An Empirical Study of Fault Prediction with Code Clone Metrics

Yasutaka Kamei*, Hiroki Sato†§, Akito Monden†, Shinji Kawaguchi†¶,
Hidetake Uwano‡, Masataka Nagura†§, Ken-ichi Matsumoto† and Naoyasu Ubayashi*
*Graduate School and Faculty of Information Science and Electrical Engineering, Kyushu University
†Graduate School of Information Science, Nara Institute of Science and Technology
‡Department of Information Engineering, Nara National College of Technology
§Currently, Hitachi, Ltd., ¶ Currently, Japan Manned Space Systems Corporation

***Abstract*—In this paper, we present a replicated study to predict fault-prone modules with code clone metrics to follow Baba's experiment [1]. We empirically evaluated the performance of fault prediction models with clone metrics using 3 datasets from the Eclipse project and compared it to fault prediction without clone metrics. Contrary to the original Baba's experiment, we could not significantly support the effect of clone metrics, i.e., the result showed that F1-measure of fault prediction was not improved by adding clone metrics to the prediction model. To explain this result, this paper analyzed the relationship between clone metrics and fault density. The result suggested that clone metrics were effective in fault prediction for large modules but not for small modules.**

***Keywords*-Code clone, Fault prediction, Product metrics, Empirical study, Logistic regression analysis**

I. Introduction

Recent code clone researches[1] have investigated many different software related issues, from software maintenance [2], [17] to software license violation [21] as research motivation shifts from *"how do we effectively detect code clones from a large software system? [13], [18]"* to *"how do we make use of the detected code clones? [4], [24]"*.

A useful application of code clone in software testing/maintenance is to predict software quality with clones. Baba *et al.* [1] has applied code clone metrics to a fault-prone module prediction model [3], [22] . Through a case study using 32 modules collected from one project, they showed that the improvements of recall and precision were 0.125 and 0.097 compared to a prediction model without clone metrics.

However, the performance of a prediction model often depends on a dataset being used such as development domain and development size [9]. To improve the generality of experimental result of Baba's study, it is preferable to conduct a replicated experiment using other datasets. To our knowledge, no study has reported the replicated experiment of fault prediction with clone metrics.

We evaluated prediction performance using module datasets from three versions (v3.0, v3.1 and v3.2) in Eclipse project. However, unlike the result in Baba's study, we could not obtain the improvement of the prediction performance (Section III). In this paper, we analyze a relationship between clone metrics and bug density to understand why no clone metrics could improve the prediction performance (Section IV).

[1]A code clone is a code portion in source files that is identical or similar to another [14]

Table 1
EVALUATION SETTINGS

	Baba *et al.* [1]	This study
Granularity	Component (a set of files)	File
Modeling approach	Logistic regression	Logistic regression
# of modules	40, 32	8,313, 9,663, 11,525
% of faulty modules	About 80.0%	17.2%, 18.6%, 18.4%
Clone metrics	2 metrics	5 metrics

This paper provides the following contributions:

- We conducted a replicated experiment of the original Baba's study using a large open source system. The result showed that clone metrics did not improve the performance of a fault prediction model.
- Through the detailed analysis of the results, we found that clone metrics are effective for fault prediction at the component-level, but not effective at the file-level.

The remainder of this paper is organized as follows. Section II surveys related work. Section III describes our study design and the results of replication. Section IV presents an analysis to figure out why we could not obtain the improvement of prediction performance. Section V presents the threats to validity. Finally, Section VI concludes the paper.

II. Related Work

In this section, we go through previous works related to fault prediction models and the impact of code clones on software reliability.

A. Predicting fault modules in software systems

Many fault prediction models have been proposed in literatures [8], [19], [23], [26]. Test managers and quality managers identify faulty modules using a prediction model, then allocate more test efforts to the modules considered to be fault-prone.

Nagappan and Ball [23] used relative code churn metrics, which measure the amount of code change, to predict fault density. They showed that process metrics (i.e., relative code churn metrics) are better fault predictors than product metrics such as McCabe's cyclomatic complexity. Zimmermann and Nagappan [26] introduced the use of network analysis on dependency graphs for fault prediction. They showed that network metrics could identify 60% of the files developers considered as critical. Mizuno and Kikuno [19] applied a generic text discriminator to predict faults. The result of their experiment showed that their approach could classify 78% of the actual faulty modules as fault-prone. To our knowledge, there is no study on fault prediction models using clone metrics except Baba's study [1].

