ImpactScale: Quantifying Change Impact to Predict Faults in Large Software Systems

Kenichi Kobayashi[1], Akihiko Matsuo[1], Katsuro Inoue[2], Yasuhiro Hayase[3], Manabu Kamimura[1], Toshiaki Yoshino[4]

[1]Fujitsu Laboratories Limited
Kawasaki, Kanagawa, Japan
{kenichi, a_matsuo, kamimura.manabu}@jp.fujitsu.com

[2]Osaka University
Suita, Osaka, Japan
inoue@ist.osaka-u.ac.jp

[3]University of Tsukuba
Tsukuba, Ibaraki, Japan
hayase@cs.tsukuba.ac.jp

[4]Fujitsu Limited
Tokyo, Japan
yoshino.toshi@jp.fujitsu.com

***Abstract*—In software maintenance, both product metrics and process metrics are required to predict faults effectively. However, process metrics cannot be always collected in practical situations. To enable accurate fault prediction without process metrics, we define a new metric, *ImpactScale*. ImpactScale is the quantified value of change impact, and the change propagation model for ImpactScale is characterized by *probabilistic propagation* and *relation-sensitive propagation*. To evaluate ImpactScale, we predicted faults in two large enterprise systems using the effort-aware models and Poisson regression. The results showed that adding ImpactScale to existing product metrics increased the number of detected faults at 10% effort (LOC) by over 50%. ImpactScale also improved the predicting model using existing product metrics and dependency network measures.**

I. INTRODUCTION

Software fault prediction has achieved a great outcome as a mean of supporting reviews and tests, controlling qualities, and managing risks [1][2]. However, in the maintenance or post-release phase of software lifecycle, fault prediction is a relatively difficult task. Fault prediction requires metrics correlated with faults as explanatory variables. Several studies have reported the set of faulty modules in pre-release phase differs from the set of ones in post-release phase [3][4]. Metrics useful in pre-release phase may often become useless in post-release phase. For example, Fenton et al. [3] reported the case that McCabe's cyclomatic complexity correlated with pre-release faults but little correlated with post-release faults.

To improve predictive performance in maintenance, process metrics have been used in addition to product metrics. Product metrics are metrics extracted from a snapshot of source code, and process metrics are metrics extracted from project activity records such as histories of changes, inspections and defects [5]. Some studies have reported that formerly changed code fragments tend to be fault-prone and metrics extracted from such changes improved predictive performance [4][5][6].

However, in practical maintenance activities, documents, records and human knowledge are often lost. In such situations, process metrics cannot be obtained. Our goal is to define a new product metric which correlates with faults and improves predictive performance in maintenance.

Even in long maintained software systems, one of surviving factors of faults is software dependency [7][8]. When a module is changed in a system, some of the modules dependent on it should be also changed if needed. The process of change is repeated until no change is needed. If modules under such a ripple effect of changes [9] were not fully captured, faults may happen. Logical coupling [7] is co-change information in software products, and it is well-known that metrics based on logical coupling have relations with faults [5]. Hassan et al. [8] reported that logical coupling can be used to recognize change propagation. Logical coupling can be regarded as the observation of implicit dependency through software changes.

There are various types of software dependencies. Product metrics regarding direct dependencies are categorized into syntactic dependency, and process metrics regarding logical coupling are categorized into logical dependency [7]. Cataldo et al. [10] compared the strengths of the correlations between various dependencies and faults. Their examination showed that the correlation between syntactic dependency and faults was insignificant or weak and that the correlation between logical dependency and faults was significant and the strongest. The results are one of the cases that product metrics are insufficient for fault prediction in maintenance.

Zimmermann et al. applied social network analysis (SNA) on a software dependency graph representing relationships between binary modules of software systems [11]. They reported that adding network measures from SNA literature could improve the performance of fault prediction. Although the network measures are product metrics, they are not covered by the syntactic dependency categorized by Cataldo.

We assumed that the network measures used by Zimmermann and the logical dependency used by Cataldo share common factors of fault-proneness. Therefore, we assumed that change impact analysis [12] on source code enables us to extract implicit dependency, such as relations exposed by logical coupling. Change impact analysis is a technique that detects affected areas of source code when some part is changed. In Cataldo's examination the number of logical couplings of a given module was most correlated with faults. Therefore, we expected the scale of estimated areas affected by any changes correlates with fault-proneness as well as the number of logical couplings, and we hypothesized as follows:

> **Hypothesis 1:** A metric that quantifies the scale of change impact can improve the performance of fault prediction in large software systems.

doi: 10.1109/ICSM.2011.6080771 ©2011 IEEE ICSM 2011, Williamsburg, VA, USA

Accepted for publication by IEEE. © 2011 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

However, it is difficult to compute the exact affected areas of change impact. Static analysis [13] has a nature that it may derive excessive (false positive) areas. Besides, it requires enormous computation time to improve the accuracy. Dynamic analysis [14] can easily capture dynamically bound areas, but it has a nature that it may fail to capture affected areas which are seldom used. In reality, there are many cases for which dynamic analysis cannot be performed. A practical technique to find the affected areas has been proposed for the case where the change to a given module is already known [15]. However, since we need to know the area before the change is given, it is difficult to compute the areas while minimizing false positives.

Fortunately, to meet our goal it is not necessary to solve the problem of computing the affected areas. It is enough to solve a more relaxed problem of calculating the total approximate quantity of the affected areas. We defined a propagation model in which change impact is propagated probabilistically and relation-sensitively, and we hypothesized as follows and called the quantity *ImpactScale*.

Hypothesis 2: The propagation model with relation-sensitive propagation has enough predictive performance to substitute for an accurate but expensive analysis.

The remainder of the paper is organized as follows: we will define ImpactScale in section II. The change propagation model for ImpactScale will be also described. In section III, we will evaluate the improvement of the predictive performance by adding ImpactScale. In section IV, we will discuss about ImpactScale and its definition. The threats to validity of ImpactScale will be discussed in section V. We will talk about related works in section VI and will conclude in section VII.

II. IMPACTSCALE

In this section, we define a new metric, *ImpactScale* (abbr. as *IS*), and we describe the change propagation model for it.

A. Dependency Graph and Propagation Graph

In this paper, a *dependency graph* is a multi-relational directed graph, where the nodes are code entities (methods, functions, and so on) and data entities (database tables, global variables, and so on), and where the edges represent the relationships or dependencies between nodes. The edges are labeled with the *relation type*, such as call, data, import or inheritance dependency. In a dependency graph, multiple edges between any two nodes are allowed. Formally, the dependency graph G_D is defined as $G_D = \langle V, E \rangle$, where V is a set of nodes and E is a set of edges. Edge $e \in E$ is defined as $e = \langle s, t, rel \rangle$, where $s \in V$ is a source node, and $t \in V$ is a target node, and $rel \in$ Rel is a relation type. Rel is the set of all relation types.

