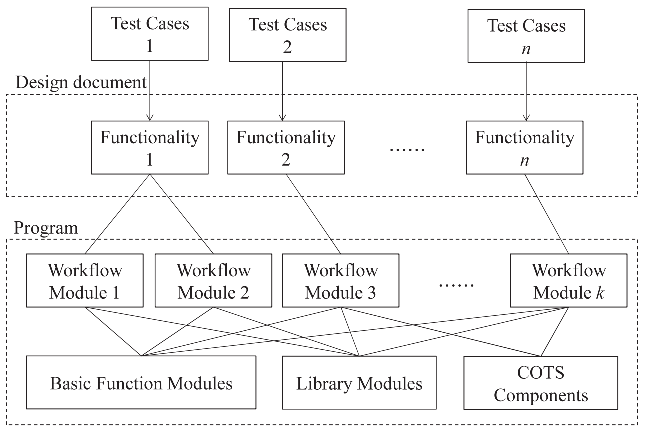


Fig. 1. Modules in target software.

To apply fault module prediction to this software, the primary question we need to answer is "what is the appropriate level of 'module' to be predicted?" Previously, some studies showed package-level prediction was more effective (in terms of recall and accuracy) than file-level prediction [28], [34] while others showed the opposite conclusion (in terms of effort-aware evaluation) [8].

In this study, we approach this question from the perspective of test case allocation. In acceptance testing, each functionality specified in the design document can be independently tested, that is, a set of test cases is developed for each functionality, not for each workflow module or other module/component. Therefore, to optimize test effort allocation, in this paper the appropriate granularity level of "module" to be predicted is a "functionality." Please note that the software purchaser-side company does not need to care about the location of faults in the code because "debugging" after testing is done by the developer-side organization.

In terms of measurements of the module metrics of source code, our targets are the workflow modules only. Other modules/components (i.e., basic function modules, library modules, and COTS components) are out of scope for measurements because more than two functionalities used each of these modules/components; thus it is not considered as a part of an individual functionality.

The programming language used in the workflow modules is a recently developed scripting language that runs on a virtual machine. It is a procedural language somewhat similar to C language. We developed a measurement tool to measure source code metrics such as cyclomatic complexity and nested block depth (NBD).

In addition to the source code metrics, design metrics can be measured from the architectural design document for each functionality. The design document consists of natural language pages and business flow diagrams.

The target software had eight releases in the past. This paper targets the five most recent releases, which we refer to as releases 1 to 5 (5 is the most recent). Releases 1 to 4 are used to construct fault-prone module prediction models, and release 5 is our target to predict fault-prone modules (functionalities) and to conduct the simulation of effort allocation. Table 1 shows the number of modules (functionalities) in each release. The total size of each release is

TABLE 1
Size and the Number of Modules in Each Release

Release	# of modules (functionalities)
1	32
2	34
3	36
4	36
5 (latest)	36

around 25,000 lines of code (KLOC). The percentage of modules having a fault ranges from 14 to 41 percent in each release. (For confidentiality, we cannot show the exact number of faults, sizes of modules, or the name of the programming language.)

2.3 Related Work

While many case studies applying fault prediction to industry datasets have been reported [22], [23], [24], [31], few studies have estimated the reduction of test effort or increase of software quality achieved by fault prediction. Li et al. [17] reported experiences of applying field defect prediction in ABB Inc. Their experiences include practical issues of how to select an appropriate modeling method and how to evaluate the accuracy of prediction across multiple releases in time. They evaluated the usefulness of prediction based on experts' opinions. They reported that modules (subsystems) identified by experts as the most defect prone are among the top four defect-prone modules identified by the prediction model. They also reported that the module prioritization result was actually used by a test team to uncover additional defects in a module previously considered to be low defect prone. Unfortunately, they did not provide any quantitative information about the effort required for additional testing and the number of uncovered additional defects.

Mende and Koschke [18] and Kamei et al. [9] proposed the effort-aware measure P_{opt} to evaluate the fault prediction accuracy. While conventional evaluation measures such as recall, precision, Alberg diagrams, and ROC curves ignore the costs of quality assurance activities, their measure assumes that testing or reviewing a module is roughly likely to be proportional to the size. We used their measure to find the lower bound of the required prediction accuracy needed to discover as many faults as actual testing (Section 4.5).

3 Model Construction

3.1 Fit/Test Dataset

Since the sample size of the fit dataset (for building a prediction model) greatly affects the quality of prediction, we decided to include all modules in releases 1 to 4 in the fit dataset. That is, a total of 138 modules were included in the fit dataset. On the other hand, the test dataset (for evaluating the model) consisted of 36 modules of release 5.

3.2 Objective Variable

There are three candidates for the objective variable: 1) the probability of having a fault, 2) the number of faults, or 3) the fault density. Many researchers have chosen candidate 1 to distinguish fault-prone modules (having at least one fault) from fault-free modules (having no faults). In this case, the number of faults (in fault-prone modules) is ignored. In most datasets, only a small percentage of modules have more than one fault [32]. On the other hand, some practitioners have chosen candidate 2 because they want to allocate quality assurance resources based on the number of expected bugs that exist before testing [11], [17]. Also, some researchers have chosen candidate 3 rather than candidate 1 or candidate 2 because larger files trivially have more defects [14]. In summary, there is no clear consensus about which variable should be predicted.

In this study, we predicted both *the number of faults* and *fault density*, then found out which variable was preferable based on the simulation of test effort allocation. On the other hand, we did not predict the simple probability of having a fault because in our study some modules had two or more faults and such low quality modules should be distinguished from modules having just one fault. Note that we counted faults that we had needed to fix in acceptance testing, while we ignored faults that were reported but not fixed.

3.3 Predictor Variables

Metrics for each module (functionality) could be measured from the design documents and source code as they were available at the purchaser-side organization. To select a set of metrics to be used as predictor variables, we needed to balance the tradeoff between predictive power and the cost of measurement because we needed to develop a measurement tool for our scripting language, which was relatively new and there was no existing measurement tool available.

Table 2 shows the metrics we decided to measure. Since the target project was an enhancement project, and much research has revealed that changes to existing code is the most influential factor for fault injection [2], [6], [20], we measured change metrics such as lines of code added/deleted to the previous release. Also, as control flow metrics are significant influential factors, we measured cyclomatic complexity (VG) and nested block depth as well as their change metrics. Data complexity is also important; however, because its measurement requires deeper code analysis, we did not measure data-related metrics.

For design metrics, we measured natural language documents and business flow diagrams, which are elements of architectural design documents. As shown in Table 2, we measured both base metrics and their change metrics.

As a result we employed 17 predictor variables, while the sample size for model building is 138. We believe this is a reasonable number because a rule of thumb is that five to 10 data points (= modules in this paper) are required for every predictor variable in a typical prediction model [19], [29].