B. *Impact of code clones on software reliability*

There are several studies that analyzed whether or not code clones are harmful to software systems in terms of software reliability.

Monden *et al.* [20] has tried to clarify the relationship between code clones and software reliability. The experimental result using an industry system showed that clone-included modules are likely to be more reliable (less faulty) than non-clone modules. The result also showed that the modules including very large code clones ($200 \leq$ SLOC) are less reliable than non-clone modules. Bettenburg *et al.* [4] performed an empirical study on the effect of inconsistent changes to clones at the release level. They showed that only 1% to 3% of the inconsistent changes introduced faults. Selim *et al.* [24] studied the impact of code clones on software faults using survival analysis. Using two open source systems, they showed that clone-included modules are not always more faulty than non-clone modules. In contrast to those studies, we make use of clone metrics to build a fault prediction model.

III. Evaluation Experiment

A. *Overview*

To replicate the experiment conducted by Baba *et al.* [1], we empirically evaluate the performance of a fault prediction model with conventional metrics (i.e., product metrics) and clone metrics. Table 1 shows evaluation settings of Baba's experiment and our experiment. This study uses a source file as a module and logistic regression analysis to build a fault prediction model.

B. *Dataset*

1) Target Project: A target of our study is the Eclipse software system, one of the well-known open development platforms. We collected module datasets from three versions (v.3.0, v.3.1 and v.3.2). They contained 8,313 modules in version 3.0, 9,663 modules in version 3.1 and 11,525 modules in version 3.2. The percentages of faulty modules were about 17.2%, 18.6% and 18.4% in these datasets.

Table 2
MEASURED METRICS OF ECLIPSE DATASET

	Metrics name	Definition
Product metrics	SLOC	Source Lines of Codes
	MLOC	LOC executable
	PAR	Number of parameters
	NOF	Number of attributes
	NOM	Number of methods
	NORM	Number of overridden methods
	NSC	Number of children
	NSF	Number of static attributes
	NSM	Number of static methods
	NBD	Nested block depth
	VG	Cyclomatic complexity
	DIT	Depth of Inheritance Tree
	LCOM	Lack of Cohesion of Methods
	WMC	Number of Weighted Methods per Class
	SIX	Specialization Index (NORM+DIT)/NOM
Clone metrics	NOC	Number of code fragments of any clone set in the file
	ROC	Ratio of duplication in the file
	LEN	Number of tokens in the clone set
	NIF	Number of source files including any fragments of the clone set
	McC	Number of conditional branch and iteration structure in the clone set

2) Measured Metrics: For our experiment, we measured product metrics and clone metrics (Table 2). The product metrics measure the static structure of source code such as source lines of codes and McCabe's cyclomatic complexity. The product metrics are measured using the Eclipse Metrics plug-in [7].

We measured two clone metrics Baba *el al.* used (NOC, ROC) and three well-known clone metrics (LEN, NIF, McC) in addition. We used CCFinderX [6] to detect code clones from source codes and to measure clone metrics. We set a threshold value of RNR[2]as 0.5 because previous study suggested that clone sets whose RNR values are less than 0.5 are deemed uninteresting [12].

While NIF and ROC were calculated based on a unit of a file, LEN, NIF and McC were calculated based on a unit of a clone set. Since a source file often includes multiple code clones, we need to calculate representative values (e.g., maximum and median) in each module. We used the maximum value for this study. For example, if a file had clones A, B and C, and clone A had 20 tokens, clone B had 30 tokens, and clone C had 40 tokens, then LEN in the file is considered to be 40.

3) Recovery of bugs: Based on the condition shown in Table 3, we collected bug reports to determine whether or not each module is faulty from Bugzilla[3], which is provided by the developer community of Eclipse. From the bug reports,

[2]RNR is the ratio of non-repeated code sequence in the clone set [12]
[3]https://bugs.eclipse.org/bugs/

Table 5
EXPERIMENT RESULT

	Clone Metrics					Prediction Performance (3.0 → 3.1)		
No.	NOC	ROC	LEN	NIF	McC	Recall	Precision	F1-measure
1						.135	.639	.222
2	o	o				.130	.628	.215
3	o					.128	.625	.212
4		o				.131	.636	.218
5			o	o	o	.136*	.570	.219
6			o			.136*	.642*	.224*
7				o		.133	.570	.216
8					o	.136*	.642*	.224*
9	o	o	o	o	o	.131	.561	.213
	Clone Metrics					Prediction Performance (3.1 → 3.2)		
No.	NOC	ROC	LEN	NIF	McC	Recall	Precision	F1-measure
10						.198	.629	.302
11	o	o				.203*	.627	.306
12	o					.200	.626	.303
13		o				.203*	.627	.306
14			o	o	o	.200	.624	.303
15			o			.200	.632*	.304
16				o		.200	.624	.303
17					o	.199	.631	.303
18	o	o	o	o	o	.203*	.626	.307*

