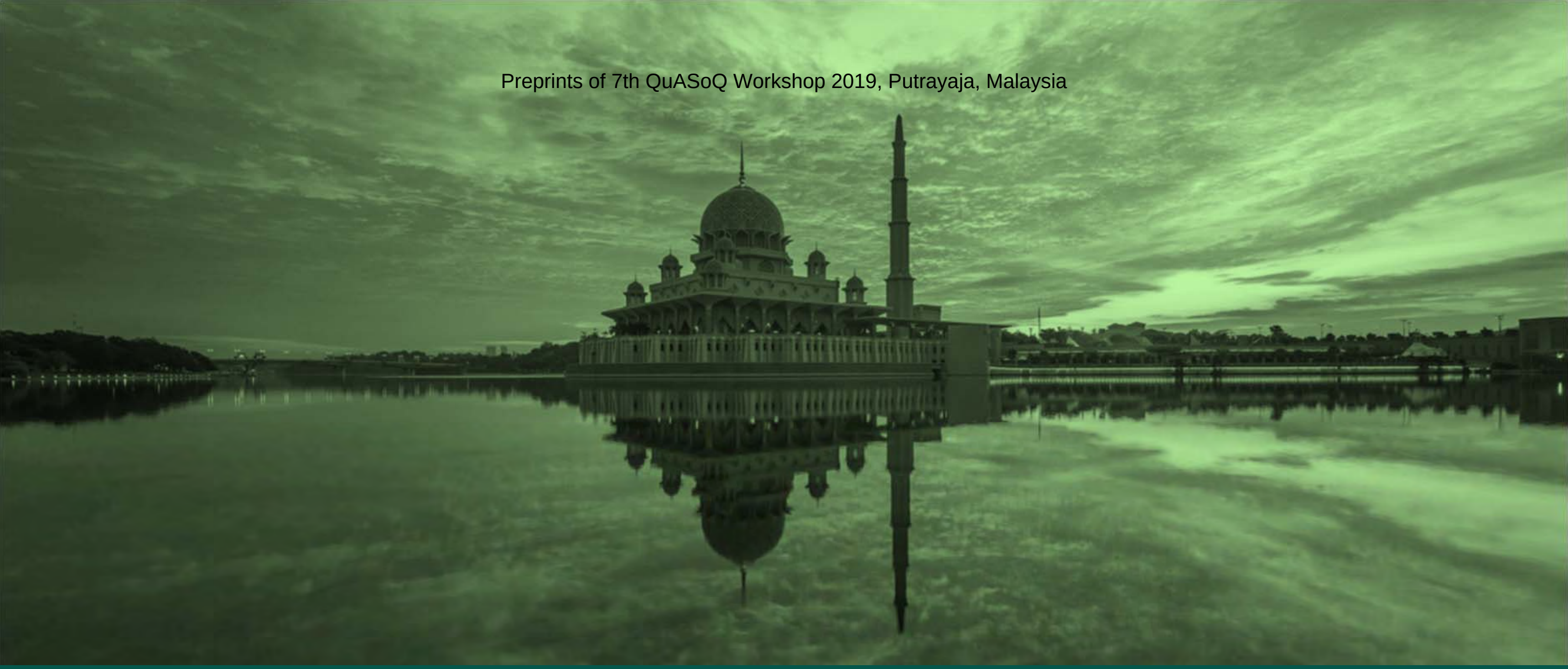


QuASoQ 2019

7th International Workshop on
Quantitative Approaches to Software Quality

co-located with APSEC 2019
Putrajaya, Malaysia, December 2nd , 2019

Editors:

Horst Lichter, RWTH Aachen University, Germany
Thanwadee Sunetnanta, Mahidol University, Thailand
Toni Anwar, University Petronas, Malaysia

Empirical Study of Fault Introduction Focusing on the Similarity among Local Variable Names

Hirohisa Aman
Center for Information Technology
Ehime University
Matsuyama, Ehime, Japan
aman@ehime-u.ac.jp

Sousuke Amasaki
Faculty of Computer Sc. & Systems Eng.
Okayama Prefectural University
Soja, Okayama, Japan

Tomoyuki Yokogawa
Faculty of Computer Sc. & Systems Eng.
Okayama Prefectural University
Soja, Okayama, Japan

Minoru Kawahara
Center for Information Technology
Ehime University
Matsuyama, Ehime, Japan

***Abstract*—Well-chosen variable names play significant roles in program comprehension and high-quality software development and maintenance. However, even though all variables have easy-to-understand names, attention needs to be paid to the similarity among those names as well because a highly similar pair of variables may decrease the readability of code and might cause a fault. In order to analyze the relationship of variable similarity with the risk of introducing faults, this paper collects variable data from ten Java open source development projects and conducts an empirical study on the following three research questions: (1) The distribution of similarity differs among software development projects?, (2) What is the appropriate threshold of similarity?, and (3) How can the threshold of similarity contribute to the fault-prone method prediction? Then, the empirical results show the following findings: (1) The distribution of similarity is nearly identical regardless of the project; (2) Programmers should avoid giving similar names to different variables to prevent a fault introduction, and the threshold of Levenshtein similarity is 0.35; (3) By classifying Java methods with the above threshold, the risk of overlooking fault-introducing events in the fault-prone prediction model is effectively reduced; The recall of the prediction model improves by 12.8% on average.**

I. Introduction

Code changes form essential parts of software development and maintenance, i.e., drive a software evolution. On the other hand, a code change also has a risk of introducing a new fault [1]. Because the code change is a human intellectual activity, the risk of a fault introduction would become higher when the program is more complex and harder to understand. Hence, the readability of code plays a significant role in the successful development and maintenance [2]. By making the code readable, the programmer can review the code more clearly and deeply, and may quickly detect a potential fault if it exists. Moreover, readable code is easy-to-understand for many other programmers as well, and they can review and maintain the source code smoothly. One of the most critical matters for producing readable code is the proper naming of the variables [3]. Indeed, there is a report saying that 24% of code review feedbacks were related to the variable naming [4]. Well-chosen names of variables can be useful clues to the understanding of what the program does [5] and can lead to the advantages of a low maintenance cost [6]. On the other hand, we can readily decrease the readability and understandability of programs by selecting meaningless names for the variables.

There have been various studies on a better naming of variables in the past. For example, Lawrie et al. [7] and Scanniello et al. [8] reported empirical results that fully spelled English words or their abbreviated forms are better for naming variables in terms of program comprehension and effective fault detection. Binkley et al. [9] showed that the camel case-style naming is useful for enhancing the understandability.