3.4 Prediction Models

Until now, various types of fault-prone module predictors have been used, including the most commonly used linear discriminant analysis [23], logistic regression analysis [21], classification tree [10], support vector machine [8], and random forest [16]. Since we need to predict a "number" (the number of faults) rather than predicting a probability

TABLE 2
Measured Metrics

Type		Name	Definition
Source Code Metrics	Base Metrics	TLOC	Source Lines of Codes
		NBD	Nested block depth
		VG	Cyclomatic complexity
	Change Metrics	ADD	# of added lines
		DEL	# of deleted lines
		CHG	#of changed lines
		ΔNBD	Increase of NBD from previous release
		ΔVG	Increase of VG from previous release
Design Metrics	Base Metrics	PAGE	# of pages
		MOD	# of workflow modules
		PNOD	# of processing nodes in a business flow diagram
		DNOD	# of decision nodes in a business flow diagram
	Change Metrics	REV	# of revisions
		ΔPAGE	Increase of PAGE from previous release
		ΔMOD	Increase of MOD from previous release
		ΔPNOD	Increase of PNOD from previous release
		ΔDNOD	Increase of DNOD from previous release

or conducting a classification, we decided to use random forest, which can predict a number and is one of the promising approaches in fault-prone module prediction [16]. We also employed a linear regression model and Classification and Regression Trees (CART), which are commonly used modeling techniques in software engineering studies [9].

Since we need to predict both the number of faults and fault density, there arises a question whether fault density should be directly predicted or the number of faults should be predicted first, then divided by the module size. We try both approaches in this paper, and compare their prediction performances.

4 Assessment Method

4.1 Effort Allocation Strategies

There are several possible strategies to assign test effort to each module after prediction. This paper tests the following seven strategies:

- *[A1] Equal test effort to all modules*
  This is the most naive strategy, which we consider as a baseline strategy.
- *[A2] Test effort $\propto$ module size*
  Given a module set $(m_1, \ldots, m_n)$, the allocated test effort t_i for the ith module m_i is defined as

$$t_i = t_{total} \cdot S_i / S_{total},$$

where t_{total} is total test effort of all modules, S_i is the size of the ith module, and S_{total} is the total size of all modules.

This is a basic strategy used in industry to assign more test effort to larger modules. Arisholm et al. [1] pointed out that the effort of testing or reviewing a module is likely to be roughly proportional to the size. Indeed, many companies use baseline values for test case density; for example, one must run at least "10 test cases per thousand lines of code" in operational testing.

- *[A3] Test effort $\propto$ new/modified size $+ 0.1 \times$ reused size*
  Allocated test effort t_i is defined as

$$t_i = t_{total} \cdot \left(S_i^{new} + 0.1 \times S_i^{reused}\right) / \left(S_{total}^{new} + 0.1 \times S_{total}^{reused}\right),$$

where S_i^{new} is the lines of new or modified code of the ith module, S_{total}^{new} is the total lines of new or modified code, S_i^{reused} is the lines of reused code of the ith module, and S_{total}^{reused} is the total lines of reused code.

This strategy is an improvement of A2, which distinguishes new/modified code and reused code. Since reused code is much less faulty than new/modified code, this strategy counts only 10 percent of reused lines. (Although 10 percent may not be the optimal value, this is often used in Japanese software companies.)

- *[B1] Test effort $\propto$ # of predicted faults*
  Allocated test effort t_i is defined as

$$t_i = t_{total} \cdot \hat{F}_i / \hat{F}_{total},$$

where $\hat{F}_i$ is the number of predicted faults in the ith module and $\hat{F}_{total}$ is the total number of predicted faults in all modules.

This strategy is a straightforward way to find more faults by allocating more test effort where more faults are predicted.

- *[B2] Test effort $\propto$ predicted fault density*
  Allocated test effort t_i is defined as

$$t_i = t_{total} \cdot (\hat{F}_i / S_i) / \sum_{i=1}^{n} (\hat{F}_i / S_i).$$

Since if faults are evenly distributed larger modules clearly have more faults, some researchers are more interested in predicting fault density rather than simply the number of faults [14]. This strategy allocates more effort to modules with higher fault density, reducing the effect of size.

- *[B3] Test effort $\propto$ # of predicted faults $\times$ module size*
  Allocated test effort t_i is defined as

$$t_i = t_{total} \cdot \hat{F}_i \cdot S_i / \sum_{i=1}^{n} (\hat{F}_i \cdot S_i).$$

This strategy allocates more effort on larger modules if they are likely to contain faults. This is a combination of [A2] and [B1].

- *[B4] Test effort $\propto$ # of predicted faults $\times$ log(module size)*
  Allocated test effort t_i is defined as

$$t_i = t_{total} \cdot \hat{F}_i \cdot \log(S_i) \Big/ \sum_{i=1}^{n} (\hat{F}_i \cdot \log(S_i)).$$

Since strategy B3 allocates enormous effort to an extremely large module, faults in small modules might not be found if the total test effort is limited. Strategy B4 tries to reduce the effect of such very large modules, while still giving larger modules additional effort.

4.2 Fault Discovery Model

This section proposes a fault discovery model that can estimate the number of discoverable faults with respect to the given test resources, the resource allocation strategy, and the set of modules to be tested.

We extend the exponential Software Reliability Growth Model (SRGM). The exponential SRGM is one of the nonhomogeneous Poisson process (NHPP) models, and it is also known as the Goel-Okumoto model [5]. We decided to use this model because it is the simplest NHPP model that has a constant fault detection rate (CFDR) per one fault at an arbitrary testing time [33], which means parameter estimation is much easier than other SRGMs. The exponential SRGM represents the relationship between testing time (effort) and the cumulative number of detected faults as shown in

$$\hat{H}(t) = a[1 - \exp(-bt)]. \tag{1}$$

$\hat{H}(t)$: Expected value of the cumulative number of faults detected by a given testing time (effort).
t : Testing time (effort).
b : Probability of detecting each fault per unit time.
a : The number of initial faults before testing.

In this model, b denotes the ease of finding a fault in software. In our case, b must be individually defined for each module since modules are different in size and complexity. However, the estimation of b for every module is practically impossible since we have 36 modules in the test dataset and more than half of them contain no faults. Alternatively, simply using the same b for all modules is inadequate because some modules are much larger than others, and thus the ease of finding a fault varies widely among modules (e.g., a fault in 10 lines of code is much easier to find than one in 1,000 lines of code).

As a feasible way to handle this problem, this study added a module size parameter to b so that the ease of finding a fault became dependent on the size of the target module.

Equation (2) is our extended SRGM that computes discoverable faults in the ith module based on the given test effort and module size:

$$\hat{H}_i(t_i) = a_i[1 - \exp(-b_i t_i)], \quad b_i = b_0/S_i. \tag{2}$$

$\hat{H}_i(t_i)$: Expected value of the cumulative number of faults detected by a given testing time (effort) of module m_i.
t_i : Testing time (effort) of module m_i.
b_i : Probability of detecting each fault per unit time (effort).
a_i : The number of initial faults in module m_i before testing.

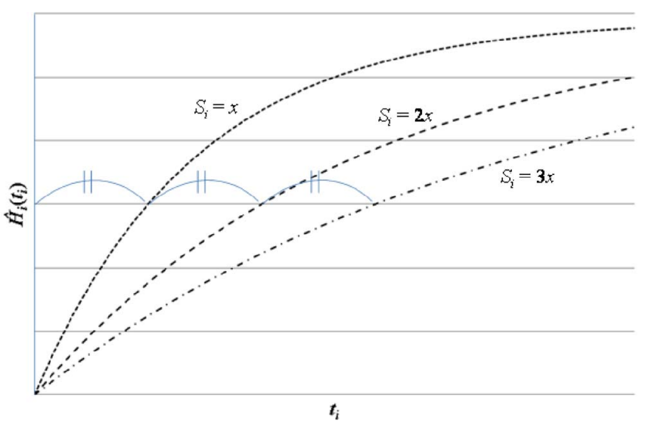


Fig. 2. Example of fault discovery curves for different module sizes.