*:Best Performance

Table 3
CONDITION FOR COLLECTING BUG REPORTS

Classification	Eclipse
Product	Platform
Status of faults	Resolved, Verified, Closed
Resolution of faults	Fixed
Severity	Except Enhancement
Priority	All

Table 4
CONFUSION MATRIX

	True class	
Classified as	Faulty	Not Faulty
Faulty	TP	FP
Not Faulty	FN	TN

we obtained module names to associate faults with modules. We also obtained the reported date and the modified date of faults to associate faults with versions.

In this paper, we associate faults, modules and versions by using Gyimothy's approach [11].

First, we associate bug reports with modules. Each bug report contains a patch file, and the name of a fixed module is described in the patch file. Hence, we can associate faults with modules. Then, we associate faults with versions. We use the reported date and the modified date of a bug to figure out which version of a module is faulty. As shown in Figure 1, for example, we consider the module for which a bug is reported between version 3.0 and version 3.1, and the bug is modified between version 3.2 and version 3.3 as a faulty module between version 3.0 and version 3.2.

C. Approach

In this experiment, we perform cross-release prediction of post-release failures. For cross-release prediction, a training dataset is built from a released project in the past, and a test dataset is built from following release.

A logistic regression model outputs a probability, between 0 and 1, for each module. We use a threshold value 0.5, which means that if the output of the logistic regression model is greater than 0.5, the module is classified as faulty module, otherwise, it is classified as not [10], [11].

To evaluate the prediction performance, we employ commonly-used measures: precision, recall and F1-measure [15]. These criteria can be measured from a confusion matrix, as shown in Table 4. A module can be classified as faulty module when it is truly faulty module (true positive, TP); it can be classified as faulty module when actually it is not faulty module (false positive, FP); it can be classified as not faulty module when it is actually faulty module (false negative, FN); or it can be classified as not faulty module when it is truly not faulty module (true negative, TN).

Recall is the ratio of correctly predicted faulty modules to the actual faulty modules ($Recall = \frac{TP}{TP+FN}$) and precision

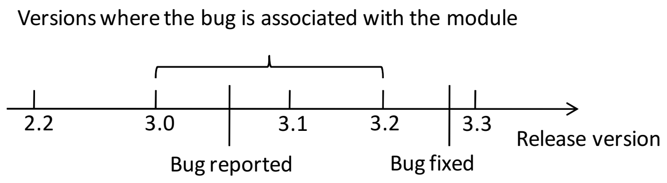

Figure 1. Eclipse version where a bug is associated with a module

is the ratio of actual faulty modules to the modules predicted as faulty ($Precision = \frac{TP}{TP+FP}$). F1-measure is the harmonic mean of recall and precision and is used to provide an overall measure ($F1measure = \frac{2 \times Recall \times Precision}{Recall + Precision}$).

D. Result

We present the result in Table 5. A "o" symbol in the column "Clone Metrics" indicates which clone metrics are used to build a prediction model. Each row from No.1 to No.9 shows the results in Eclipse v3.0 $\rightarrow$ v3.1 and each row from No.10 to No.18 shows the results in Eclipse v3.1 $\rightarrow$ v3.2. In No.1 and No.10, we use only product metrics to build the prediction model. That is, this result means a baseline. The result using product metrics and clone metrics Baba *et al.* used is presented in No.2-4 and No.11-13. The result of our extended work is presented in No.5-9 and No.14-18.

The result showed that the performances of prediction models using LEN (No.6) and McC (No.8) were the best in Eclipse v3.0 $\rightarrow$ v3.1. In Eclipse v3.1 $\rightarrow$ v3.2, the prediction model using all the clone metrics (No.18) showed the best performance. However, the improvements of F1-measure in each version were only 0.002 and 0.005 compared to No.1 and No.10. This result suggests that clone metrics would have no effect on the improvement of the performance in fault prediction.

No clone metrics could improve the performance of a fault prediction model.

IV. Analysis

A. Overview

We compare relationships between clone metrics and bug (fault) density among versions to understand why there is no effect of clone metrics on fault prediction.