Although the naming of variables has been studied, we should consider not only the naming of an individual variable but also the relationship among the names of variables. If two variables have highly similar names to each other, they may be easily confused. For example, suppose a method (or a function) has some local variables including lineIndex and lineIndent. Although each of these two names looks easy-to-understand, a mix of them may be confusable [10]. When a programmer uses a sophisticated editor which can automatically complement or suggest the name of a variable, the programmer might accidentally select a similar but wrong variable [11]. When we use two or more variables, we should give more distinguishable names to the variables while keeping the meaningfulness of the names.

In this paper, we report our quantitative study on the risk of introducing faults by focusing on the similarity among the names of the local variables. The key contributions of this paper are as follows:

- We present a quantitative guideline of the local variable name to prevent the fault introduction: When a programmer declares two or more local variables in a Java method, the names of them should be dissimilar to each other. The threshold of Levenshtein similarity between variable names is 0.35.

- We empirically show that the classification of Java methods by the above threshold can be helpful in the fault introduction prediction; Especially, it reduces the risk of overlooking fault introductions effectively: The classification based on the similarity of local variable names improves the recall, the precision, and the F value of the random forest-based fault introduction prediction model by 12.8%, 2.45%, and 4.2% on average, respectively.

The remainder of this paper is organized as follows: In Section II, we describe the related work regarding variable names and present our research motivation and research questions (RQs). Then, in Section III, we report the empirical work that we conducted on the RQs. Finally, we give our conclusion and future work in Section IV.

II. Related Work and Motivation

In this section, we briefly explain the related work focusing on the names of local variables. Then, we describe our research motivation and put our research questions (RQs) to clarify our goal in this paper.

A. Naming Variables

There have been several empirical studies regarding better naming of variables in the past.

Lawrie et al. [7] conducted an empirical study focusing on the comprehensibility of variables from the perspective of the naming form. They prepared three variations of a variable name as (a) fully spelled English word, (b) an abbreviated form of the word, and (c) a single character then got 128 programmers to compare the ease of comprehension of them. The empirical results showed that the trend of comprehensibility is "(a) $\geq$ (b) $>$ (c)," but there is no statistically significant difference between (a) and (b). That is, a fully spelled English word or a well-chosen abbreviated work would be a better name for a variable than a single character name. Scanniello et al. [8] also performed an empirical study regarding the relationships of the variable naming with the program comprehension and the fault detection/fix. The empirical results involving 100 programmers showed a trend similar to the one reported by Lawrie et al. [7].

When a variable has a more complex role, programmers tend to use a longer and more descriptive name. The major ways of such naming are the concatenations of two or more words (or abbreviated words) by the camel case (for example, indexOfArray) or the snake case (for example, index_of_array) [9].

Although the above previous studies are remarkable empirical work to discuss the appropriate naming of variables, their main focus is on the naming of an *individual* variable. In other words, the *relationship* between names has not been well-discussed. When we see two or more variables and the names of them are similar to each other, we might get confused by the similar names even if each variable has an easy-to-understand name.

B. Similar Names of Variables

As mentioned above, when there are two or more local variables and the names are highly similar to each other, the code comprehensibility and the readability may be low even if each of the names is easy-to-understand. Binkley et al. [11] concerned the risk of selecting the wrong variable when the name is long. Indeed, many programmers may use an advanced code editor like Atom[1] or an integrated development environment like Eclipse[2], which can automatically complete the name of the variable or line up available variables [12]. Then, a programmer might overlook a wrong completion or choose the wrong candidate (see Fig. 1).

Tashima et al. [10] analyzed the issue of similar variable names from the perspective of the fault-proneness. In work by Tashima et al., they measured the similarity between two names by the Levenshtein distance [13]. We briefly describe their approach below. The Levenshtein distance between two names is defined to be the minimum number of operations required to change one name to the other name. The available operations are the following three ones:

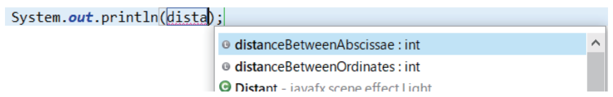
- to add one character,
- to delete one character, and
- to replace one character by another character.

For example, we can produce lineIndent from lineIndex through the following two operations: 1) replace x (in lineIndex) by n, and 2) add t to the end of it. That is, the Levenshtein distance between the above two names is two. The smaller the Levenshtein distance, the higher the similarity between the names. However, the length of the name has also an impact on the similarity assessment. For example, although the Levenshtein distance between file and pipe is also two, the similarity of (file, pipe) does not look at as the same as the similarity of (lineIndex, lineIndent). Hence, Tashima et al. used the following normalized Levenshtein distance (*NLD*) between two names, s_1 and s_2, in their work:

$$NLD(s_1, s_2) = \frac{LD(s_1, s_2)}{\max\{\lambda(s_1), \lambda(s_2)\}}, \qquad (1)$$

where $LD(s_1, s_2)$ is the Levenshtein distance between s_1 and s_2, and $\lambda(s_i)$ is the length (character count) of s_i (for $i = 1, 2$).

Tashima et al. conducted an empirical study using Java programs and showed that more fault fixes tend to occur in Java methods which have a pair of local variables with highly similar names [10]. Although their study is a useful previous work which drew attention to the similarity of the variable names, there are the following two problems to be solved.



```
System.out.println(dista);
```

distanceBetweenAbscissae : int
distanceBetweenOrdinates : int
Distant - javafx.scene.effect.Light

Fig. 1. Example of code completion which may cause a wrong selection.

[1]https://atom.io/
[2]https://www.eclipse.org/

1) *The fault introduction events were missed*: The previous work used a snapshot when a fault was fixed. It is better to focus on the fault introduction events rather than the fault fix events for discussing the relationship between the similarity of variable names and the fault-proneness.
2) *The threshold of the similarity was not discussed*: The previous work did not discuss the threshold of the similarity. To construct the guideline about the similarity, we need to find an appropriate threshold value of similarity.

These two problems are our research motivations in this paper.

C. Research Questions

To address the two problems mentioned in Section II-B, we consider the following three research questions (RQs). In this study, we use the following Levenshtein similarity (LS),

$$LS(s_1, s_2) = 1 - NLD(s_1, s_2) \ , \qquad (2)$$

as our similarity measure because *NLD* is an inverse measure of similarity.

RQ1: *The distribution of similarity differs among software development projects?*
Because different projects may have different developers, the trends of naming variables might also differ among projects. To discuss an appropriate threshold value of the similarity between variable names, we first need to check whether the difference of projects affects the distribution of similarity or not. If there is a significant variation in the distributions, we have to study the appropriate threshold value for each project separately. Otherwise, we can discuss the standard threshold value, which can be commonly used for all projects in this study.

RQ2: *What is the appropriate threshold of similarity?*
If there are two or more local variables whose names are highly similar to each other in a Java method, they are confusable and may adversely affect the code quality of the method. Although Tashima et al. [10] empirically showed that the presence of such a confusing pair of local variables in a Java method is related to the fault-proneness of the method, they did not discuss the appropriate threshold value of the similarity between variable names. Thus, we explore the appropriate threshold in this study and present the result as a guideline about the variable naming.