S_i : Size of module m_i.
b_0 : Constant.

In this model, the fault-detection rate is inversely proportional to the module size. We assume that all modules have the same parameter b_0, that is, given a certain amount of test effort, the ease of finding a fault becomes the same if the module size is the same. Similarly, for example, a double size module requires twice the test effort to find the same number of faults (Fig. 2).

Note that there is no direct relationship between the predictor variables of fault prediction models and the fault discovery model because these variables are used to express "how faulty a module is" while the fault discovery model expresses "how difficult it is to find a fault in a module."

4.3 Parameter Estimation

Before using our extended model (2), we need to estimate a_i and b_0 from the available data in acceptance testing. Obviously, the true value of a_i is unknown because we will never be aware of faults remaining in software after testing. (Note that there is no test that can find 100 percent of defects.) This study assumes, for example, 0.5 faults per kSLOC are still remaining in new/modified code, and 0.05 faults per kSLOC in reused (i.e., not modified) code after testing. Since we have no evidence to say 0.5 and 0.05 are correct, we conducted simulations with several different percentages in Section 5. We refer to these percentages as *remaining fault rates* R_1 (in new/modified code) and R_2 (in reused code). Then, a_i is estimated by the following:

$$a_i = H_i + R_1 \frac{S_i^{new}}{1{,}000} + R_2 \frac{S_i^{reused}}{1{,}000}, \tag{3}$$

where H_i is the actually detected faults in the ith module, S_i^{new} is the lines of new/modified code of the ith module, and S_i^{reused} is the lines of reused code of the ith module.

Next, we need to estimate b_0, which indicates the fault detection rate per unit effort, from actually allocated test effort and faults detected. Since the time series data of test effort and detected faults were unavailable, this study conducted a rough estimation of b_0. Specifically, from (2), we consider the following equation holds for a collection of modules:

$$H_{total} = a_{total}[1 - \exp(-(b_0/S_{total})t_{total})], \tag{4}$$

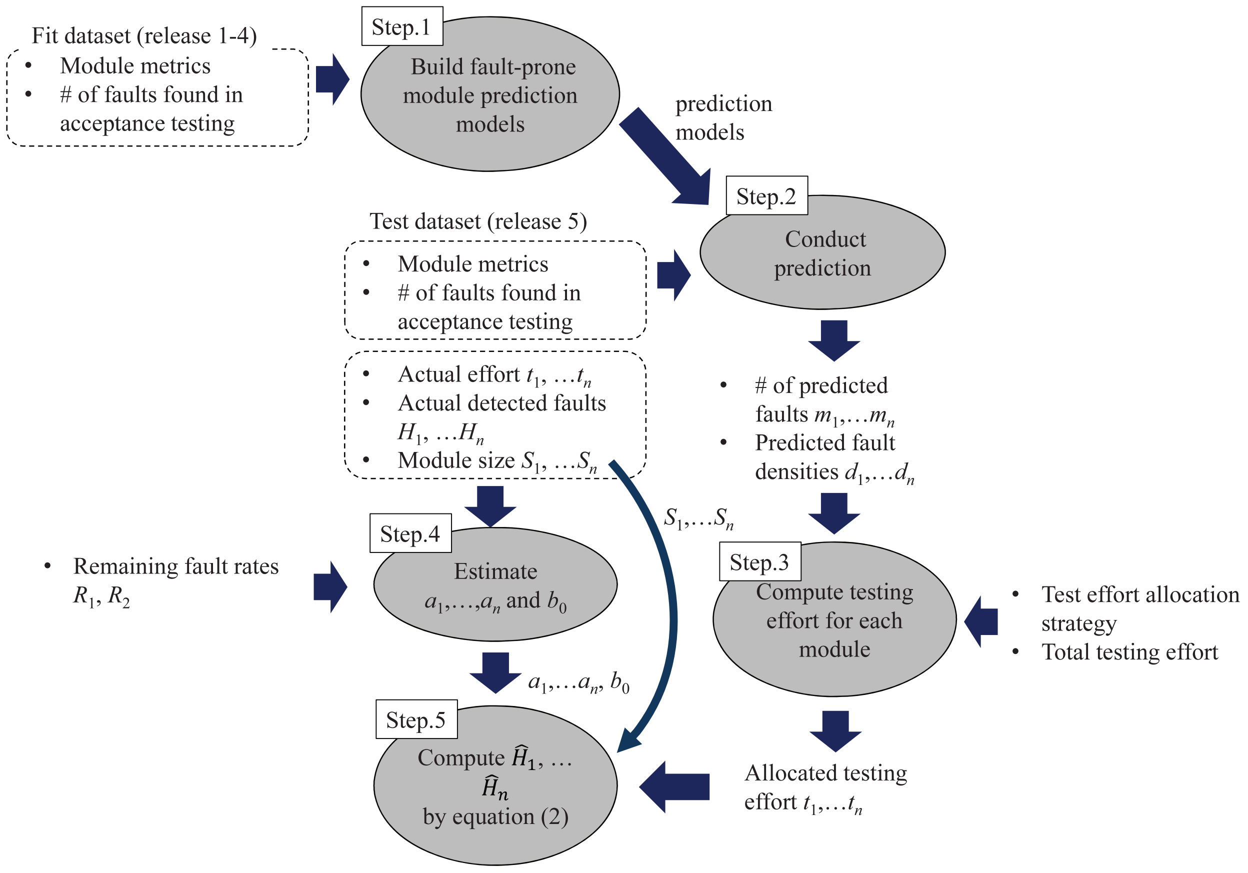

Fit dataset (release 1-4)
Module metrics
# of faults found in acceptance testing
Step.1
Build fault-prone module prediction models
prediction models
Step.2
Conduct prediction
Test dataset (release 5)
Module metrics
# of faults found in acceptance testing
Actual effort $t_1, \dots t_n$
Actual detected faults $H_1, \dots H_n$
Module size $S_1, \dots S_n$
# of predicted faults $m_1,\dots m_n$
Predicted fault densities $d_1,\dots d_n$
$S_1,\dots S_n$
Step.4
Estimate $a_1,\dots,a_n$ and b_0
Remaining fault rates R_1, R_2
Step.3
Compute testing effort for each module
Test effort allocation strategy
Total testing effort
$a_1,\dots a_n$, b_0
Step.5
Compute $\hat{H}_1, \dots \hat{H}_n$ by equation (2)
Allocated testing effort $t_1,\dots t_n$

Fig. 3. Simulation procedure.

where S_{total} is the total size of all modules, t_{total} is the total test effort, H_{total} is the total number of detected faults, and $a_{total} = \sum_{i=1}^{n} a_i$.

From (4), b_0 is computed as follows:

$$b_0 = -S_{total}/t_{total} \cdot \log(1 - H_{total}/a_{total}). \tag{5}$$

4.4 Simulation Procedure

Fig. 3 shows our simulation procedure. In Step 1, we build the fault-prone module prediction models from the fit data set, followed by Step 2, which conducts prediction using the test data set. Next, in Step 3, given the prediction results and affordable (total) test effort, the allocated test effort for each module is computed based on the given test effort allocation strategy (in Section 4.1).