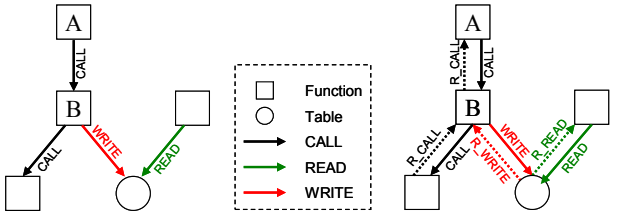


Figure 1. Example of dependency graph G_D (left part) and propagation graph G_P (right part)

The left part of Figure 1 shows an example of a dependency graph. In the figure, there are three relation types, CALL, READ, and WRITE. When function A calls function B, it is said that "there is a CALL relationship from function A to function B." Function A and B may share some status through arguments. Since it can be hardly determined whether two functions A and B were designed under the top-down-approach or the bottom-up-approach, either change propagation from A to B or from B to A cannot be ruled out. In order to capture remote logical dependencies, we take a very conservative policy that change propagation in a dependency graph has bidirectional nature.

A *propagation graph* consists of a dependency graph and the reversed edges of all of its edges. A propagation graph corresponding to $G_D = \langle V, E \rangle$ is defined as $G_P = \langle V, E_P \rangle$, where E_P is defined as $E_P = E \cup \{\text{reverse}(e) \mid e \in E\}$. reverse($e$) means the reversed edge $\langle t, s, \text{R\_}rel \rangle$ of an edge $e = \langle s, t, rel \rangle$. Relation type R_*rel* means the reversed relation type of *rel*. The quantity of change impact is calculated on a propagation graph. The right part of Figure 1 shows the propagation graph G_P corresponding to the dependency graph G_D in Figure 1.

B. Probabilistic Propagation

Haney [9] proposed an analysis model that a change in a module probabilistically propagates to another module, and there are some studies which assumed that changes propagate probabilistically [16][17]. In the propagation graph in Figure 2, it is assumed that the change impact from v_0 to v_1 is propagated with probability r_1, when there is edge e_1 from v_0 to v_1. The same goes for v1 and v2. It is also assumed that the change impact from v_0 to v_2 is propagated with probability $r_1 r_2$.

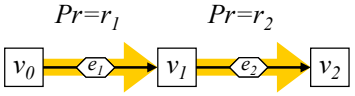


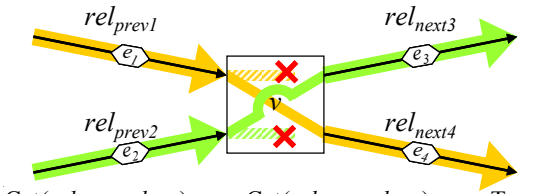
Figure 2. Example of probabilistic propagation

C. Relation-sensitive Propagation

Since we took the aforementioned conservative propagation policy, there is a worry that the affected areas are overestimated. To prevent excessive estimation, we introduced some light-weight constraints to the propagation policy. In call graph analysis, context-sensitive analysis, which refers previous call and status histories, is often used to improve the accuracy of the analysis [13]. Since the analysis requires enormous computation time, we could not use it. However, we borrowed an idea from it to solve our problem.

To prevent explosion of computation time, we used a minimal context, the relation type of the previous edge, which is practically collectable and requires minimal computation time. Concretely, whether propagation from one node to its next is cut or not is determined by the projection Cut. Cut is a projection whose domain is a pair of the relation type of the previous edge and the relation type of the next edge, and it is defined as Cut: Rel × Rel → {True, False}. For example, in the propagation graph in Figure 3, at node v, the relation type of the previous edge e_1 is rel_{prev1}, and the relation type of the next edge e_3 is rel_{next3}. The projection Cut for the graph is given for in the bottom half of the figure. The change impact from edge e_1 to edge e_3 does not propagate, because Cut(rel_{prev1}, rel_{next3}) is

True. We call such propagation controlled by relation types *relation-sensitive propagation*.



$\mathrm{Cut}(rel_{prev1}, rel_{next3}) = \mathrm{Cut}(rel_{prev2}, rel_{next4}) = \mathrm{True}$

$\mathrm{Cut}(rel_{prev1}, rel_{next4}) = \mathrm{Cut}(rel_{prev2}, rel_{next3}) = \mathrm{False}$

Figure 3. Example of relation-sensitive propagation

D. Definition of ImpactScale

In this subsection, we give the definition of ImpactScale. In the propagation graph G_P=<V, E_P>, path p from node s to node t is denoted as $p = (e_1, e_2 \ldots e_n)$, where edge e_i is each edge in path p in the order of pass. The source of e_1 is s, and the target of e_n is t. The same node can be passed two or more times in a path. Strictly speaking, the path is a *trail*, in the terminology of graph theory. The propagation rate r_i of each edge e_i is given as

$$r_i = R_P \cdot \mathrm{m}(e_i),$$

where R_P is the base propagation rate and is hypothetically 0.5 in this paper, and $\mathrm{m}(e_i)$ is the propagation modifier attached to edge e_i, and it is normally 1.0[1], though it can take any value in the range of (0, 1] or $1/R_P$. $\mathrm{Q}_{\mathrm{path}}(p)$, the quantity of change impact via path p, is given as

$$\mathrm{Q}_{\mathrm{path}}(p) = \frac{1}{R_P}\prod_{i=1}^{n} r_i = R_P{}^{n-1}\prod_{i=1}^{n}\mathrm{m}(e_i).$$

$\mathrm{P}_{s,t}$ is the set of all paths reachable from s to t. $\mathrm{P}_{s,t}$ is calculated by path finding in the propagation graph. When a search reaches a node in a path, projection Cut determines whether the search goes forward to the next node through the next edge or is terminated. The previous edge of the starting node is not defined, thus the propagation from it must occur. $\mathrm{Q}(s, t)$, the quantity of change impact from s to t, is given as

$$\mathrm{Q}(s,t) = \begin{cases} \max_{p \in \mathrm{P}s,t} \mathrm{Q}_{\mathrm{path}}(p) & : \ \mathrm{P}_{s,t} \neq \phi \\ 0 & : \ \mathrm{P}_{s,t} = \phi \end{cases}.$$

Calculating $\mathrm{Q}(s, t)$ is essentially equivalent to solving the shortest path problem. ImpactScale, the quantity of change impact from s, is denoted as $\mathrm{IS}(s)$, and it is given as

$$\mathrm{IS}(s) = \sum_{t \in \mathrm{V} \setminus s} \mathrm{Q}(s,t)$$

(V\s means a set of all elements of V except s). Finally, the quadruplet ImpactScale = <IS, R_P, Rel, Cut> is the definition of ImpactScale.

E. Cut Rules

The set Rel and projection Cut can be configured arbitrarily. In this paper, Rel is given as Rel = {CALL, READ, WRITE, R_CALL, R_READ, R_WRITE}. CALL means a function call, READ means a read access, and WRITE means a write access (which also implies a read access). R_CALL, R_READ, and R_WRITE are the reversed relation types, respectively. Cut is given as

$$\mathrm{Cut}(p,n) = \begin{cases} \mathrm{True} & : \ p = \mathrm{CALL} \wedge n = \mathrm{R\_CALL} & (\mathrm{Rule\,1}) \\ \mathrm{True} & : \ p = \mathrm{R\_CALL} \wedge n = \mathrm{CALL} & (\mathrm{Rule\,2}) \\ \mathrm{True} & : \ p = \mathrm{READ} & (\mathrm{Rule\,3}) \\ \mathrm{False} & : \ otherwise & \end{cases}.$$

The condition part of the above notation of projection Cut is called *cut rules*.