RQ3: *How can the threshold of similarity contribute to the fault-prone method prediction?*
Once we obtain an appropriate threshold value of the similarity between variable names through the empirical study regarding RQ2, then we can classify Java methods by the threshold. Thus, we examine the usefulness of the classification by the threshold of similarity. More specifically, we build fault-prone method prediction models with and without using the above threshold and compare the prediction performance values between them to evaluate the usefulness of the classification.

We conduct an empirical study on the above three RQs in the following section.

III. Empirical Study

In this section, we report the empirical study that we conducted to address the three RQs mentioned above. First, we describe our aim and data source in Section III-A and explain the procedure of our data collection and analysis in Section III-B. Then, we show our results in Section III-C and give our discussions about the results in Section III-D. Finally, we describe our threats to validity in Section III-E.

A. Aim and Data Source

This study aims to tackle the above three RQs through an empirical data analysis. To this end, we collect fine-grained method-level data of code changes from open source development projects and analyze the risk of introducing faults in terms of the similarity among the names of local variables.

We used ten open source development projects as our data source[3] (see Table I). The main reasons why we use these projects are as follows:

1) The code repository is Git;
2) The primary development language is Java;
3) The issue (bug) tracking system is Apache JIRA;
4) The developers specify the corresponding issue IDs in the commit messages when they commit fixed source files into the code repository.

The reasons 1) and 2) aim to perform a lightweight method data collection. In this study, we collect the change history of methods (functions) from a project. To carry out our data collection effectively, we utilize a Git-based fine-grained code repository, Historage [14], [15] which manages the source code at the method level rather than the file level; Git repositories are convertible to Historage repositories. Because the supported language of Historage repository is Java, we focus only on Java projects in this study.

The reasons 2), 3), and 4) are requirements for collecting fault introduction data. To detect fault introduction events, we use the well-known SZZ algorithm [16]–[18] and one of its implementations, SZZ Unleashed [19]. The SZZ algorithm

TABLE I
Studied open source software development projects

Name	Size (KLOC)	#Contributors	SHA-1
Beam	447	406	`c961139`
Flink	709	501	`4973d8a`
Groovy	187	272	`325653a`
HBase	754	217	`36f0929`
Hive	1,386	183	`4c87512`
Kafka	244	527	`fd79dd0`
RocketMQ	86	139	`971fa8e`
Storm	281	294	`0ea3c89`
Zeppelin	119	279	`4219d55`
Zookeeper	119	77	`7256d01`

[3]`https://{beam,flink,hbase,hive,kafka,rocketmq,storm,zeppelin,zookeeper}.apache.org/`, `http://groovy-lang.org/`

links a reported issue (bug) with the corresponding issue-fix commits. The SZZ Unleashed is designed to collect issue data from Apache JIRA and to make the above links to commits on Git. Because the issue-commit linking is based on whether or not the issue ID appears in the commit message, the above reason 4) is a requirement in this study.

We performed a project search on GitHub with the search keyword "org:apache," and selected the most popular[4] ten projects (see Table I) satisfying the above requirements.

B. Procedure

We collect data of Java methods and their local variables from each of the studied projects and analyze the collected data in the following procedure.

1) *Fine-grained repository construction*:
Because local variables belong to a Java method, we need to collect data of local variables from each method. Moreover, to capture the fault introduction events in a method, we have to examine the code change history at the method level rather than the file level. Since the Git repositories of the studied projects maintain the code change history at file-level, we convert the repositories into finer-grained method-level repositories—the Historage repositories [14], [15]—by using the conversion tool, Kenja[5] [20].
2) *Fault introduction commit detection*:
According to the SZZ algorithm [16]–[18], we detect the commits in which the code changes introduced faults; We use the SZZ Unleashed [19], an implementation of the SZZ algorithm. In the above detection, we exclude the following kinds of methods because they are not related to any fault introduction: Java methods for testings, demos, and document generations.
3) *Data collection of the fault-proneness of methods and the names of local variables*:
If a method experiences a fault introduction, we define the method to be *faulty*. For a faulty method, we focus on the commit in which the fault introduction occurred first and extract the corresponding revision of the method. Then, for each pair of local variables (including formal parameters) in the selected revision, we compute the similarity between the names of variables. To link a computed similarity with the method in our analysis, we adopt the highest similarity in a method as the representative value of the method.
When a method is not faulty, we single out the latest revision of the method as a sample for our data analysis and compute the similarity of local variables as well.
We henceforth denote the highest similarity between local variable names in method m by $\mathit{HLS}(m)$.
4) *Data analysis for answering RQ1 (The distribution of similarity differs among software development projects?)*:
We compare the distributions of computed $\mathit{HLS}(\cdot)$ across the studied projects by using the summary statistics—the minimum, the first quartile (25 percentile), the median, the mean, the third quartile (75 percentile), and the maximum—and the box plots. Moreover, we randomly select 140 samples[6] (methods) from each project, i.e., $1400 (= 140 \times 10)$ samples in total, and perform the Kruskal-Wallis test [21] at significance level 0.05 with the following hypotheses.
   - *Null hypothesis*: There is no difference in the median similarity across the studied projects.
   - *Alternative hypothesis*: At least one project's median similarity is different from the others.
5) *Data analysis for answering RQ2 (What is the appropriate threshold of similarity?)*:
Let M be the set of all Java methods to be analyzed. We consider the following two subsets of M, which are divided by a threshold τ:
   - $M_L(\tau) = \{\ m \in M \mid \mathit{HLS}(m) \le \tau\ \}$
   - $M_H(\tau) = \{\ m \in M \mid \mathit{HLS}(m) > \tau\ \}$

   Now we explore the appropriate threshold of similarity (τ^*) as follows.
First, for a τ, we compute the following faulty method rates ($\mathit{FMR}_L(\tau)$ and $\mathit{FMR}_H(\tau)$) in $M_L(\tau)$ and $M_H(\tau)$, respectively:
$$\mathit{FMR}_x(\tau) = \frac{\left|\{m \in M_x(\tau) \mid m \text{ is faulty}\}\right|}{|M_x(\tau)|}\ , \quad (3)$$
where $x \in \{L, H\}$.
Then, we evaluate the effect of the classification by τ, using the following odds ratio $\mathit{OR}(\tau)$:
$$\mathit{OR}(\tau) = \frac{\mathit{FMR}_H(\tau) / \{1 - \mathit{FMR}_H(\tau)\}}{\mathit{FMR}_L(\tau) / \{1 - \mathit{FMR}_L(\tau)\}}\ . \quad (4)$$
The higher $\mathit{OR}(\tau)$ means that the classification by τ is more effective in terms of the fault-prone method detection. We adopt the threshold τ such that $\mathit{OR}(\tau)$ has the highest value, as the appropriate threshold τ^*.
6) *Data analysis for answering RQ3 (How can the threshold of similarity contribute to the fault-prone method prediction?)*:
To examine how the classification by τ^* can contribute to the fault-prone method prediction, we compare the prediction performances of the prediction models with and without using the similarity-based classification. Although various mathematical models have been studied for the fault-prone method prediction in the past, we use the random forest in this study because it has been widely known as one of the most promising models [22], [23].
6a) Examination by random forests without using the similarity-based classification: First, we build a random forest[7] for predicting whether a method is faulty or not,