On the other side, Step 4 estimates parameters b_0 and $a_1, \dots, a_n$ of (2), and finally, in Step 5, the expected number of discoverable faults in each module $\hat{H}_i$ is computed based on the allocated test effort $t_1, \dots, t_n$ given from Step 3.

Our goal was to find a test effort allocation strategy that maximized the total discoverable faults $\hat{H}_{total}$ $(= \sum_{i=1}^{n} \hat{H}_i)$ with respect to the affordable (total) test effort. Therefore, we conducted Steps $1, \dots, 5$ for different effort allocation strategies and total test effort levels. We also conducted the procedure with different remaining fault rates R_1 and R_2 because the true values are unknown.

4.5 Evaluation Criteria of Fault Prediction

We need to evaluate the accuracy of fault prediction in Step 3 since it greatly affects the simulation result. Obviously, if the prediction accuracy is very low, then test strategies $\mathrm{B1}, \dots, \mathrm{B4}$ that rely on the prediction cannot discover many faults.

Among various evaluation measures such as recall, precision, F-value, Alberg diagram [23], and ROC curve [16], we decided to use the normalized P_{opt} [9], [18] because it can evaluate the prediction performance in terms of testing effort. P_{opt} is defined as $1 - \Delta_{opt}$, where Δ_{opt} is the area between the LOC-based cumulative lift charts of the optimal model and the prediction model (Fig. 4). In this chart, the x-axis is considered as the required test effort and the y-axis is the maximum number of discoverable faults by the

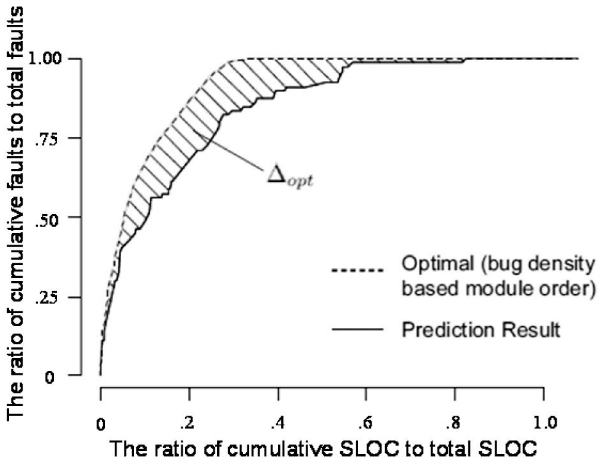


Fig. 4. Example of LOC-based cumulative lift chart (from [9]).

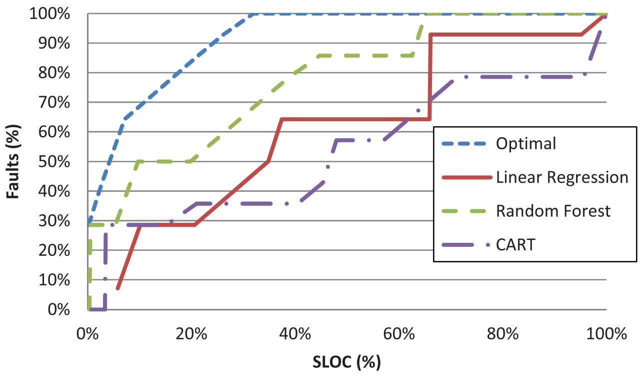


Fig. 5. LOC-based cumulative lift chart (fault density models).

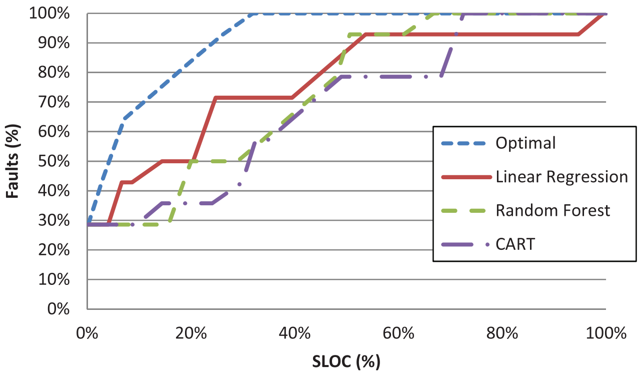


Fig. 6. LOC-based cumulative lift chart (fault count models).

assigned test effort. Since the x-axis indicates the cumulative size of modules, test effort is assumed to be proportional to the size in the LOC-based cumulative lift chart.

Then, the normalized P_{opt} is defined as

$$Norm(P_{opt}) = \frac{P_{opt} - \min(P_{opt})}{\max(P_{opt}) - \min(P_{opt})}, \tag{6}$$

where $max(P_{opt})$ and $\min(P_{opt})$ are calculated on the LOC-based cumulative lift chart in which all modules are ordered by predicted fault density. Here, $max(P_{opt})$ is always 1 because $\Delta_{opt} = 0$ for an ideal curve. To compute $min(P_{opt})$, we consider the worst case of prediction, i.e., modules are ordered in the x-axis by the actual fault density in the "increasing" order.

In contrast to $Norm(P_{opt})$, other measures such as ROC curves evaluate the prediction performance based on the assumption that the test effort is the same across modules, but this assumption is rarely true in many cases.

5 Case Study

5.1 Simulation Setting

5.1.1 Test Effort

In this study, we measured the test effort as the number of test cases instead of person-hours. This implies that the test cases are (approximately) equal in the sense that one person-hour of test effort is equal to another person-hour of test effort. However, we believe using the number of test cases (as the unit for test effort) is preferable to represent the real-world relationship between the test effort and the probability of fault discovery because a fault is discovered by executing a test case. Also, from the perspective of test planning, the company demanded to know the number of required test cases for each module rather than the effort (person-hours) itself.

In our simulation, we used total test effort t_{total} from zero up to 200 percent of the actual effort (test cases) of this project, and computed the total discoverable faults $\hat{H}_{total}$ with respect to the prediction result and test effort allocation strategy.

5.1.2 Remaining Fault Rates

In this case study, we examined three cases of remaining faults rates: $(R_1, R_2) = (1, 0.1), (0.5, 0.05), (0.3, 0.03)$. The estimated parameter b_0 for these cases are $b_0 = 45.2, 64.0, 79.3$, respectively.

According to the IPA/SEC White Papers on Software Development Projects in Japan [7], the average fault rate found in six months after release (i.e., after acceptance testing) is 0.967 per kSLOC in 2-tier client server systems and 0.156 per kSLOC in enhancement projects. Therefore, we consider our R_1 and R_2 values are close to realistic.

5.1.3 Model Construction and Prediction

As described in Section 3, three modeling techniques, random forest, linear regression, and CART (regression tree), were used to construct prediction models from the fit data set of 138 modules. For each modeling technique, we built two models that predicted the fault density and the number of faults, respectively. Afterward, the prediction was conducted using the test dataset of 36 modules.

To build prediction models, we used the statistical computing and graphics toolkit R [25] and its *MASS*, *rpart*, and *randomForest* libraries. In linear regression, forward-backward variable selection based on AIC was applied. In CART, to control the tree growth we used the default parameter values of rpart library, which can be seen by "help(rpart.control)" command in R. Details of how parameter works can be seen in paper [30]. In random forest, we also used the default parameter values of randomForest library, for example, the number of trees to grow $ntree = 500$, and the number of variables randomly sampled as candidates as each split $mtry = sqrt(p)$, where p is the number of predictor variables.