The above cut rules are heuristic. Their validity will be discussed in section IV. In this paragraph we explain their intent. The propagation from CALL to CALL is assumed to be under the top-down-approach design. The propagation from R_CALL to R_CALL is assumed to be under the bottom-up-approach design. Rule 1 and Rule 2 cut propagations out of the both design. These rules are designed on the assumption that the propagation will rarely switch from one design approach to the other. Rule 3 is designed based on the analogy of dataflow analysis.

F. Computational Complexity

Most of the calculation of ImpactScale is occupied by path finding. Since a propagation graph for ImpactScale is a multi-relational graph, ordinary graph algorithms cannot be applied. Whaley et al. [18] presented a technique for context-sensitive analysis. The technique duplicates nodes and reduces a complex path finding problem with context to a simple path finding problem without context. By using the technique, path finding for ImpactScale is also reduced to path finding of a simple-relational graph, and ordinary algorithms can be applied. When Dijkstra's algorithm is applied to a propagation graph G_P=<V, E_P>, the computational complexity of the problem is $O(R|E_P| + R|V|(\log(R|V|))$, where R is the number of relation types. It can be calculated in practical time, even if the target software system is fairly large.

G. Examples

The left part of Figure 4 is an example of calculating ImpactScale of node C. Paths (C→A), (C→D), (C→X), (C→X, X→F), (C→X, X→F, F→E), and (C→X, X→F, F→H) are found. Since rel(C→A) is R_CALL and rel(A→B) is CALL, path (C→A, A→B) is cut by Rule 2 at edge A→B. Similarly, path (C→X, X→F, F→H, H→G) is cut by Rule 1 at edge H→G. ImpactScale of C is the sum of Q(C,A), Q(C,D), Q(C,X), Q(C,F), Q(C,E) and Q(C,H); that is 4.0.

The right part of Figure 4 is an example of calculating ImpactScale of node J. It has a peculiar pattern of *relation-sensitive propagation*. Though path (J→K, K→L) is cut by Rule 1 at edge K→L, path (J→K, K→Y, Y→K, K→L) is not cut. As a result, the change impact from node J propagates to node L. ImpactScale of J is the sum of Q(J,K), Q(J,Y) and Q(J,L); that is 1.625. Data node Y works as the medium between node J and node L. If node Y was missing, only path (J→K) would be found, and ImpactScale of J would equal 1.0.

[1] Values other than 1.0 are used to adjust propagation rates, e.g., among different abstraction levels of entities such as database tables and variables.

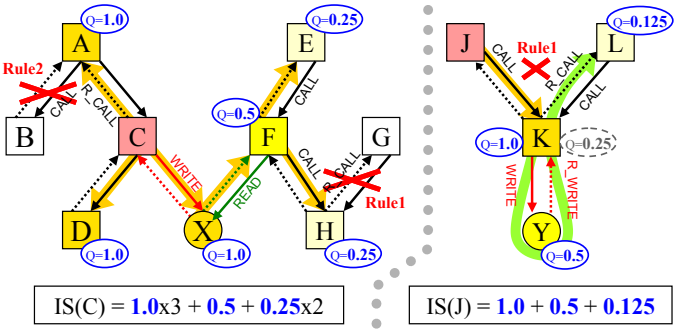


Figure 4. Examples of calculating ImpactScale

H. Use Cases

ImpactScale is used in various situations. We have been using it in the following cases.

- Quality improvement in the restricted budget and schedule. This topic is further discussed in this paper.
- Rapid effort estimation of bug-fixing tasks. It is assumed that size of the change impact of the focused source code affects the man-hours of fixing tasks.
- Rapid quality assessment of a whole software system
- Watching code change activities to keep software modularity. A prominent increase of ImpactScale may be a symptom of modularity violations.

III. Evaluation

To test Hypothesis 1, 2 and the significance of ImpactScale, the following research questions are our main concerns.

RQ1: *Does adding ImpactScale to existing product metrics improve predictive performance?*

RQ2: *Does adding ImpactScale to existing product metrics and network measures improve predictive performance?*

Following subsections are organized as follows.

- In subsection A-C, we describe the experimental setup.
- For RQ1, we evaluate prediction performances using binary classification in subsection D and using fault-density prediction and effort-aware models [22] in subsection E.
- For RQ2, we evaluate using effort-aware models and hierarchical model analysis in subsection F.

A. Target Software Systems

For evaluation, we chose large enterprise accounting software systems from two different companies. The criteria of selecting them were that they are maintained for the long term, considerably large, and described in languages that can be easily analyzed. The intent of the last criterion is to minimize the influence of the quality of the source code analysis in this evaluation. The collected data sets of the two systems are called DS1 and DS2. The profiles of them are shown in Table I.

Both systems are written in COBOL language. We collected fault reports of both systems from over 40 months.

TABLE I. PROFILES OF TARGET SOFTWARE SYSTEMS

Data set	Modules	Total LOC	Faults	Faulty modules
DS1	5.8k	1.6M	269	215
DS2	7.6k	3.7M	250	208

One module is one "program" in COBOL language, which is a unit of the target of a call operation and corresponds to a function in other languages. One "program" is also one source code file. The collected metrics are described in Table II.

TABLE II. COLLECTED PRODUCT METRICS

Metrics	Description
LOC	Number of lines of code without comments and blank lines
WMC	Total sum of McCabe's Cyclomatic Complexity of sections
MaxVG	Max value of McCabe's Cyclomatic Complexity of sections
Sections	Number of sections (which corresponds to the number of blocks in a given module)
Calls	Number of calls
Fan-in	Number of modules which call a given module
Fan-out	Number of modules which are called by a given module
IS	ImpactScale

B. Measuring ImpactScale

The procedure of measuring ImpactScale is as follows. First, we extracted call operations (CALL) and read/write operations to databases and files (READ, WRITE) from the target software system by using static analysis. Then, we built a dependency graph. Next, for each node in the graph, we calculated ImpactScale. In this evaluation, m(e) is always 1.0. Total calculation time was at most several tens of seconds (the CPU used was a Core2Duo 2.5GHz). Figure 5 shows the distributions of the measured ImpactScale and its statistics. In the figure, most of modules have small ImpactScale and only small part of modules have large ImpactScale. In our experience, the distributions of ImpactScale of most systems have similar tendencies. Figure 6 shows an example of calculating ImpactScale of a module in DS1.

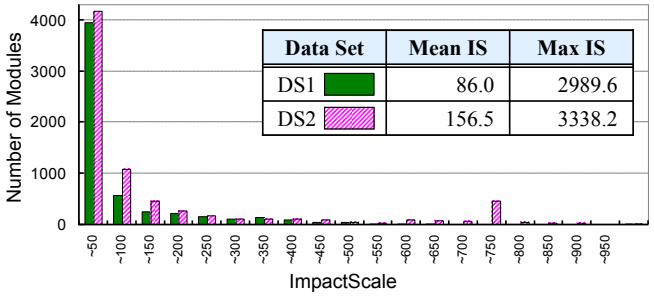


Figure 5. Measured ImpactScale

C. Cross Validation

To evaluate predictive performance using the data sets, we performed 100 times random sub-sampling cross validations for each evaluation. For each validation, we randomly selected 2/3 of all modules of the target data set as a training set, and the remaining 1/3 of modules were used as a test set.