The reason why we use bug density, not an existence of a fault (faulty or not), is that we would like to remove spurious relationship among source lines of code (SLOC), clone metrics and the existence of a fault. In general, the larger the SLOC, the larger the value of clone metrics and the more likely a fault is introduced in a module. Even if there is no actual relationship between clone metrics and the existence of a fault, high correlation value could happen. Therefore, we use bug density, which is normalized by dividing the number of bugs by SLOC.

Dealing with size: SLOC slightly has an effect on bug density. The modules is likely to have larger bug density when it has more lines of code [16]. We classify and analyze modules with sizes.

Dealing with imbalance: Our data sets are relatively imbalanced, i.e., there exists a large difference between the number of faulty modules and non-faulty modules. Many modules in our data sets show 0 as bug density. Therefore, it is difficult to analyze a relationship between clone metrics and bug density at fine-grained granularity (i.e., file-level). To deal with this issue of data imbalance, we lift fine-grained modules up to the course-grained. We classify clone metrics into some class values. Then we bind modules that have same class value of clone metrics together into one course-grained module and analyze a relationship between bug density and clone metrics at the course-grained granularity.

B. Approach

We calculate bug density as follows:

Step 1. Classification by SLOC: We classify modules into three parts: a small size (SLOC < 100), a middle size ($100 \leq$ SLOC < 500) and a large size ($500 \leq$ SLOC).

Step 2. Classification by clone metrics: We classify the modules categorized in Step 1 into some classes based on clone metrics.

Step 3. Measurement of bug density: We calculate bug density by dividing sum of the number of bugs by sum of SLOC in each course-grained module.

C. Result and Discussion

Figure 2 shows the relationship between clone metrics and bug density. X-axis indicates clone metrics and y-axis indicates bug density. The top part of this figure shows the result for a small size, the middle part shows for a middle size and the bottom part shows for a large size.

Version 3.0: We found that relationships between clone metrics and bug density varied among three module sizes. For example, for LEN, the bug density of the small size [a-1] weakly demonstrates an upward tendency (Spearman's correlation value was 0.36). On the other hand, the bug density of the large size [a-3] strongly demonstrates a downward tendency (the correlation values was -0.89). The results for NIF and McC showed a somewhat similar tendency as LEN. For NIF, although the small size [b-1] had the upward tendency (0.83), the large size [b-3] had the downward tendency (-0.71). For McC, the middle size [c-2] had little correlation (0.14), but the large size [c-3] had the downward tendency (-0.71).

This finding explains the reason why we could not significantly support the effect of the clone metrics for fault prediction. Since the relationships between clone metrics and bug density varies among modules sizes, the performance of the prediction model would not improve because fault prediction models were built from a dataset including all the modules.

Version 3.1: Similar to the analysis result of the version 3.0, the relationships between clone metrics and bug density were different among three module sizes. For example, for LEN, although the small size [a-1] had little correlation (-0.17), the large size [a-3] strongly had the downward tendency (-0.91).