5.2 Prediction Accuracy

Figs. 5 and 6 show the prediction accuracy as an LOC-based cumulative lift chart. Fig. 5 shows the cases with fault densities directly as predicted (we refer to these as *fault density models*), and Fig. 6 shows the cases with the number of faults as predicted, then divided by the module size (we refer to these as *fault count models*). Table 3 shows the results by the $Norm(P_{opt})$ measure.

Interestingly, the two types of prediction (fault density and fault count) showed quite different results. The reason is not totally clear, but this may have happened because even the simple log-transformation of variables can lead to different results [13]. Indeed, the fault density and the fault count each have different value distributions, and

TABLE 3
Prediction Accuracy by $Norm(P_{opt})$

Prediction type	Modeling technique	$Norm(P_{opt})$
Fault density	Random forest	.831
	Linear regression	.633
	CART	.539
Fault count	Random forest	.781
	Linear regression	.818
	CART	.723

models are adjusted to minimize the mean squared error of each distribution.

Among the fault density models (Fig. 5), the random forest showed the best performance $(Norm(P_{opt}) = 0.831)$, while the linear regression (0.633) and CART (0.539) were far beyond it. On the other hand, among the fault count models (Fig. 6), the linear regression showed the best (0.818) and others also showed comparable performances (0.781 and 0.723).

We consider that the random forest's $Norm(P_{opt}) = 0.831$ and 0.781 are comparable to past results. Kamei et al. [9] evaluated the performance of package-level prediction for three datasets, and showed that $Norm(P_{opt})$ was 0.51 in the worst case and 0.89 in the best case (average was 0.73) when they conducted cross-release prediction using random forest with package-level product and process metrics. Our result with random forest is close to the best case of Kamei et al.'s result [9].

In addition, it should be noticed that the prediction accuracy of all models (random forest, linear regression, and CART) may not be stable, i.e., it greatly depends on the dataset (project) [9]. One possible way to handle such instability is to conduct a pilot prediction using past release data of a project to make sure reasonable prediction accuracy can be expected in that project.

5.3 Results of Simulation

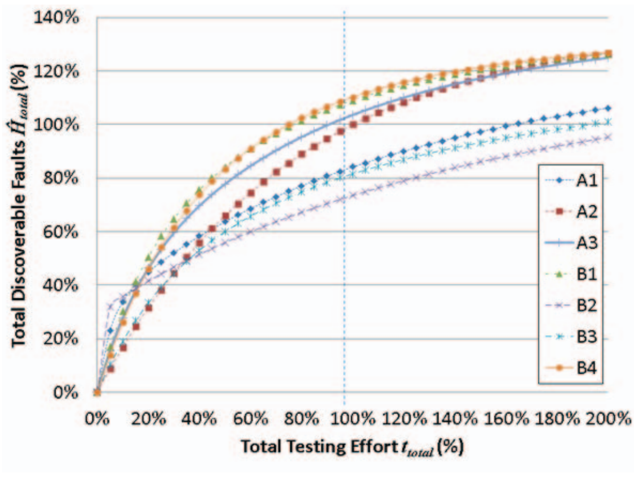
Here, we present the results of simulation with respect to our research questions.

(RQ1) What is the appropriate strategy to allocate test effort to each module after prediction?

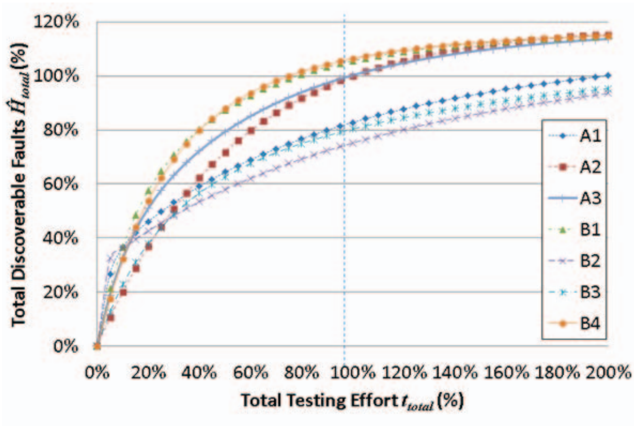
Fig. 7 shows the simulation results that compare the seven test allocation strategies when using the best fault prediction model (random forest, $Norm(P_{opt}) = 0.831$). Figs. 7a, 7b, and 7c show results with different (R_1, R_2). In these figures, the x-axis shows the total test effort t_{total} in percentage where $t_{total} = 100\%$ means that test effort is equal to the actual effort of this project. The y-axis shows the total discoverable faults $\hat{H}_{total}$ in percentage where $\hat{H}_{total} = 100\%$ means that $\hat{H}_{total}$ is equal to the actual discovered faults H_{total}.

From Figs. 7a, 7b, and 7c, the following trends were observed for each strategy:

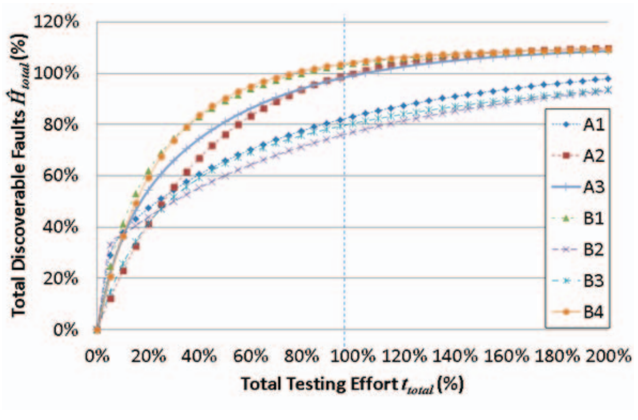
- *Strategy A1 (equal effort).* This strategy worked well only if the test effort is extremely limited ($t_{total} \leq 10\%$).
- *Strategy A2 (effort $\propto$ module size).* This is a basic strategy that can discover 100 percent faults by 100 percent test effort (note that this relationship can be derived from (2) and (5)). This strategy showed the best performance if a system requires extremely high reliability and a huge test resource is available ($t_{total} = 200\%$). Strategies B1 and B4 also showed comparable performance for $t_{total} = 200\%$; however, from the point of view of cost effectiveness, B1 and B4 should not be used because they require fault prediction, while A2 does not. This implies that fault prediction is useless when plenty of test resources are available.



(a) Simulation with $(R_1, R_2) = (1, .1)$



(b) Simulation with $(R_1, R_2) = (.5, .05)$



(c) Simulation with $(R_1, R_2) = (.3, .03)$

Fig. 7. Comparison of effort allocation strategies.