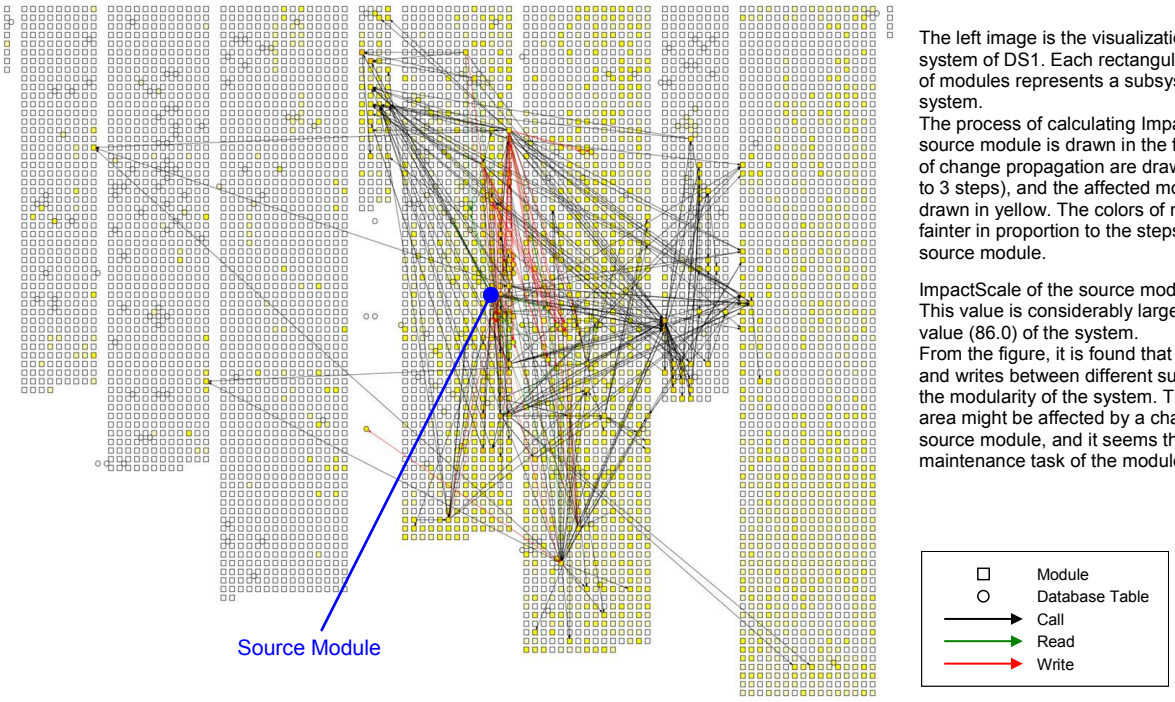


Figure 6. Example of calculating ImpactScale in the actual software

D. *Binary Classification (RQ1)*

We evaluate how much the predictive performance is improved by adding ImpactScale to existing product metrics set using a binary classifier.

1) *Logistic regression*

As a classifier, we used logistic regression model. An expression of logistic regression model is represented as

$$y = \frac{\exp(b_1x_1 + b_2x_2 + \ldots + b_0)}{1 + \exp(b_1x_1 + b_2x_2 + \ldots + b_0)},$$

where y is a response variable, and each x_i is an explanatory variable, and each b_i is a partial regression coefficient. The expression $b_1x_1+b_2x_2+ \ldots +b_0$ is called a linear predictor. To estimate the partial regression coefficients, the maximum likelihood estimation method is used. A response variable y ($0<y<1$) is interpreted as the probability of fault-proneness.

2) *Model selection*

To select the best prediction model, the set of the optimal explanatory variables should be selected for the training set of each validation. To build predicting models, we prepared metrics in Table II as the candidate set of explanatory variables. For each metric X, log(X) (or log(X+1), if the metric contains zero value) was also added into the set as the alternative candidate. That is, for each metric X, there were three choices, X, log(X), or without X. Log-transformation for skewed data is a standard technique. To select the best model, we used AIC (Akaike's Information Criteria). If the likelihood of the model is higher, and if the number of parameters is fewer, AIC becomes less. Less AIC means the model is better. To select the best model without ImpactScale, 3^7 combinations were tried, and the model with the least AIC was taken. To select the best model with ImpactScale, 3^8 combinations were tried. To avoid multicollinearity, we rejected the models with a variable whose VIF (Variance Inflation Factor) is more than 10 [19].

3) *Training and testing*

For each validation, we selected and trained the best model, "MET", using existing product metrics in Table II excluding ImpactScale, and we selected and trained the best model, "MET+IS", using existing product metrics and ImpactScale. For training, if the number of faults is more than zero, it is regarded as a positive. Otherwise it is regarded as a negative.

Then, we predicted all modules in the test set. For each module in the test set, if the response variable y is more than given *cutoff* value, the module is classified as a faulty module. Since both DS1 and DS2 had low fault rate (3.7% and 2.7%), we chose 0.1 for *cutoff*.

4) *Performance measures*

To evaluate predictive performance, for each validation, we computed *precision*, *recall* and *F1* values using a confusion matrix shown in Table III. "FP" means the number of modules predicted as faulty but observed as not faulty, and vice versa. Recall is calculated as TP/(TP+FN). Recall close to 1.0 means most faults are detected. Precision is calculated as TP/(TP+FP). Precision close to 1.0 means few modules are mis-predicted as faulty. F1 is a summarizing measure of recall and precision and calculated as their harmonic mean, 2/(1/recall+1/precision).

TABLE III. CONFUSION MATRIX

		Predicted	
		positive	**negative**
Observed	**positive**	TP	FN
	negative	FP	TN

5) Results

Table IV shows the averages of performance measures of the 100 times validations for both data sets. "Model MET" and "Model MET+IS" columns show the calculated measures for both models. "Imprv. by IS" column shows each improvement from MET to MET+IS. The results show adding ImpactScale improves the predictive performance statistically significantly for every performance measure in both DS1 and DS2. This evaluation supports RQ1 is YES.

TABLE IV. PERFORMANCE IMPROVEMENT IN BINARY CLASSIFICATION

Data set	Perf. measure	Model MET		Model MET+IS		Imprv. by IS†
		mean	(stddev)	mean	(stddev)	
DS1	Precision	0.148	(0.029)	0.168	(0.031)	+0.020
	Recall	0.315	(0.051)	0.392	(0.048)	+0.077
	F1	0.200	(0.033)	0.234	(0.034)	+0.034
DS2	Precision	0.139	(0.030)	0.162	(0.033)	+0.023
	Recall	0.253	(0.042)	0.334	(0.057)	+0.081
	F1	0.177	(0.029)	0.216	(0.034)	+0.039

†All improvements are significant (P<0.001) in Wilcoxon's signed rank test.