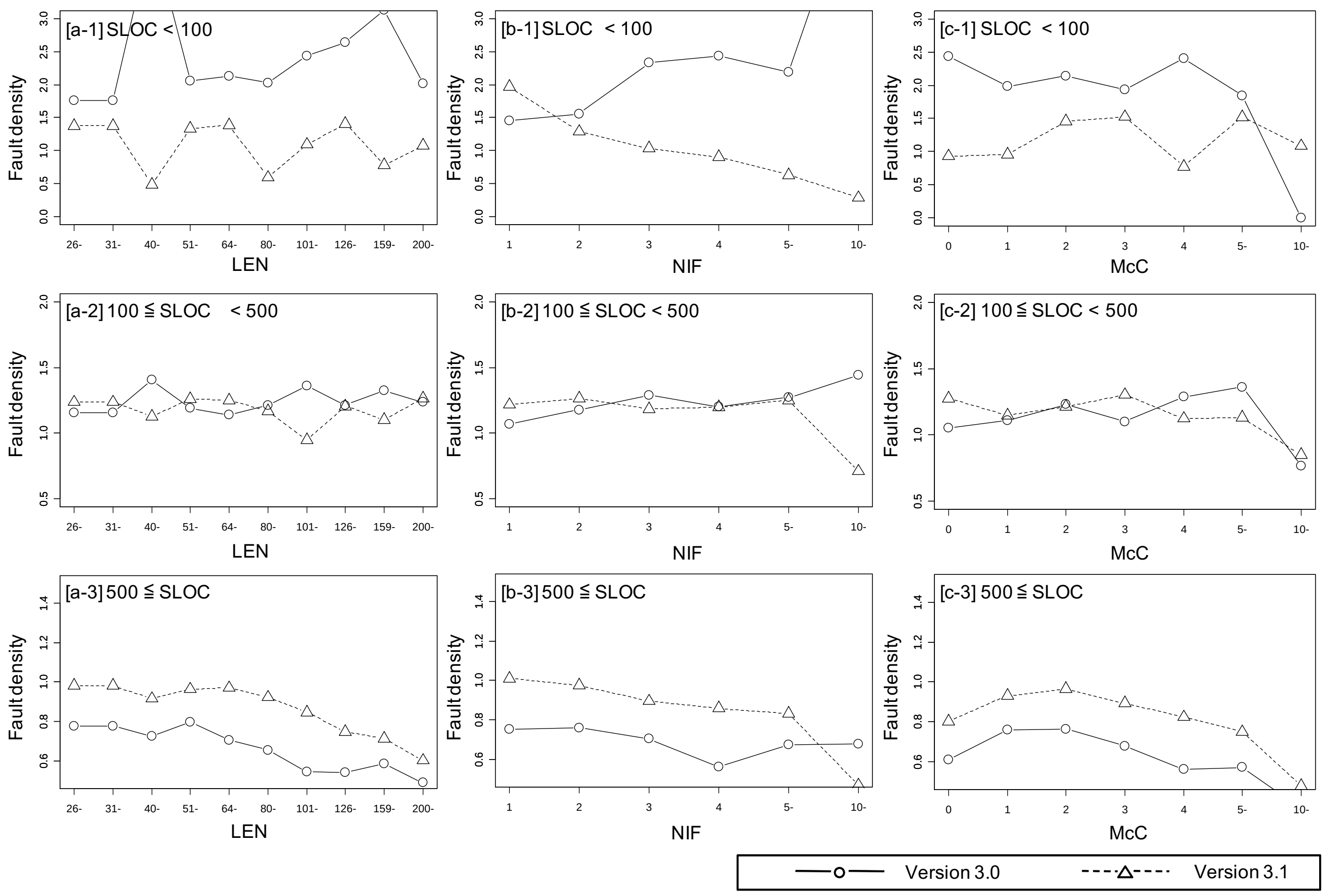

Figure 2. Relationship between clone metrics and bug density

Comparison between version 3.0 and version 3.1: In small size modules, we found that relationships between clone metrics and bug density were different between version 3.0 and version 3.1. For example, for LEN, the bug density in version 3.0 weakly had an upward tendency (0.36), but the version 3.1 had little correlation (-0.17). For NIF, contrary to the result of version 3.0 (0.83), the version 3.1 had negative correlation (-1.00).

This finding also explains why clone metrics are not effective to improve the prediction performance.

On the other hand, for the large size modules, relationships between clone metrics and bug density showed the same tendency in version 3.0 and version 3.1[4]. For example, the correlation values for version 3.0 and version 3.1 for LEN were -0.89 and -0.91 respectively. This suggests why Baba's experiment could improve the prediction performance by clone metrics since Baba used component level (i.e. large size) modules to build a prediction model. These results indicate that clone metrics are effective for a fault prediction model for large size modules such as component level but not for small size (e.g., SLOC $<$ 500).

The relationships between clone metrics and bug density were different among module sizes as well as among versions. This is the reason why clone metrics showed no effect on fault prediction. However, further analyses showed that clone metrics could improve the prediction performance for large size modules.

V. Threats to validity

In this section, we discuss the threats to the validity of our work. This study detected code clones from modules using CCFinderX, because this tool was commonly used in software reliability researches [4] and [24]. Replicated studies using other clone detection tools (e.g., CP-Miner [18] and DECKARD [13]) would be preferable to improve the generality of our findings.

We used a dataset collected from one foundation (Eclipse). Although this project is large and long-lived, this project might not be the representative of all projects

[4]The result for version 3.2 also showed a similar tendency as version 3.0 and version 3.1.

out there. For example, the studied project is implemented mainly in Java, therefore, our results may not generalize to other commercial or open source projects in other programing languages (e.g., functional programming languages). However, we believe that our work has a significant contribution to the validation of the empirical knowledge.