[4]The popularity is evaluated by the "stars" score on GitHub.
[5]`https://github.com/niyaton/kenja`

[6]We decided the simple size "140" by a simulation using random numbers, where the significance level is 0.05, and the power of a test is 0.8.
[7]We use the randomForest function provided by the randomForest package of R version 3.6.1, with its default settings.

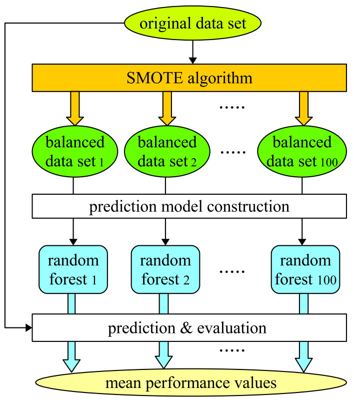


Fig. 2. Summary of the prediction model construction with SMOTE algorithm and the evaluation in step 6a.

by using only the fundamental code metrics: the lines of code (LOC) [24], the cyclomatic complexity (CC) [25], and the number of local variables in a method.
Then, we compute the following performance values: the recall, the precision, and the F value (the harmonic mean of the recall and the precision). These values form the baselines of evaluation in our study.
During the prediction model construction, we have to pay attention to the imbalance between positive samples and negative samples: we have less number of faulty methods than non-faulty ones in our data sets. Such an imbalance of data often leads to a poor prediction model. To overcome this issue of imbalanced data, we use SMOTE algorithm [26], which oversamples the minority by generating similar artificial data and undersamples the majority to balance the data set[8]. By using such a balanced data set, we construct the random forest for predicting faulty methods. Because the behavior of the SMOTE algorithm depends on the random number, we repeat the following two steps 100 times:

6a-1) construct the prediction model, and
6a-2) compute the performance values.

Then, we adopt the mean performance values as the baselines (see Fig. 2).
6b) Examination by random forests with using the similarity-based classification: Next, by τ^* obtained at the step 5, we divide the set of all methods M into $M_L(\tau^*)$ and $M_H(\tau^*)$. Then, we obtain the mean performance values of the prediction models by repeating the following three steps 100 times (see Fig. 3):

6b-1) construct the prediction models for $M_L(\tau^*)$ and $M_H(\tau^*)$, respectively,
6b-2) integrate the prediction results produced by the two models, and

[8]We use the SMOTE function provided by the DMwR package of R version 3.6.1; To balance the ratio of faulty methods and non-faulty ones as "fifty-fifty" in our data set, we increase the samples of faulty methods by 10-fold through the oversampling and randomly choose (undersample) the same number of non-faulty methods.

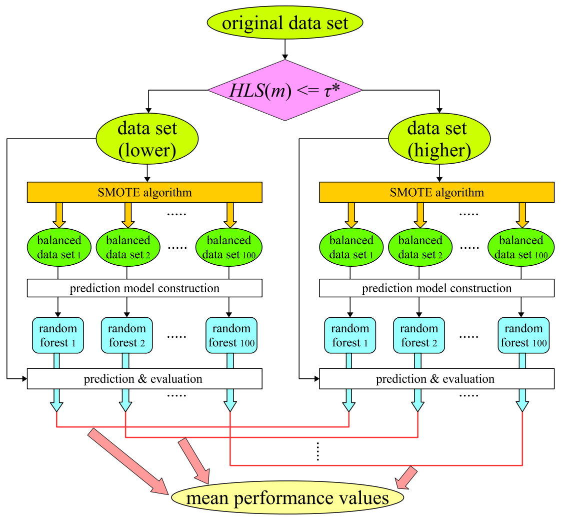


Fig. 3. Summary of the prediction model construction with SMOTE algorithm and the evaluation in step 6b.

6b-3) compute the performance values of the prediction.

6c) Comparison of the prediction performances: Finally, we evaluate the effect of the similarity-based classification by comparing the mean performance values obtained in the step 6a and 6b.

C. Results

By performing the steps 1) – 3) described in Section III-B, we collected 86, 104 methods (including constructors) from the studied ten projects and detected fault introductions by the SZZ algorithm. Notice that the above methods are the ones that have two or more local variables (including formal parameters) because this study focuses on the similarity among the names of local variables. Table II presents the number of methods and that of faulty ones for each project; For example, in Beam project, 63 out of 7, 491 methods are faulty.

We show the results of the steps 4)–6) below, which are corresponding to RQ1, RQ2, and RQ3, respectively. The set of our empirical data is available at http://se.cite.ehime-u.ac.jp/data/QuASoQ2019/.

TABLE II
NUMBER OF ANALYZED METHODS

Project	#Methods	#Faulty methods (faulty rate)
Beam	7, 491	63 (0.8%)
Flink	12, 854	437 (3.4%)
Groovy	9, 926	448 (4.5%)
HBase	13, 170	468 (3.6%)
Hive	28, 313	523 (1.9%)
Kafka	1, 089	30 (2.8%)
RocketMQ	1, 998	21 (1.1%)
Storm	6, 875	68 (1.0%)
Zeppelin	2, 688	249 (9.3%)
Zookeeper	1, 700	88 (5.2%)
Total	86, 104	2, 395 (2.8%)

TABLE III
SUMMARY STATISTICS OF THE SIMILARITY AMONG THE NAMES OF LOCAL VARIABLES IN THE STUDIED PROJECTS

Project	Min.	25%	50%	Mean	75%	Max.
Beam	0	0.143	0.250	0.332	0.500	0.958
Flink	0	0.143	0.286	0.346	0.556	0.967
Groovy	0	0.167	0.286	0.354	0.556	0.960
HBase	0	0.143	0.267	0.336	0.500	0.955
Hive	0	0.167	0.333	0.384	0.600	0.962
Kafka	0	0.125	0.250	0.320	0.500	0.947
RocketMQ	0	0.143	0.289	0.342	0.539	0.938
Storm	0	0.143	0.267	0.340	0.500	0.962
Zeppelin	0	0.165	0.282	0.351	0.500	0.955
Zookeeper	0	0.123	0.250	0.324	0.500	0.952
All	0	0.150	0.286	0.356	0.556	0.967

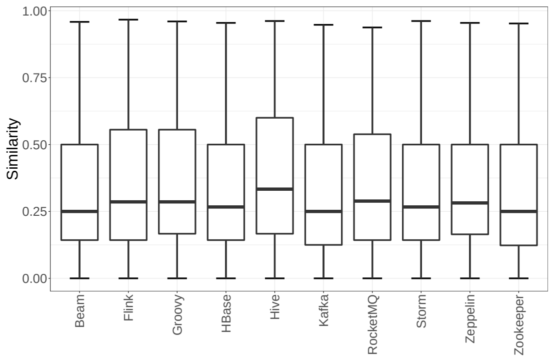


Fig. 4. Boxplot of the similarity among the names of local variables.