E. Effort-aware evaluation (RQ1)

1) Effort-aware model and performance measures

Recent studies pointed out that the effort of testing modules should be taken into consideration in predictive performance evaluations [20][21][22][23]. Modules classified as fault-prone tend to be large because of the correlation between faults and size [3]. Arisholm et al. reported the effort of testing a module is roughly proportional to the size of the module [20]. In practical maintenance tasks, budget and schedule are often very demanding. Therefore, cost-effectiveness of fault prediction had become important concerns for practitioners.

To involve this discussion, we use the effort-aware model proposed by Mende et al [22]. In the effort-aware model, the relative risk $R_{dd}(x)$ is used to prioritize modules to test or inspect. $R_{dd}(x)$ is defined as *#errors*(x) / E(x), where *#errors*(x) is the number of faults in module x and E(x) is the required effort for module x. We use LOC for E(x) as previous studies [22][23]. In this case, $R_{dd}(x)$ means fault density. In the effort-aware model, $R_{dd}(x)$ (i.e., fault density) is predicted, and modules are tested or inspected in the descending order of fault density. To review predictive performance, effort-based cumulative lift chart is used as shown in Figure 7. In the figure, the solid curve represents the overall prediction outcome. The curve is called *cost-effectiveness curve* [20] or *effort-vs-PD curve* [21]. The X coordinate of the curve represents the relative cumulative effort, and the Y coordinate represents the fault detection rate at the spent effort. If the curve is steeper, the prediction is more effort-effective. If the curve is under the diagonal line, the prediction is almost random and meaningless. The dashed curve labeled as "Optimal Model" is the outcome of the perfect prediction and the upper limit of the performance.

Figure 7 also explains two performance measures used in this paper. "AUC" is the Area Under the effort-vs-PD Curve [21]. AUC represents the accumulated performance in all range. If AUC is close to 1 or its upper-limit, it means the predictive performance is high. If AUC is around 0.5 or less, the prediction is meaningless. "ddr" is the "defect detection rate" [22]. In this paper, "ddr_x" means fault detection rate at the x% of effort. In maintenance, affordable effort is usually low. Thus, this measure directly answers to practitioners' questions such as, "How many faults are detected in the review of the first 10% of LOC?" High ddr_x means high predictive performance. Though some previous studies used 20% as x, 20% is too big for large systems in our industrial experience. Therefore, we mainly use ddr_{10} rather than ddr_{20} in this paper. If ddr_{10} is less than 0.1, it means the benefit of the prediction is nothing.

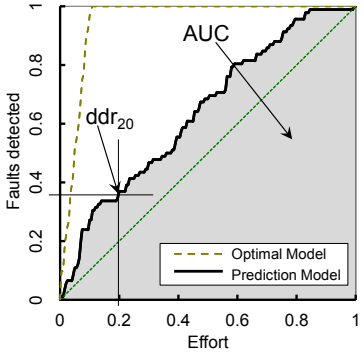


Figure 7. Example of effort-based cumulative lift chart, AUC and ddr

2) Fault density prediction using Poisson regression

To predict fault density, a predicting model which outputs a continuous value or count value is required. In this paper, we used Poisson regression model [24], which is a common model to predict the count of probabilistic events and has been used in the software fault prediction literature [5][25]. An expression of Poisson regression model is generally represented as

$$y = \exp(b_1x_1+b_2x_2+ \ldots +b_0).$$

To estimate the partial regression coefficients, the maximum likelihood estimation method is used. A response variable y ($0<y$) is interpreted as the expectation of the number of faults. Since its response variable is count of events, the density of events can be straightforwardly derived. To predict density, an offset term "log(LOC)" with a fixed coefficient 1.0 is added to the linear predictor as follows:

$$y = \exp(b_1x_1+b_2x_2+ \ldots +b_0+\log(\mathrm{LOC})).$$

Notwithstanding the existence of the offset term, "log(LOC)" can be still added to explanatory variables.

3) Model Selection, training and testing

We selected best models, trained and tested as subsection D except that the number of faults is directly for training.

4) Results

Table V and Figure 8 show the results of 100 times validations. In the figure, each red dashed curve shows the performance of the model without ImpactScale (DS1-MET / DS2-MET). Each blue solid curve shows the performance of the model with ImpactScale (DS1-MET+IS / DS2-MET+IS). White dots on the curves are measured points of ddr_{10} and ddr_{20} shown in Table V. In both DS1 and DS2, MET+IS models outperform MET models in most range of the curves and all three measures in the table. Especially, the improvement of ddr_{10} is remarkable. The results mean that 1.5 times faults can be detected in the 10% of effort. This evaluation supports RQ1 is YES.

TABLE V. PERFORMANCE IMPROVEMENT IN FAULT DENSITY PREDICTION AND EFFORT-AWARE MODEL

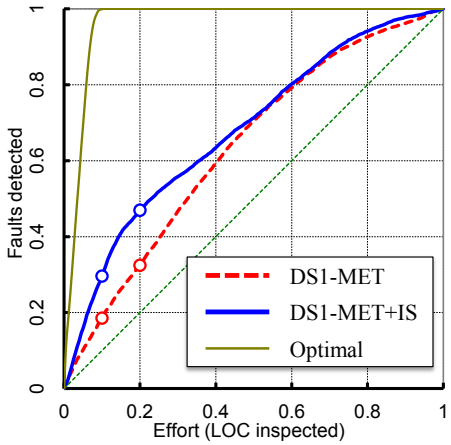
Perf. measure	DS1-MET mean	DS1-MET (stddev)	DS1-MET+IS mean	DS1-MET+IS (stddev)	Imprv. by IS†
AUC	0.635	(0.027)	0.680	(0.027)	+0.045
ddr_{10}	0.186	(0.042)	0.296	(0.051)	×1.60
ddr_{20}	0.325	(0.043)	0.470	(0.055)	×1.45

The upper-limit of the AUC is 0.977.

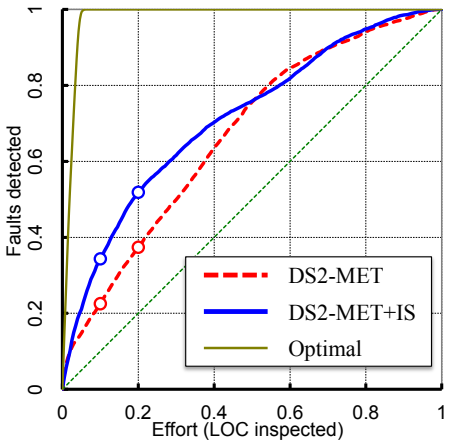
Perf. measure	DS2-MET mean	DS2-MET (stddev)	DS2-MET+IS mean	DS2-MET+IS (stddev)	Imprv. by IS†
AUC	0.669	(0.025)	0.714	(0.025)	+0.045
ddr_{10}	0.225	(0.047)	0.343	(0.046)	×1.53
ddr_{20}	0.374	(0.053)	0.518	(0.051)	×1.39

The upper-limit of the AUC is 0.984.

†All improvements are significant (P<0.001) in Wilcoxon's signed rank test.