- *Strategy A3 (effort $\propto$ new/modified size $+ 0.1 \times$ reused size).* This strategy outperformed A2 when $t_{total} \leq 80\%$. In case of $t_{total} = 100\%$, this strategy still outperformed A2 when $(\mathrm{R1}, \mathrm{R2}) = (1, 0.1)$, while it underperformed A2 when $(\mathrm{R1}, \mathrm{R2}) = (0.3, 0.03)$. This indicates that strategy A3 should be used instead of A2 when strategy A2 overlooks many faults in new/reused code.
- *Strategy B1 (effort $\propto$ predicted faults).* This strategy outperformed strategies A1, A2, and A3 except for the situation of a huge test resource ($t_{total} \approx 200\%$). This confirmed that the actual engineers' strategy to allocate quality assurance resources based on the number of expected bugs is reasonable [17].
- *Strategy B2 (effort $\propto$ fault density).* This worked well only if the test effort is very limited ($t_{total} \leq 10\%$). Otherwise, this is the worst strategy. From a further analysis, we found that this strategy did not assign enough effort to large modules; thus, faults in large modules were not discovered. This happened because large modules had much lower fault densities than small modules, which corroborates observations in past studies [3], [15].
- *Strategy B3 (effort $\propto$ predicted faults $\times$ module size).* This was not workable at all. This result is somewhat surprising because this strategy is a combination of A2 and B1, both of which performed much better. From a further analysis, we found that there exists a very small module (less than 50 SLOC) containing four faults, and only a small amount of test effort was assigned to the module because the prediction model concluded there would be less than one fault. This indicates that strategy B3 is too sensitive to faulty prediction in small modules.
- *Strategy B4 (predicted faults $\times$ log(module size)).* This showed slightly better performance than strategy B1 when t_{total} is larger than 60 percent. Therefore, if we consider reducing the test effort up to 40 percent, then B4 is the best strategy.

From these results, answers to RQ1 are as follows:

- Strategy B1 and B4 are the two best strategies that can possibly reduce the test effort.
- In the case where there is plenty of test effort available, the basic strategy A2, which does not rely on the fault prediction, was the best. This implies that for extremely high-reliability systems such as aerospace systems, we should not rely on fault prediction (because any prediction involves prediction error, and the prediction itself requires a significant amount of cost.) In addition, we should not apply fault prediction to extremely low-reliability systems. For example, if most of modules contain faults, we do not need to predict which module is faulty.
- In a case of extremely limited test effort, one can consider using strategy B2 (or A1).

Note that these results use the high prediction accuracy model ($Norm(P_{opt}) = 0.831$). We will explore with different accuracies next.

(RQ2) What is the required level of prediction accuracy?

Tables 4a, 4b, and 4c show the required test effort t_{total} to detect as many faults as the actually discovered faults for different (R_1, R_2). In all cases, strategy B4 showed better performance than strategy B1 (as well as all other strategies). Therefore, we concluded that B4 was the best strategy in this case study. When using strategy B4, the random forest model in fault density prediction ($Norm(P_{opt}) = 0.831$) showed the most stable results, which saved $24 \sim 25$ percent effort, regardless of the remaining fault rates (R_1, R_2).

We see that the accuracy measure $Norm(P_{opt})$ and the required testing effort are not consistent, i.e., higher $Norm(P_{opt})$ does not necessarily mean lower testing effort. For example, in the case of $(R_1, R_2) = (0.5, 0.05)$, the $Norm(P_{opt}) = 0.539$ case performed better than the $Norm(P_{opt}) = 0.633$ case when using strategy B1 or B4. Also, for some models, required testing effort became extremely large as the remaining fault rates (R_1, R_2) became smaller. Typically, the linear regression model in fault count prediction ($Norm(P_{opt}) = 0.818$) with strategy B4 saved 26 percent effort when $(R_1, R_2) = (1, 0.1)$, but this model even increased the effort when $(R_1, R_2) = (0.3, 0.03)$. These results imply that we cannot rely on the measure $Norm(P_{opt})$ solely because the required testing effort depends on not only the prediction accuracy, but also the prediction type, the modeling technique, the test strategy, and the remaining fault rates (R_1, R_2). This means simulation of test effort allocation is definitely needed before applying any test strategy based on the fault prediction.

However, to answer RQ2, we could consider the fact that there was no chance to save testing effort when $Norm(P_{opt}) < 0.781$ for any cases $(R_1, R_2) = (0.3, 0.03)$, $(5.0, 0.05)$, or $(1, 0.1)$. As an answer to RQ2, we suggest around $Norm(P_{opt}) \cong 0.78$ as the lower bound to consider using the prediction result. At the same time, we also warn that $Norm(P_{opt}) \geq 0.781$ does not mean there is always a chance to save testing effort, as we see the linear regression model (of $Norm(P_{opt}) = 0.818$) could not save effort at all when $(R_1, R_2) = (0.3, 0.03)$. Therefore, we warn that the suggested required accuracy $Norm(P_{opt}) \cong 0.78$ is just a rough idea and is not the true lower bound. Also, note that this is basically a preliminary example which needs to be repeated on other systems to assess its general validity.

(RQ3) How much is the test effort reduced by the prediction? And, how much effort is needed to conduct prediction?

As shown in Table 4, the maximum test effort we could save in this simulation was 25 percent when we employed the most stable model (the defect density model of random forest) and strategy B4.

Regarding the effort needed to conduct prediction, the company required about 6 percent of the total test effort for metrics collection, data cleansing, and modeling. (Note that this 6 percent does not include the cost required to develop the source code measurement tool.)

Therefore, $25\% - 6\% = 19\%$ effort could be saved with our best prediction model.

5.4 Analysis

This section tries to clarify the important metrics in our fault prediction model. We analyze the impact of the metrics by *IncNodePurity*, which is the mean decrease in node impurity of a random forest model [4]. A higher

TABLE 4
Required Test Effort to Discover 100 Percent Faults

(a) Simulation with (R_1, R_2) = (1, .1)

Prediction type	Modeling technique	$Norm(P_{opt})$	Required testing effort t_{total} (%)						
			A1	A2	A3	B1	B2	B3	B4
Fault density	Random forest	0.831	163%	100%	91%	77%	232%	193%	75%
	Linear regression	0.633	163%	100%	91%	139%	919%	110%	119%
	CART	0.539	163%	100%	91%	124%	614%	147%	108%
Fault count	Random forest	0.781	163%	100%	91%	99%	470%	112%	88%
	Linear regression	0.818	163%	100%	91%	80%	274%	167%	74%
	CART	0.723	163%	100%	91%	110%	358%	146%	102%

(b) Simulation with (R_1, R_2) = (.5, .05)

Prediction type	Modeling technique	$Norm(P_{opt})$	Required testing effort t_{total} (%)						
			A1	A2	A3	B1	B2	B3	B4
Fault density	Random forest	0.831	199%	100%	99%	78%	266%	255%	75%
	Linear regression	0.633	199%	100%	99%	195%	1669%	131%	160%
	CART	0.539	199%	100%	99%	129%	750%	162%	112%
Fault count	Random forest	0.781	199%	100%	99%	105%	577%	119%	92%
	Linear regression	0.818	199%	100%	99%	96%	470%	248%	87%
	CART	0.723	199%	100%	99%	121%	458%	160%	111%

(c) Simulation with (R_1, R_2) = (.3, .03)

Prediction type	Modeling technique	$Norm(P_{opt})$	Required testing effort t_{total} (%)						
			A1	A2	A3	B1	B2	B3	B4
Fault density	Random forest	0.831	229%	100%	105%	80%	295%	298%	76%
	Linear regression	0.633	229%	100%	105%	375%	3213%	253%	311%
	CART	0.539	229%	100%	105%	132%	863%	171%	115%
Fault count	Random forest	0.781	229%	100%	105%	108%	667%	124%	95%
	Linear regression	0.818	229%	100%	105%	139%	767%	383%	124%
	CART	0.723	229%	100%	105%	127%	528%	167%	117%

IncNodePurity indicates that a variable plays a more important role in a prediction model.