Results regarding RQ1 (The distribution of similarity differs among software development projects?)

Table III presents the summary statistics of the similarity by the studied projects, and Fig. 4 shows the box plots of them. From the table and figure, the distributions of similarity in projects seem to be close to each other.

Moreover, the p-value of Kruskal-Wallis test was $0.3387 (> 0.05)$, so we cannot reject the null hypothesis "There is no difference in the median similarity across the studied projects." That is, there does not seem to be a significant difference in the central tendency of similarity distribution among projects.

From the above results, our answer to RQ1 (The distribution of similarity differs among software development projects?) is: *The distribution of similarity is nearly identical regardless of the project.*

Results regarding RQ2 (What is the appropriate threshold of similarity?)

In the data analysis regarding RQ1, we have seen that the distribution of similarity is nearly identical regardless of the project. Hence, we seek an appropriate threshold of similarity between the names of local variables without distinction of the project.

From Table III and Fig. 4, the range of "relatively high similarity" seems to be between 0.3 and 0.6. Thus, by changing the threshold τ from 0.3 to 0.6 at intervals of 0.01, we divided the set of all methods M into $M_L(\tau)$ and $M_H(\tau)$, obtained the faulty method rates $FMR_L(\tau)$ and $FMR_H(\tau)$, and computed the odds ratio of them, $OR(\tau)$, as the measure of effect. We show the change of the odds ratio over τ in Fig. 5.

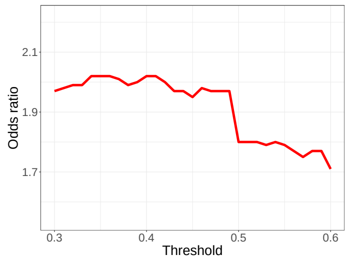


Fig. 5. Changes of the odds ratio over τ.

In the figure, the odds ratio becomes the highest around $\tau = 0.35$–0.4, and it drops around $\tau = 0.5$. Table IV presents a part of the computed odds ratios (OR), the faulty method rates (FMRs), and the ratio between FMRs; In the table, we highlight the highest odds ratio and the highest ratio between FMRs by boldface together with the "$*$" mark.

In Table IV, six thresholds ($\tau = 0.34, 0.35, 0.36, 0.37, 0.40$, and 0.41) showed the highest odds ratio, i.e., the highest effect of classification by that threshold for detecting fault-prone methods. Because the ratio between faulty method rates also gets the highest value at $\tau = 0.35$ within these six thresholds, we consider it is the appropriate threshold: $\tau^* = 0.35$.

From the above results, our answer to RQ2 (What is the appropriate threshold of similarity?) is: *The appropriate threshold of similarity is around* 0.35.

Results regarding RQ3 (How can the threshold of similarity contribute to the fault-prone method prediction?)

Table V shows the mean performance values of the random forest models constructed in the steps 6a and 6b and the improvements of them by using the similarity-based classi-

TABLE IV
PART OF COMPUTATIONAL RESULTS: THE ODDS RATIO, FAULTY METHOD RATES, AND THE RATIO BETWEEN FAULTY METHOD RATES

threshold	odds ratio	faulty method rates (%)		ratio
τ	$OR(\tau)$	$FMR_L(\tau)$	$FMR_H(\tau)$	$\frac{FMR_H(\tau)}{FMR_L(\tau)}$
0.30	1.97	1.91	3.70	1.932
0.31	1.98	1.91	3.71	1.944
0.32	1.99	1.91	3.73	1.954
0.33	1.99	1.91	3.73	1.954
0.34	**2.02***	1.97	3.90	1.983
$\boxed{\mathbf{0.35}}$	**2.02***	1.97	3.91	**1.984***
0.36	**2.02***	1.97	3.92	1.983
0.37	2.01	1.99	3.92	1.971
0.38	1.99	2.01	3.93	1.953
0.39	2.00	2.01	3.95	1.960
0.40	**2.02***	2.04	4.02	1.975
0.41	**2.02***	2.04	4.02	1.977
0.42	2.00	2.05	4.02	1.960
0.43	1.97	2.09	4.03	1.928
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
0.60	1.71	2.43	4.09	1.682

TABLE V
COMPARISON OF MEAN PERFORMANCE VALUES

	Baseline random forest (RF)			RF + similarity classification			Improvement (%)		
Project	Recall	Precision	F value	Recall	Precision	F value	Recall	Precision	F value
Beam	0.4814	0.0528	0.0950	0.6208	0.0515	0.0951	+28.95%	−2.36%	+0.11%
Flink	0.3845	0.2104	0.2707	0.4569	0.1961	0.2732	+18.83%	−6.80%	+0.90%
Groovy	0.4876	0.2520	0.3314	0.5400	0.2660	0.3557	+10.73%	+5.54%	+7.31%
HBase	0.4187	0.2863	0.3394	0.4856	0.2988	0.3694	+15.97%	+4.37%	+8.80%
Hive	0.4052	0.1565	0.2255	0.4667	0.1522	0.2293	+15.17%	−2.72%	+1.66%
Kafka	0.7507	0.2415	0.3643	0.7777	0.2155	0.3363	+3.60%	−10.77%	−7.69%
RocketMQ	0.9833	0.1364	0.2390	0.9838	0.1327	0.2333	+0.05%	−2.67%	−2.36%
Storm	0.4910	0.0870	0.1477	0.6375	0.0809	0.1433	+29.83%	−7.13%	−2.96%
Zeppelin	0.4522	0.4491	0.4499	0.5183	0.5161	0.5165	+14.60%	+14.93%	+14.81%
Zookeeper	0.5717	0.4362	0.4937	0.6341	0.4548	0.5287	+10.91%	+4.27%	+7.10%
All	0.5426	0.2308	0.2957	0.6121	0.2365	0.3081	+12.80%	+2.45%	+4.20%

fication; In the table, “Baseline random forest (RF)” column and “RF + similarity classification” column correspond to the mean performance values of the random forests constructed in the step 6a (Fig. 2) and 6b (Fig. 3), respectively, and “Improvement” column presents the improvement rates of the latter value from the former value:

$$\frac{\text{latter value} - \text{former value}}{\text{former value}} \times 100\ (\%). \tag{5}$$

For example of the recall in Beam project, the former value is 0.4814 and the latter one is 0.6208, so the improvement rate is computed as: $(0.6208-0.4814)/0.4814\times 100 \simeq\ +28.95\ (\%)$.