(a) Effort-based cumulative lift chart of DS1



(b) Effort-based cumulative lift chart of DS2

Figure 8. Effort-based comparison between models without ImpactScale (MET) and with ImpactScale (MET+IS) (100 times average)

F. *Comparison with network measures (RQ2)*

Zimmermann et al. introduced social network analysis (SNA) on a software dependency graph to predict faults [11]. They and several replication studies [26][27] showed the effectiveness of network measures. Since ImpactScale is also measured on a dependency graph, in order to confirm its meaningfulness, RQ2 must be assessed.

RQ2: *Does adding ImpactScale to existing product metrics and network measures improve predictive performance?*

1) *Network measures in SNA*

Network measures are measures of various network topological characteristics. Zimmerman collected 58 network measures using UCINet tool [28] and predicted faults with them. To see the complete list of the measures and their explanation, refer to [11] and [29]. We also used UCINet as previous studies [11][26][27]. We collected network measures on the same dependency graph of DS1 on which ImpactScale was measured. Since the dependency graph of DS2 is too large for UCINet, we used only DS1 in this evaluation.

2) *Principal Component Regression*

We also use fault density prediction by Poisson regression and the effort-aware model. Explanatory variables are so many (over 60) that multicollinearity problem is inevitable. To cope with multicollinearity, we use principal component analysis (PCA) [30] as previous studies. PCA is an unsupervised algorithm to find principal components (PCs). Since all PCs are orthogonal, multicollinearity can be avoided in regression using PCs as explanatory variables. This combination is called principal component regression.

3) *Model selection, training and testing*

We set up four models as follows.

MET	a model consisted of existing product metrics
MET+IS	a model consisted of existing product metrics and ImpactScale
MET+SNA	a model consisted of existing product metrics and network measures
MET+SNA+IS	a model consisted of existing product metrics, network measures and ImpactScale

For each validation, for each model, all of metrics and measures were log-transformed and then were transformed to PCs using PCA. Next, PCs up to 99% cumulative variance were used as explanatory variables in Poisson regression. The average numbers of used PCs were 6.0 (MET), 7.0 (MET+IS), 27.8 (MET+SNA), and 28.8 (MET+SNA+IS). Testing procedure is the same with subsection E.

4) *Results*

To compare the models, we use hierarchical modeling. The results are shown in Figure 9. Four circles represent the four predicting models with their performance measures, AUC and ddr_{10}. Each solid arrow means a simpler model is extended to a more complex model by adding a set of variables. For example, arrow (a) shows that adding ImpactScale to MET improves the predictive performance by 0.054 in AUC and by 0.074 in ddr_{10}. It is consistent with the results of subsection D and E. Arrow (b) shows the effectiveness of network measures, and it supports previous SNA studies.

To assess RQ2, we focus on arrow (d) and (e). Arrow (d) shows that adding ImpactScale to MET+SNA is significant. It means that ImpactScale contains distinct explanatory factor from network measures and that adding ImpactScale is always meaningful. Arrow (e) shows that ImpactScale has higher detection rate than network measures in the case effort is restricted. However, overall predictive performance is slightly lower. At any rate, it can be said the predictive performance delivered by solo ImpactScale is comparable to a set of network measures. These results support RQ2 is YES. Network measures will be discussed further in Related works section.

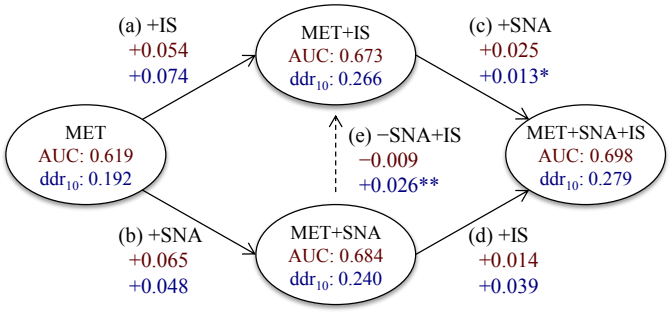


Figure 9. Hierarchical model comparison between network measures and ImpactScale. All improvements and deterioration are significant in Wilcoxon's signed rank test. (*: P<0.05, **: P<0.01, unmarked: P<0.001)

IV. Discussion

In this section, we discuss whether ImpactScale surely contributes to fault prediction and whether the definition of ImpactScale in section II is valid.

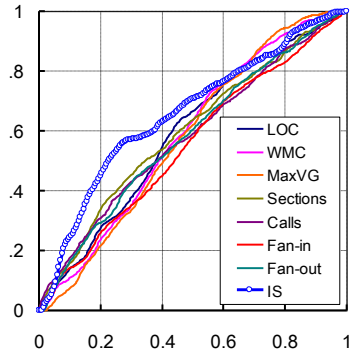
A. *Contribution to Fault Prediction*

First, we tested the correlations between ImpactScale and other metrics. Table VI shows the Spearman rank correlations of all pairs of metrics in Table II in DS1. In the table, the correlations of ImpactScale are at most 0.38 and obviously low. The result in DS2 is similar. For network measures, the correlations between ImpactScale and them are at most 0.60 and enough low to deliver the improvement shown in Figure 9. The results mean ImpactScale is independent of other metrics.

TABLE VI. SPEARMAN RANK CORRELATIONS BETWEEN METRICS (THE FIGURES OVER 0.5 ARE SHOWN IN BOLD LETTERS.)

	WMC	MaxVG	Section	Calls	Fan-in	Fan-out	IS
LOC	**0.90**	**0.71**	**0.80**	**0.69**	-0.26	**0.66**	0.18
WMC	-	**0.88**	**0.59**	**0.51**	-0.13	0.48	0.11
MaxVG		-	0.35	0.34	-0.05	0.32	0.04
Section			-	**0.78**	-0.33	**0.79**	0.31
Calls				-	-0.32	**0.94**	0.33
Fan-in					-	-0.32	0.17
Fan-out						-	0.38

Next, we predicted the fault density by Poisson regression with single metric. The results (100 times average) of the prediction are shown in Figure 10. For each metric except ImpactScale, its AUC is only slightly higher than 0.5, and its ddr_{10} is around 0.1; therefore, it has little predictive ability. In contrast, ImpactScale delivers obviously high predictive performance. Therefore, ImpactScale strongly contributes to the fault prediction in DS1. The same goes for DS2.



Metrics	AUC	ddr_{10}
LOC	0.591	0.137
WMC	0.586	0.107
MaxVG	0.578	0.073
Sections	0.596	0.156
Calls	0.579	0.173
Fan-in	0.549	0.139
Fan-out	0.580	0.162
IS	**0.654**	**0.242**

Figure 10. Results of fault density predictions with single metric

On the basis of the results of RQ1, RQ2 and this subsection, we can say that Hypothesis 1 is valid in this paper.

Hypothesis 1: A metric that quantifies the scale of change impact can improve the performance of fault prediction in large software systems.

B. *Validity of ImpactScale Definition*

In this subsection, we tested the validity of the definition of ImpactScale. We modified the definition of ImpactScale and examined how much the predictive performance of the model varies by using the ddr_{10}.