Fig. 8 shows IncNodePurity assigned by the fault density model of random forest (our best fault density model.) Abbreviations in Fig. 8 are explained in Table 2. In this case study, two design metrics, PNOD and ΔPNOD, which are related to processing nodes in a business flow diagram, were the most influential factors. Interestingly, the code churn metrics ADD, DEL, and CHG did not contribute so much in our case. This suggests that practitioners who want to reduce the acceptance test effort based on fault prediction should measure design documents as well as source code.

Table 5 shows the selected (or important) variables in each model. For random forest, the top five important variables (in terms of IncNodePurity) are shown. For linear regression, selected and statistically significant ($p < 0.05$) variables are shown. For CART, selected variables are shown. As shown in Table 5, fault density models and fault count models showed quite different results. In fault density models, only PNOD is the common important variable among three models. On the other hand, in fault count models, REV and MOD are commonly important. Four variables ΔMOD, PAGE, NBD, and VG were not selected in all models; however, their related variables MOD, ΔPAGE, ΔNBD, and ΔVG were all selected in one of the models. These results indicate that practitioners who want to conduct fault prediction should not reduce the number of variables because a variable that is not useful in some model could be useful in other models. It is important future work to investigate why the selected variables are so different among the models.

5.5 Threats to Validity

In this section, we discuss the threats to the validity of our work. The results of this study rely on the fault discovery model, which we extended from the conventional exponential SRGM. Since the model is a simple abstraction of real-world

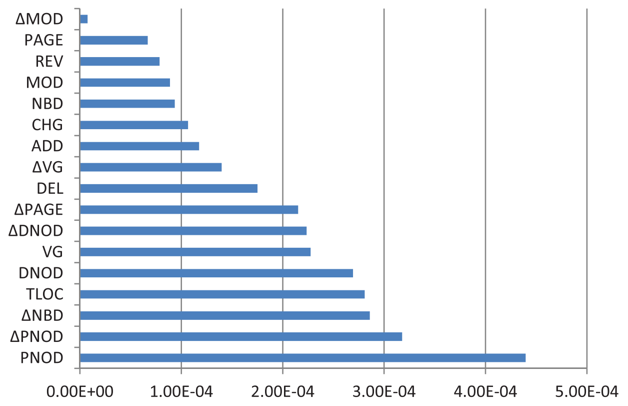


Fig. 8. IncNodePurity metrics for each metric in the fault density model of random forest.

TABLE 5
Selected or Important Variables in Each Model

Prediction type	Modeling technique	Selected or important variables					
Fault density	Random forest	PNOD	ΔPNOD	ΔNBD	TLOC	DNOD	
	Linear regression	ΔDNOD	CHG	DEL	PNOD	DNOD	ΔPAGE
	CART	DEL	TLOC	PNOD			
Fault count	Random forest	REV	ΔVG	CHG	MOD	DEL	
	Linear regression	REV	ΔDNOD	ΔPAGE	MOD	PNOD	DNOD
	CART	CHG	REV	MOD	ADD		

testing, our estimation might be too simplified to match the actual testing situation. However, because this is the first attempt to estimate the reduction of test effort by fault prediction, we expect further research to improve the model to be more realistic. For example, we might consider using complexity metrics (e.g., cyclomatic complexity) instead of module size S_i in (2) because more test effort is needed to discover a fault in a more complex module. But still, we also believe we will have a very similar result because the cyclomatic complexity is highly correlated with the module size (in our case, correlation $\text{coefficient} = 0.966$).

This paper used datasets collected from five releases of one software project. We need to conduct case studies with other projects to generalize our results.

This paper compared seven strategies to allocate test effort to each module. However, these seven are by no means complete. We need to conduct simulations with other strategies to find better strategies in the future.

This paper did not consider the severity or the potential impact of the defect. We found that the severity ratings of defects are very often subjective and inaccurate; thus, many researchers do not use the severity in their fault prediction studies [19]. However, severity is generally important in prioritizing the test effort; thus, it is our future work to assess the severity ratings in the simulation model. One possible suggestion is to build different predictor models for high-severity defects and low-severity defects, then conduct simulation for each prediction model.

6 Conclusion

To evaluate the cost effectiveness of fault prediction, this paper compared seven test effort allocation strategies using the proposed simulation model. Our findings from a case study in the telecommunication industry include the following:

- Strategy B1 (test effort $\propto$ predicted faults) and B4 (test effort $\propto$ predicted faults $\times$ log(module size)) were the two best strategies that could possibly reduce the test effort.
- When there was plenty of test effort available, the basic strategy A2, with test effort proportional to the module size (i.e., not relying on fault prediction), detected the largest number of faults.
- Strategy B2 (test effort $\propto$ fault density) worked well only if the test resource was extremely limited. Strategy A1 (equal test effort to all modules) is also workable in this situation.
- By using strategy B4 with our best fault prediction model, it showed that the test effort could be reduced by 25 percent to detect as many faults as actual discovered faults, while the company required about 6 percent of the test effort for metrics collection, data cleansing, and modeling.
- We suggest around $Norm(P_{opt}) \cong 0.78$ as the lower bound to consider using the prediction results because there was no chance to save testing effort when $Norm(P_{opt}) < 0.781$ for any cases. However, we also found that $Norm(P_{opt})$ is not a reliable measure, and it should be considered a very rough idea of the required accuracy.

These results suggest that reduction of the test effort is achieved only if the appropriate test strategy is employed with a sufficiently high fault-prediction accuracy. However, where sufficient data are available to fit a prediction model and develop good fault prediction accuracy, the appropriate test strategy can significantly reduce the necessary level of test effort while still maintaining the same level of fault detection or provide a higher level of fault detection with the same test effort.

While considerable future work is needed to confirm these results and generalize them, it seems clear that fault prediction models provide another tool for practitioners to use in reducing the costs of testing or increasing the quality produced by testing. Such models help pinpoint where testing will be most effective in finding and removing faults. That means the test effort is applied where it can do the most good, as if the testers had a map showing where the faults were most likely to be found.

Acknowledgments

This work was conducted in part in the StagE Project, the Development of Next Generation IT Infrastructure, supported by Ministry of Education, Culture, Sports, Science and Technology, Japan. Also, part of this work was conducted under Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (C) (22500028).

References

[1] E. Arisholm, L.C. Briand, and E.B. Johannessen, "A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models," *J. Systems and Software*, vol. 83, no. 1, pp. 2-17, 2010.
[2] M. D'Ambros, M. Lanza, and R. Robbes, "An Extensive Comparison of Bug Prediction Approaches," *Proc. Seventh IEEE Working Conf. Mining Software Repositories*, pp. 31-41, 2010.
[3] V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM*, vol. 27, pp. 42-52, 1984.

[4] A.S. Foulkes, *Applied Statistical Genetics with R.* Springer, 2009.

[5] A.L. Goel and K. Okumoto, "Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures," *IEEE Trans. Reliability,* vol. 28, no. 3, pp. 206-211, Aug. 1979.

[6] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Trans. Software Eng.,* vol. 26, no. 7, pp. 653-661, July 2000.

[7] "Information-Technology Promotion Agency, Japan (IPA) Software Engineering Center (SEC) ed.," White Papers on Software Development Projects in Japan, 2010-2011 Ed., 2010.

[8] Y. Kamei, A. Monden, and K. Matsumoto, "Empirical Evaluation of SVM-Based Software Reliability Model," *Proc. Fifth ACM/IEEE Int'l Symp. Empirical Software Eng.,* vol. 2, pp. 39-41, 2006.