In the table, the recall value, the precision value, and the F value improves by 12.8%, 2.45%, and 4.2% on average of all projects. Moreover, each of all projects shows an improvement in recall value as well. On the other hand, the precision values decrease in six out of ten projects. Because of the reduction of precision values in those projects, the F values also get lower in three out of ten projects.

As a result, the classification using the above threshold ($\tau^* = 0.35$) always works for improving the recall of the fault-prone method prediction. That is, the similarity-based classification contributes to the reduction of risk of overlooking fault-introducing events. Even though some projects showed decreases in the precision and the F values, these measures also got improved on average.

From the above results, our answer to RQ3 (How can the threshold of similarity contribute to the fault-prone method prediction?) are: *That threshold contributes to the reduction of risk of overlooking faults in the fault-prone method prediction. Moreover, although the precision value might be decrease in some cases, the performance of prediction tends to get improved—the recall:* $+12.8\%$*; the precision:* $+2.45\%$*; the F value:* $+4.2\%$.

D. Discussions

To answer RQ1, from each of Java methods in the studied projects, we collected the highest Levenshtein similarity (HLS) among the names of local variables in the method and compared the distribution of HLS values across the projects. As a result, the distributions seem to be close to each other. Moreover, there is no significant difference in the median among the projects. Because the highest similarity tends to be less than 0.5 in many Java methods, we can say that programmers would avoid having a pair of highly similar (confusing) variable names within the same method regardless of the project. To see this trend intuitively, we show samples of variable pairs appearing in the studied projects in Table VI. If the similarity between the two names is greater than 0.5, these names tend to get harder to discriminate each other because the half or more parts of them are duplicated. Especially, pairs of variable names whose similarity is around 0.8–0.9 may lead to an erroneous selection as they are a pair of a word and its plural form or a pair of almost the same names in which the only difference between them is the used number (for example, 1 or 2). It is better to avoid using such highly similar pairs for preventing human errors during the programming activity, and we should support it by providing a quantitative guideline and an automated checking tool.

TABLE VI
SAMPLE PAIRS OF VARIABLE NAMES

Similarity	Pairs of variable names
0.1	(from, components) (name, aggregator)
0.2	(annotation, node) (value, input)
0.3	(method, parameters) (filePath, fieldTypes)
0.35	(descriptorProperties, propertiesMap) (walDirPath, walDirForServerNames)
0.4	(that, other) (metaMethod, metaClass)
0.5	(residualElements, residualSource) (callable, callback)
0.6	(transform, translator) (data, datum)
0.7	(annotation, annotated) (newEntries, newEntry)
0.8	(file, files) (serverName, serverNode)
0.9	(sideInput, sideInputs) (offset1Adj, offset2Adj)

Next, to answer RQ2, we experimented with the similarity-based classification of Java methods and evaluated the effect of classification by the odds ratio in terms of the faulty method rate, while changing threshold τ. As a result, the odds ratio showed the highest value around 0.35–0.4, and we found that the appropriate threshold is 0.35. As we have seen in Table VI, pairs of variable names with the similarity are around 0.35–0.4 may have one or more common words in the compound names. These pairs would not be especially confusing. However, if they become a more similar pair, i.e., the similarity gets larger than 0.5, the risk of a human error such as a mix-up of the variable would also increase. Hence, we consider $\tau = 0.35$ would be a reasonable threshold to warn regarding the confusing pair of variable names.

Finally, to answer RQ3, we prepared two different random forest-based prediction models with and without using the similarity-based classification, and we evaluated how the prediction model with using the similarity (Fig. 3) outperforms the baseline model (Fig. 2) which did not use the similarity. As a result, the mean prediction performance values showed the following improvements: the recall improved by $+12.8\%$, the precision did by $+2.45\%$, and the F value did by $+4.2\%$. Notably, the recall got improved in all of the ten projects. Thus, we consider that the similarity-based classification can work for preventing the overlooking of faults more effectively. Although projects Kafka and RocketMQ showed relatively small improvements ($+3.6\%$ and $+0.05\%$) in Table V, the following two things would be significant reasons: their recall values were already at relatively high levels in the baseline model, and these projects have the least faulty methods in the data set (Table II). On the other hand, the precision values decreased from the baseline in some projects. In general, if we predict more objects as *positive* (i.e., *faulty*) to prevent overlooking the true-positive objects, the precision value tends to decrease. In our data set, because of a small number of faulty methods (Table II), we cannot avoid the depression of the precision when we increase the number of *positive* prediction cases. Nonetheless, the top four projects in terms of the faulty rate (Groovy, HBase, Zeppelin, and Zookeeper) showed improvements in both the recall and the prediction. Therefore, the similarity-based classification tends to help for improving the performance of the prediction model.

In the experiment for answering RQ3, we used only the optimal threshold (τ^*) obtained in the previous experiment regarding RQ2 because we aim to examine the effect of τ^* on the fault-prone method prediction model. Although other threshold values might also have similar effects, a detailed further investigation using various threshold values is our future work.

E. Threats to Validity

We discuss the threats to validity that can affect our results.

Conclusion Validity: The naming of variables may depend on the developers' experience and preference, so the heterogeneity of the project might have an impact on our empirical results. Because there was no significant difference in the median of similarity among the studied projects, we performed our study using a single threshold common to all projects. However, it may be better to collect more data from more projects and to analyze the effect of the threshold by the project domain. That is one of our significant future work.

Internal Validity: We quantitatively analyzed the relationship between the fault-introducing risk in a Java method and the similarity among the names of local variables in the method. However, the observed fault-introducing events may not always be related to the local variables of interest. In other words, there might be a fault at the part which is independent of the variable, and this is a threat to the internal validity. We need a finer-grained code analysis to link a fault-introducing event with the local variables in the future.

Construct Validity: Although we measured the similarity between variable names by using the Levenshtein distance, it is not the only way of evaluating the similarity. There are other edit distance metrics such as the longest common subsequence distance [27] and the Hamming distance [28], and the use of a different metric may lead to a different evaluation of similarity. Moreover, we can consider not only edit distance but also *semantic* distance by using the Doc2Vec [29] method. A further analysis using other similarity metrics is our significant future work.