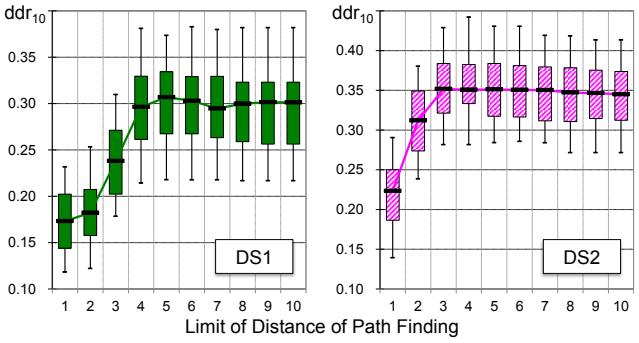


Figure 11. Effect of considering distant nodes

First, we tested the meaningfulness of considering distant nodes on a propagation graph. We limited the maximum length of path for calculating ImpactScale. Figure 11 shows the effect of considering distant nodes by changing the definition of ImpactScale using the best model in subsection III.E including modified ImpactScale for DS1 and DS2. For each box plot, the horizontal axis shows each limit value of the maximum graph distance for path finding. When the limit is 1, ImpactScale is very similar to the sum of Fan-in and Fan-out. Therefore, if ddr_{10} at limit 1 were large enough, considering distant nodes would be meaningless. In the figure, the ddr_{10} curve of DS1 is increasing until limit 5, and the ddr_{10} curve of DS2 is increasing until limit 3. These results mean that the consideration is meaningful. Besides, the minimum distances to be considered are different depending on the software systems. Since distant enough nodes hardly influence the ddr_{10} of both DS1 and DS2, the limit distance has only to be an adequately large number.

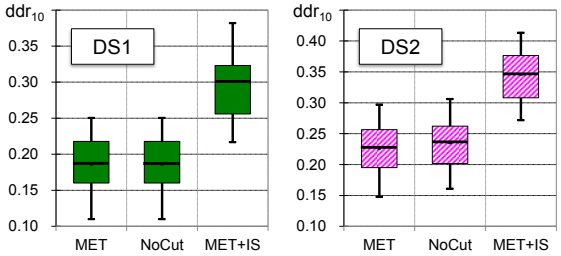


Figure 12. Effect of relation-sensitive propagation

Next, we tested the meaningfulness of the *relation-sensitive propagation*. In each box plot in Figure 12, the model MET+IS is the best model with ImpactScale, which was selected in the manner in subsection III.E. The model MET is the best model without ImpactScale. The model NoCut is the best model with modified ImpactScale that does not cut any propagation[3]. On the basis of the comparison, the ddr_{10} of NoCut has very little improvement over the ddr_{10} of MET. Therefore, it can be said that the definition without the cut rules is meaningless. Conversely, it can be said that the cut rules make the correlation between ImpactScale and faults higher. The results examined by using ddr_{20} and AUC were similar to Figure 11 and Figure 12. Consequently, we can say that Hypothesis 2 is valid in this paper.

Hypothesis 2: The propagation model with relation-sensitive propagation has enough predictive performance to substitute for an accurate but expensive analysis.

V. THREATS TO VALIDITY

In this section we discuss the threats to the validity of our proposal and results. ImpactScale has no language specific feature in its definition, but the target systems of our evaluation are written in COBOL. Since COBOL differs from other languages in paradigms and manners, some consideration should be made to expand the discussion in this paper to other languages. For example, we used a function (a program in COBOL) as a unit for predicting faults, but in other languages such a unit may usually be a class or a source file, which is a group of functions. The target systems are in the domain of accounting software. Further evaluations for systems in various domains of software should be performed.

We used seven traditional metrics to evaluate our results. If target systems were written in object oriented (OO) languages, many existing OO metrics [31] should be considered. We still need to study further whether ImpactScale is even meaningful in addition to a rich set of metrics in such OO languages.

The measurement of ImpactScale requires miscellaneous dependency information such as a call graph. If the accuracy of the information is low, the calculated ImpactScale may not correctly reflect the total quantity of the affected areas. Since ImpactScale is a summary statistic, it is expected not to be sensitive to the accuracy, but we have not estimated the influence of the accuracy yet. In this paper, to avoid the influence, we chose systems written in COBOL because dynamic bindings in such systems are usually small.

[3] That is, for $\forall p,n \in$ Rel, projection Cut(p,n) = False.

VI. RELATED WORKS

Some existing code metrics, such as Fan-in, Fan-out, LCOM and CBO [31], are used to measure the software structure consisting of code entities and their relations [2]. They consider only direct relations between adjacent nodes in a dependency graph, but our evaluation proved that indirect and distant nodes in the graph strongly affect faults. Therefore, we should take those nodes into consideration to predict faults.

Process metrics regarding the amount of changes work well as explanatory variables in fault prediction [4][5][6]. It is interesting whether ImpactScale is independent of such process metrics. Cataldo et al. [10] reported that the amount of changes and logical coupling are independent of each other and both correlate with the faults. Since logical coupling and ImpactScale share common factors such as dependencies and change impact, we expect that ImpactScale is also independent of the amount of changes.

Geiger et al. showed some relations between code clones and logical couplings [32]. Since code clones cannot be fully captured by call dependencies and data dependencies, some parts of logical couplings are not covered by ImpactScale. It is easy to add relations of code clones into a propagation graph for ImpactScale (e.g., add a new relation type CLONE), and the addition may improve predictive performance further.

In recent studies, metrics traversing software dependency graphs are emerging. Inoue et al. proposed Component Rank [33] to find significant components by considering *use* dependencies between software components by using a Markov Chain model. Hayase et al. proposed Maintenance Point [1] to estimate the maintenance effort using a dependency graph constructed by change impact analysis. Maintenance Point is calculated as the weighted sum of efforts of the affected modules. Several studies [16][17] have proposed change prediction models based on the assumption that change impact propagates probabilistically. Tsantalis's model [17] predicts the change probability of a module using change history. Each change must be identified as propagated or originating change by human intervention, thus the model has scalability problem.

Zimmermann et al. [11] regarded binary modules as actors and applied SNA to the dependency graph. They and replication studies [26][27] analyzed the target system with principal component regression by using over 50 network measures in SNA and existing product metrics. They showed that adding network measures to existing product metrics improved the predictive performance. The principal components composed of a lot of variables are very difficult to interpret by human analysts. In contrast, since ImpactScale is a designed and stand-alone metric correlated with faults, its interpretation is easy and intuitive. Besides, ImpactScale can capture data dependency handily by relation-sensitiveness.

VII. CONCLUSION

We defined a new product metric *ImpactScale* based on the hypothesis that the scale of change impact is the significant factor of faults in large software systems. The change propagation model for ImpactScale is characterized by *probabilistic propagation* and *relation-sensitive propagation*. ImpactScale can be calculated in practical time for fairly large systems, and its interpretation is intuitive.

We predicted faults of two large enterprise software systems and evaluated the predictive performance using precision/recall/F1 measures and the effort-aware models. In all evaluations, the predicting models with ImpactScale more accurately predicted faults than did the models without ImpactScale. By adding ImpactScale to existing product metrics, the number of faults detected in the first 10% of the total LOC increased by 50% or more. We also compared ImpactScale with network measures. The predictive performance of ImpactScale was comparable with over 50 network measures, and adding ImpactScale enhanced the model with a set of network measures and existing product metrics. These results support the effectiveness of a new metric, ImpactScale.