[9] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A.E. Hassan, "Revisiting Common Bug Prediction Findings Using Effort Aware Models," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 1-10, 2010.

[10] T.M. Khoshgoftaar and E.B. Allen, "Modeling Software Quality with Classification Trees," *Recent Advances in Reliability and Quality Engineering,* pp. 247-270, World Scientific, 1999.

[11] T.M. Khoshgoftaar, A. Pandya, and D. Lanning, "Application of Neural Networks for Predicting Program Fault," *Annals of Software Eng.,* vol. 1, pp. 141-154, 1995.

[12] S. Kim, E.J. Whitehead Jr., and Y. Zhang, "Classifying Software Changes: Clean or Buggy?" *IEEE Trans. Software Eng.,* vol. 34, no. 2, pp. 181-196, Mar./Apr. 2008.

[13] B. Kitchenham and E. Mendes, "Why Comparative Effort Prediction Studies May Be Invalid," *Proc. Fifth Int'l Conf. Predictor Models in Software Eng.,* article 4, 2009.

[14] P. Knab, M. Pinzger, and A. Bernstein, "Predicitng Defect Densities in Source Code Files with Decision Tree Learners," *Proc. Third Working Conf. Mining Software Repositories,* pp. 119-125, 2006.

[15] A.G. Koru, D. Zhang, K. El Emam, and H. Liu, "An Investigation into the Functional Form of the Size-Defect Relationship for Software Modules," *IEEE Trans. Software Eng.,* vol. 35, no. 2, pp. 293-304, Mar./Apr. 2009.

[16] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings," *IEEE Trans. Software Eng.,* vol. 34, no. 4, pp. 485-496, July/Aug. 2008.

[17] P.L. Li, J. Herbsleb, M. Shaw, and B. Robinson, "Experiences and Results from Initiating Field Defect Prediction and Product Test Prioritization Efforts at ABB Inc.," *Proc. 28th Int'l Conf. Software Eng.,* pp. 413-422, 2006.

[18] T. Mende and R. Koschke, "Revisiting the Evaluation of Defect Prediction Models," *Proc. Int'l Conf. Predictor Models Software Eng.,* pp. 1-10, 2009.

[19] T. Menzies, O. Jalali, J. Hihn, D. Baker, and K. Lum, "Stable Rankings for Different Effort Models," *Automated Software Eng.,* vol. 17, no. 4, pp. 409-437, 2010.

[20] R. Moser, W. Pedrycz, and G. Succi, "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction," *Proc. 30th Int'l Conf. Software Eng.,* pp. 181-190, 2008.

[21] J.C. Munson and T.M. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Trans. Software Eng.,* vol. 18, no. 5, pp. 423-433, May 1992.

[22] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," *Proc. 28th Int'l Conf. Software Eng.,* pp. 452-461, 2006.

[23] N. Ohlsson and H. Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches," *IEEE Trans. Software Eng.,* vol. 22, no. 12, pp. 886-894, Dec. 1996.

[24] T.J. Ostrand, E.J. Weyuker, and R.M. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Trans. Software Eng.,* vol. 31, no. 4, pp. 340-355, Apr. 2005.

[25] R. "The R Project for Statistical Computing," http://www.r-project.org/, 2013.

[26] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A.E. Hassan, "High-Impact Defects: A Study of Breakage and Surprise Defects," *Proc. ACM SIGSOFT Symp. Foundations Software Eng.,* pp. 300-310, 2011.

[27] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" *Proc. Int'l Conf. Mining Software Repositories,* pp. 1-5, 2005.

[28] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting Component Failures at Design Time," *Proc. ACM/IEEE Int'l Symp. Empirical Software Eng.* , pp. 18-27, 2006.

[29] H.B. Tan, Y. Zhao, and H. Zhang, "Conceptual Data Model-Based Software Size Estimation for Information Systems," *ACM Trans. Software Eng. Methodologies,* vol. 19, no. 2, pp. 1-37, Oct. 2009.

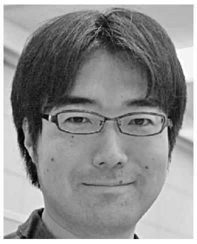
[30] T.M. Therneau, E.J. Atkinson, and M. Foundation, "An Introduction to Recursive Partitioning Using the RPART Routines," http://cran.r-project.org/web/packages/rpart/vignettes/longintro.pdf, 2012.

[31] A. Tosun, B. Turhan, and A. Bener, "Practical Considerations in Deploying AI for Defect Prediction: A Case Study within the Turkish Telecommunication Industry," *Proc. Fifth Int'l Conf. Predictor Models in Software Eng.,* pp. 1-9, 2009.

[32] B. Turhan, T. Menzies, A. Bener, and J. Distefano, "On the Relative Value of Cross-Company and Within-Company Data for Defect Prediction," *Empirical Software Eng.,* vol. 14, no. 5, pp. 540-578, 2009.

[33] S. Yamada and S. Osaki, "Software Reliability Growth Modeling: Models and Applications," *IEEE Trans. Software Eng.,* vol. 11, no. 12, pp. 1431-1437, Dec. 1985.

[34] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," *Proc. Third Int'l Workshop Predictor Models Software Eng.,* 2007.



Akito Monden received the BE degree in 1994 in electrical engineering from Nagoya University, and the ME and DE degrees in 1996 and 1998 in information science from NAIST. He is an associate professor in the Graduate School of Information Science at the Nara Institute of Science and Technology (NAIST), Japan. He was a honorary research fellow at the University of Auckland, New Zealand (2003-2004). He is a member of the IEEE, ACM, IEICE, IPSJ, and JSSST.

Takuma Hayashi received the BE degree in information science from the Okayama Prefecture University (OPU), Japan, in 2007, and the ME degree in information science from the Nara Institute of Science and Technology (NAIST), Japan, in 2009. He is a member of the staff of the Nippon Telegraph and Telephone West (NTT West) corporation, Japan.

Shoji Shinoda received the BE and ME degrees from Osaka University in 1996 and 1998, respectively. He is currently with NTT West R&D Center.

Kumiko Shirai graduated from Keio University, Japan, in 1998. She is a researcher at the R&D Center, NTT West. She received the best presentation award of the 62nd National Convention of IPSJ in 2001, and the Young Researcher Award of the 10th IPSJ Special Interest Groups (SIG) on Distributed Processing System (DPS) Workshop in 2002.

Junichi Yoshida received the BE and ME degrees from Ehime University in 1994 and 1996, respectively. He is an engineer at the NTT West R&D Center at Osaka, Japan. He joined the NTT Network Service System Laboratories, Tokyo, Japan, in 1998, where he was engaged in the development of a ATM switching system and a 10-Gb/s firewall system for network security in Photonib Era. He is currently engaged in development of client application with the NTT West R&D Center.

Mike Barker is a professor in the Graduate School of Information Science at the Nara Institute of Science and Technology, Japan. His career has been in software development and project management, with almost two decades spent in higher education at MIT and NAIST. He is a long-term member of the ACM, the IEEE, and is a PMP-certified member of PMI.

Kenichi Matsumoto received the PhD degree in information and computer sciences from Osaka University. He is a professor in the Graduate School of Information Science at yjr Nara Institute of Science and Technology, Japan. His research interests include software measurement and software process. He is a senior member of the IEEE, and a member of the ACM, the IEICE, and the IPSJ.