To evaluate the effect of the similarity-based classification, we constructed the random forest together with the SMOTE algorithm. The selection of parameters in the model construction may affect the results: We specified the parameters of oversampling and undersampling in the SMOTE function so that the ratio between faulty methods and non-faulty methods becomes fifty-fifty, and used the default parameters in the randomForest function. Although a different parameter setting may make a different result, we did not do any special tuning to avoid yet another threat to validity, i.e., the issue of the appropriate setting selection.

External Validity: The threat to external validity is that our data set consists of ten Java open source software projects. Although we collected data from various projects, our results might not represent the results on all data of all software products. Moreover, the difference in the programming language may affect the trend of variable naming. A further data collection and analysis would be needed to mitigate this threat.

IV. Conclusion

In this paper, we focused on the similarity among the names of local variables in a Java method. Because the presence of a highly similar, i.e., "confusing" pair of local variables may decrease the code readability and cause a human error, we quantitatively analyzed such a risk by using the data of fault-introducing events. Through an empirical study using the data of $86,104$ Java methods collected from ten open source projects, we got the following findings:

- To reduce the risk of introducing faults into a Java method, programmers should avoid giving similar names to different variables. The threshold of Levenshtein similarity is 0.35.
- The classification of the Java method by the above threshold works for reducing the risk of overlooking fault-introducing events in the fault-prone prediction model; The recall of the prediction model improves by 12.8% on average.

When a programmer develops a Java method having two or more local variables, it is better to make the names of local variables dissimilar to each other. If there are highly similar names, they can be confusing and may decrease the readability, and consequently, may raise the risk of introducing faults. The above findings can form a quantitative guideline.

Our future work includes: 1) to further analyze the impacts of the differences in the project domain and the programming language by collecting more project data; 2) to develop the tool or the plugin for alerting the presence of similar variables based on our empirical results; 3) to examine the similarity of variable names by using other metrics including other edit distance metrics and semantic metrics.

Acknowledgment

This work was supported by JSPS KAKENHI #18K11246.

References

[1] C. Jones, *Applied Software Measurement: Global Analysis of Productivity and Quality*, 3rd ed. New York: McGraw-Hill, 2008.

[2] D. Boswell and T. Foucher, *The Art of Readable Code: Simple and Practical Techniques for Writing Better Code*. Sebastopol, CA: Oreilly & Associates, 2011.

[3] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Softw. Quality J.*, vol. 14, no. 3, pp. 261–282, Sept. 2006.

[4] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Softw. Eng.*, Nov. 2014, pp. 281–293.

[5] A. Corazza, S. D. Martino, and V. Maggio, "Linsen: An efficient approach to split identifiers and expand abbreviations," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, Sept. 2012, pp. 233–242.

[6] T. M. Pigoski, *Practical Software Maintenance: Best Practices for Managing Your Software Investment*, 1st ed. N.J.: Wiley, 1996.

[7] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Syst. & Softw. Eng.*, vol. 3, no. 4, pp. 303–318, Dec. 2007.

[8] G. Scanniello, M. Risi, P. Tramontana, and S. Romano, "Fixing faults in c and java source code: Abbreviated vs. full-word identifier names," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 2, pp. 6:1–6:43, Jul. 2017.

[9] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The impact of identifier style on effort and comprehension," *Empir. Softw. Eng.*, vol. 18, no. 2, pp. 219–276, Apr. 2013.

[10] K. Tashima, H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "Fault-prone java method analysis focusing on pair of local variables with confusing names," in *Proc. 44th Euromicro Conf. Softw. Eng. & Advanced App.*, Aug. 2018, pp. 154–158.

[11] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, "Identifier length and limited programmer memory," *Sc. Comp. Programming*, vol. 74, no. 7, pp. 430–445, May 2009.

[12] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the elipse ide?" *IEEE Softw.*, vol. 23, no. 4, pp. 76–83, July 2006.

[13] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge Univ. Press, 1997.

[14] H. Hata, O. Mizuno, and T. Kikuno, "Historage: Fine-grained version control system for java," in *Proc. 12th Int. Workshop Principles Softw. Evolution & 7th Annual ERCIM Workshop Softw. Evolution*, Sep. 2011, pp. 96–100.

[15] ——, "Bug prediction based on fine-grained module histories," in *Proc. 34th Int. Conf. Softw. Eng.*, Jun. 2012, pp. 200–210.

[16] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. 2005 Int. Workshop Mining Softw. Repositories*, May 2005, pp. 1–5.

[17] C. Williams and J. Spacco, "SZZ revisited: Verifying when changes induce fixes," in *Proc. Workshop on Defects in Large Softw. Systems*, Jul. 2008, pp. 32–36.

[18] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 641–657, July 2017.

[19] M. Borg, O. C. Svensson, K. Berg, and D. Hansson, "SZZ Unleashed: An open implementation of the SZZ algorithm – featuring example usage in a study of just-in-time bug prediction for the jenkins project," in *arXiv:1903.01742*, Mar. 2019, pp. 1–6.

[20] K. Fujiwara, H. Hata, E. Makihara, Y. Fujihara, N. Nakayama, H. Iida, and K. ichi Matsumoto, "Kataribe: A hosting service of historage repositories," in *Proc. 11th Working Conf. Mining Softw. Repositories*, May 2014, pp. 380–383.

[21] W. W. Daniel, *Applied Nonparametric Statistics*, 2nd ed. Boston, MA: Cengage Learning, 1990.

[22] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE Trans. Pattern Analysis & Machine Intelligence*, vol. 20, no. 8, pp. 832–844, Aug. 1998.

[23] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, July 2008.

[24] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 797–814, Aug. 2000.

[25] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[26] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artificial Intelligence Research*, vol. 16, pp. 321–357, Jun. 2002.

[27] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proc. 7th Int. Symp. String Processing & Inf. Retrieval*, Sep. 2000, pp. 39–48.

[28] D. J. MacKay, *Information Theory, Inference, and Learning Algorithms*. Cambridge: Cambridge University Press, 2003.

[29] J. H. Lau and T. Baldwin, "An empirical evaluation of doc2vec with practical insights into document embedding generation," in *Proc. 1st Workshop on Representation Learning for NLP*, Aug. 2016, pp. 78–86.