We have already applied ImpactScale to our customers' enterprise systems to support quality assurance by using fault prediction. In one case, we performed extra review and inspection of modules in order of predicted fault density with the effort equivalent to 1/20 of annual man-hours in a system. In the result, eight faults were found, and system failures were prevented in advance (the system failed several times per year).

For future work, we would like to prove the effectiveness of ImpactScale on other languages such as Java. When we designed ImpactScale, we intended that it would be a predictor for the effort and the decay of software structure. Thus, we would like to apply ImpactScale to estimate them.

ACKNOWLEDGEMENT

We thank Prof. Kusumoto, Dr. Matsushita, and members of Software Engineering Laboratory of Osaka University for their advice. We appreciate valuable discussions with members of Empirical Software Engineering Group of NICTA (National ICT Australia). We are grateful to Mr. Kamakura and members of Cloud Application Center of Fujitsu for providing the data sets and their support.

REFERENCES

[1] Basili, V. R., Briand, L. C., and Melo, W. L., "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.* 22, 10, pp.751-761, 1996.

[2] Briand, L. C., Wüst, J., Daly, J. W., and Porter, D. V., "Exploring the relationships between design measures and software quality in object-oriented systems," *J. Syst. Softw.* 51, 3, pp.245-273, 2000.

[3] Fenton, N. E., and Ohlsson, N., "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.* 26, 8, pp.797-814, 2000.

[4] Ostrand, T. J., and Weyuker, E. J., "The distribution of faults in a large industrial software system," *ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*. ISSTA, pp.55-64, 2002.

[5] Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H., "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.* 26, 7, pp.653-661, 2000.

[6] Nagappan, N., and Ball, T., "Use of relative code churn measures to predict system defect density," *Int'l Conf. on Softw. Eng.* ICSE, pp.284-292, 2005.

[7] Gall, H., Hajek, K., and Jazayeri, M., "Detection of logical coupling based on product release history," *IEEE Int'l Conf. on Softw. Maint.* ICSM, pp.190-198, 1998.

[8] Hassan, A. E., and Holt, R. C., "Predicting change propagation in software systems," *IEEE Int'l Conf. on Softw. Maint.* ICSM, pp.284-293, 2004.

[9] Haney, F.M., "Module connection analysis – a tools for scheduling software debugging activities," *Fall Joint Computer Conference*, pp.173-180, 1972.

[10] Cataldo, M., Mockus, A., Roberts, J. A., and Herbsleb, J. D., "Software dependencies, work dependencies, and their impact on failures," *IEEE Trans. Softw. Eng.* 36, 2, pp.864-878, 2009.

[11] Zimmermann, T., and Nagappan, N., "Predicting defects using network analysis on dependency graphs," *Int'l Conf. on Softw. Eng.* ICSE, pp.531-540, 2008.

[12] Bohner, S. A., and Arnold, R. S. (Eds.), "*Software change impact analysis*," Bohner, S. A. and Arnold, R. S., "An introduction to software change impact analysis," IEEE Computer Society Press, pp.1-26, 1996.

[13] Grove,D., and Chambers,C., "A framework for call graph construction algorithms," *ACM Trans. Program. Lang. Syst.* 23, 6, pp.685-746, 2001.

[14] Law, J., and Rothermel, G., "Whole program path-based dynamic impact analysis," *Int'l Conf. on Softw. Eng.* ICSE, pp.308-318, 2003.

[15] Ren, X., Shah, F., Tip, F., Ryder, B. G., and Chesley, O., "Chianti: a tool for change impact analysis of Java programs," *Conf. on Object-Oriented Prog., Syst., Lang., and App.* OOPSLA, pp.432-448, 2004.

[16] Sharafat, A. R., and Tahvildari, L., "A probabilistic approach to predict changes in object-oriented software systems," *European Conf. on Softw. Maint. and Reeng.* CMSR, pp.27-38, 2007.

[17] Tsantalis, N., Chatzigeorgiou, A., and Stephanides, G., "Predicting the probability of change in object-oriented systems," *IEEE Trans. Softw. Eng.* 31, 7, pp.601-614, 2005.

[18] Whaley, J., and Lam, M. S., "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," *ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.* PLDI, pp.131-144, 2004.

[19] Chatterjee, S. and Hadi, A. S., "Regression analysis by example, 4th Edition," John Wiley and Sons, 2006.

[20] Arisholm, E. and Briand, L. C., "Predicting fault-prone components in a Java legacy system," *ACM/IEEE Int'l Symp. on Empirical Software Engineering*, ISESE, pp.8-17, 2006.

[21] Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y. and Bener, A., "Defect prediction from static code features: current results, limitations, new approaches," *J. Autom. Softw. Eng.*, Vol.17, pp.375-407, 2010.

[22] Mende, T. and Koschke, R., "Effort-aware defect prediction models," *European Conf. on Softw. Maint. and Reeng.*, CSMR, pp.107-116, 2010.

[23] Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K.., Adams, B. and Hassan, A. E., "Revisiting common bug prediction findings using effort-aware models," *IEEE Int'l Conf. on Softw. Maint.*, ICSM, pp.1-10, 2010.

[24] Cameron, A. C., and Trivedi, P. K., "Regression analysis of count data," Cambridge University Press, 1998.

[25] Khoshgoftaar, T. M., Geleyn, E., and Gao, K., "An empirical study of the impact of count models predictions on module-order models," *IEEE Int'l Softw. Metrics Symp*. METRICS, pp.161-172, 2002.

[26] Tosun, A., Turhan, B. and Bener, A., "Validation of network measures as indicators of defective modules in software systems," *Int'l Conf. on Predictor Models in Softw. Eng.*, PROMISE, pp.5:1-5:9, 2009.

[27] Nguyen, T., Adams, B. and Hassan, A., "Studying the impact of dependency network measures on software quality," *IEEE Int'l Conf. on Softw. Maint*. ICSM, pp.1-10, 2010.

[28] Borgatti, S.P., Everett, M.G. and Freeman, L.C., "UCINet for Windows: software for social network analysis," Analytic Technologies, 2002.

[29] Hanneman, R. A. and Riddle, M., "Introduction to social network methods," University of California, Riverside, 2005.

[30] Jolliffe, I. T., "Principal component analysis, 2ed," Springer, 2002.

[31] Chidamber, S. R., and Kemerer, C. K., "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.* 20, 6, pp.476-493, 1994.

[32] Geiger, R., Fluri, B., Gall, H., and Pinzger, M., "Relation of code clones and change couplings," *Fundamental Approaches to Softw. Eng., LNCS* 3922, Springer, pp.411-425, 2006.

[33] Inoue, K., Yokomori, R., Fujiwara, H., Yamamoto, T., Matsushita, M., and Kusumoto, S., "Component Rank: relative significance rank for software component search," *Int'l Conf. on Softw. Eng.* ICSE, pp.14-24, 2003.

[34] Hayase, Y., Matsushita, M., Kusumoto, S., Inoue, K., Kobayashi, K., and Yoshino, T., "A metric for estimating maintenance effort based on change impact analysis," *IEICE Transactions on Information and Systems* 90, 10, pp.2736-2745, 2007.