This study determines whether or not each module is faulty using Gyimothy's approach [11]. The approach has the limitation in collecting the unreported faults in Bugzilla. In the future, we plan to use other approaches (e.g., SZZ algorithm [25]) that make use of a version archive (such as CVS) together.

We used the logistic regression analysis to evaluate the performance of the fault prediction models, since this modeling techniques is well-known for fault prediction [3], [5]. However, using other modeling techniques may produce different results. We also used the maximum values of clone metrics (LEN, NIF, McC) as representatives of code clones in a module. Using other representative values (e.g., mean and median) may lead to different results.

VI. Conclusion

In this paper, we empirically evaluated the effects of clone metrics to fault prediction models using a dataset collected from three versions of a large open source software project. The result suggested that clone metrics would not improve the prediction performance. To explain this result, we analyzed the relationship between clone metrics and bug density. Our major finding from the detailed analysis is that clone metrics have an effect in fault prediction for large modules (e.g, component-level) but not for small modules (e.g., file-level).

In the future, we are planning to perform case studies with other open source software systems from other domains. We also consider to analyze modules of different granularity such as component-level and method-level as a prediction target.

References

[1] S. Baba, N. Yoshida, S. Kusumoto, and K. Inoue, "Application of code clone information to fault-prone module prediction," *IEICE Transactions on Information and Systems*, vol. 91-D, no. 10, pp. 2559–2561, 2008 (in Japanese).

[2] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proc. Working Conference on Reverse Engineering (WCRE'95)*, 1995, pp. 86–95.

[3] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, 1996.

[4] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at release level," in *Proc. Working Conference on Reverse Engineering (WCRE'09)*, 2009, pp. 85–94.

[5] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 864–878, 2009.

[6] CCFinderX, http://www.ccfinder.net/index.html, last viewed: 05-May-2011.

[7] Frank Sauer, "Eclipse Metrics plugin," http://sourceforge.net/projects/metrics, last viewed: 05-May-2011.

[8] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, 2000.

[9] A. R. Gray and S. G. MacDonell, "Software metrics data analysis – exploring the relative performance of some commonly used modeling techniques," *Empirical Software Engineering*, vol. 4, no. 4, pp. 297–316, 1999.

[10] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in *Proc. Int'l Conf. on Softw. Eng. (ICSE'10)*, 2010, pp. 495–504.

[11] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, 2005.

[12] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Method and implementation for investigating code clones in a software system," *Inf. Softw. Technol.*, vol. 49, pp. 985–998, 2007.

[13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. Int'l Conf. on Softw. Eng. (ICSE'07)*, 2007, pp. 96–105.

[14] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 654–670, 2002.

[15] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, 2008.

[16] 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. Softw. Eng.*, vol. 35, no. 2, pp. 293–304, 2009.

[17] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process," in *Proc. Int'l Conf. on Software Maintenance (ICSM'97)*, 1997, pp. 314–321.

[18] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng.*, vol. 32, pp. 176–192, 2006.

[19] O. Mizuno and T. Kikuno, "Prediction of fault-prone software modules using a generic text discriminator," *IEICE Transactions on Information and Systems*, vol. E91-D, no. 4, pp. 888–896, 2008.

[20] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, “Software quality analysis by code clones in industrial legacy software,” in *Proc. Int’l Symposium on Software Metrics (METRICS’02)*, 2002, pp. 87–94.

[21] A. Monden, S. Okahara, Y. Manabe, and K. Matsumoto, “Guilty or not guilty: Using clone metrics to determine open source licensing violations,” *IEEE Softw.*, vol. 28, pp. 42–47, 2011.

[22] J. C. Munson and T. M. Khoshgoftaar, “The detection of fault-prone programs,” *IEEE Trans. Softw. Eng.*, vol. 18, no. 5, pp. 423–433, 1992.

[23] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Proc. 27th Int’l Conference on Software Engineering (ICSE’06)*, 2005, pp. 284–292.

[24] G. Selim, L. Barbour, W. Shang, B. Adams, A. Hassan, and Y. Zou, “Studying the impact of clones on software defects,” in *Proc. Working Conference on Reverse Engineering (WCRE’10)*, oct. 2010, pp. 13–21.

[25] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?” in *Proc. Int’l Conference on Mining Software Repositories (MSR’05)*, 2005, pp. 1–5.

[26] T. Zimmermann and N. Nagappan, “Predicting defects using network analysis on dependency graphs,” in *Proc. Int’l Conference on Software Engineering (ICSE’08)*, 2008, pp. 531–540.