An Empirical Study on Progressive Sampling for Just-in-Time Software Defect Prediction

Xingguang Yang*†, Huiqun Yu*‡✉, Guisheng Fan*✉, Kang Yang*, Kai Shi§
*Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai 200237, China
†Shanghai Key Laboratory of Computer Software Evaluating and Testing, Shanghai 201112, China
‡Shanghai Engineering Research Center of Smart Energy, Shanghai, China
§Alibaba Group, Hangzhou, China

Abstract—**Just-in-time software defect prediction (JIT-SDP) is an active research topic in the field of software engineering, aiming at identifying defect-inducing code changes. Most existing JIT-SDP work focuses on improving the prediction performance of the model by improving the model. However, a frequently ignored problem is that collecting large and high quality defect data sets is costly. Specifically, when labelling the samples, experts in the field are required to carefully analyze the defect report information and log of code modification, which requires a lot of effort. Therefore, how to build a high-performance JIT-SDP model with a small number of training samples is an issue worth studying, which can reduce the size of the defect data sets and reduce the cost of data sets acquisition. This work thus provides a first investigation of the problem by introducing a progressive sampling method. Progressive sampling is a sampling strategy that determines the minimum number of training samples while guaranteeing the performance of the model. However, progressive sampling requires that the learning curve of the prediction model be well behaved. Thus, we validate the availability of progressive sampling in the JIT-SDP issue based on six open-source projects with 227417 changes. Experimental results demonstrate that the learning curve of the prediction model is well behaved. Therefore, the progressive sampling is feasible to tackle the JIT-SDP problem. Further, we investigate the optimal training sample size derived by progressive sampling for six projects. Empirical results demonstrate that a high-performance prediction model can be built using only a small number of training samples. Thus, we recommend adopting progressive sampling to determine the size of training samples for the JIT-SDP problem.**

Index Terms—**Just-in-time, software defect prediction, progressive sampling, mining software repositories**

I. Introduction

Defects in the software system can cause huge losses to companies [1]. Although software quality activities (such as source code checking and unit testing) can reduce the number of defects in software, they require a lot of testing resources. Therefore, how to release a high-quality software project with limited testing resources is a huge challenge in the field of software engineering [2]. Software defect prediction is an effective method. Developers use machine learning or statistical learning methods to identify the defect-proneness of program modules in advance, thereby investing more limited testing resources into defect-prone modules [2].

Corresponding Authors: Huiqun Yu (yhq@ecust.edu.cn), Guisheng Fan (gsfan@ecust.edu.cn)

Just-in-time software defect prediction(JIT-SDP) is a more fine-grained defect prediction method, which is made at change-level rather than module-level(e.g., function, file, and class) [1]. In the software development process, once the developer submits a modification to the software code, the defect prediction model will predict the defect-proneness of the code. If the change is predicted to be buggy, the corresponding developer will be assigned to check the change. Therefore, JIT-SDP has the advantages of fine granularity, instantaneity, and traceability [3], and has been adopted by many companies such as Lucent [4], BlackBerry [5], Cisco [6], etc.

Recently, JIT-SDP has received extensive attention and research. The main research work focuses on model building [7] [8], feature selection [1] [9], data annotation [10], etc. However, few studies have focused on the cost of acquiring defect data sets. Specifically, in order to obtain high-quality defect data sets, experts in specific fields are required to analyze version control systems (SVN, CVS, Git, etc.) and defect tracking systems (Bugzlla or Jira) during the data annotation phase [3]. Therefore, constructing an accurate defect data set is costly [11]. In the field of software engineering data mining [12] [13], researchers found the following relationship between the size of the data sets and the performance of the prediction model: When the data set size is small, the accuracy of the prediction model increases significantly as the size of the data increases. When the data sets size exceeds a certain number, adding more data does not lead to higher prediction performance. Therefore, how to build a high performance prediction model with fewer training samples for JIT-SDP is a problem worth studying, which brings two advantages:

- Firstly, reducing the size of training samples can reduce the cost of data sets labeling.
- Secondly, when using complex learning algorithms such as deep learning algorithms [8], reducing the size of the training data can significantly reduce the time required for model training.

In order to reduce the use of training samples, this paper first introduces progressive sampling into the study of JIT-SDP problems. We conduct experiment on the change-level defect data sets from six open source projects with 227417 changes. The main contributions of this paper are as follows:

- We introduce progressive sampling to the JIT-SDP study to determine the optimal training sample size and reduce the cost of defect data acquisition. However, progressive sampling requires that the learning curve of the prediction model be well behaved in coarse granularity. Therefore we conduct a large-scale empirical study based on the defect data sets from six open-source projects. The experimental results show that the learning curve of the prediction model is well behaved. So progressive sampling is efficient for JIT-SDP.
- We further investigate the optimal training sample size derived by progressive sampling based on six open-source projects. The experiment uses the random forest to establish a prediction model and uses AUC to evaluate the performance of the model. Empirical results show that using progressive sampling can significantly reduce the number of training samples used while guaranteeing the performance of the prediction model. Therefore, we recommend that in the practical application of JIT-SDP, using progressive sampling can effectively reduce the amount of training samples and reduce the cost of model building.

The rest of the paper is organized as follows: The related work is described in Section II. Section III introduces the progressive sampling and it's application in the scenario of JIT-SDP. Experimental setup is described in the Section IV. Section V introduces the experimental results and discussion. Section VI introduces the threats to validity. Conclusions and future work is described in the Section VII.

II. Related Work

A. Just-in-Time Software Defect Prediction

JIT-SDP is a special method for predicting software defects. Unlike traditional defect prediction, JIT-SDP is performed at change-level, which has finer granularity. Mockus and Weiss [4] first proposed the idea of JIT-SDP, and they designed a number of change metrics to predict whether changes are defect-inducing or clean. Recently, Kamei et al. [1] performed a large-scale empirical study in JIT-SDP. They collected eleven data sets from six open-source projects and five commercial projects. Their experimental results show that their prediction model can achieve 68% accuracy and 64% recall. Moreover, they find that 35% defect-inducing changes can be identified using only 20% of the effort.

Subsequently, researchers proposed various methods to improve the performance of the prediction model for JIT-SDP. Chen et al. [14] designed two objects through the benefit-cost analysis, and formalized the JIT-SDP problem into a multi-objective optimization problem. They proposed a method called MULTI based on NSGA-II [15]. The experimental results show that MULTI can significantly improve the effort-aware prediction performance for JIT-SDP. Furthermore, Yang et al. [16] found that the MULTI method is more biased towards the benefit object in the optimal solution selection. Therefore, they proposed a benefit-priority optimal solution selection strategy to improve the performance of the MULTI method. Cabral et al. [17] first found that JIT-SDP suffers from class imbalance evolution. Their proposed approach can obtain top-ranked g-means compared with state-of-the-art methods.

B. Progressive Sampling

Weiss and Tian pointed out that in the field of data mining, data acquisition is one of the main costs of the process of building a classification model [18]. Therefore, reducing the use of training data while guaranteeing the performance of the prediction model can reduce the cost of model building. In solving the actual classification task, using fewer training samples can still get a high prediction model. Thus, Provost et al. [19] proposed progressive sampling method. Progressive sampling continuously increases the number of training samples by the iterative method. Currently progressive sampling has been widely used in the field of software engineering data mining. For example, in the study of performance prediction for configurable software, obtaining data sets is costly. Thus, Sarkar et al. [12] used progressive sampling to determine the optimal number of training samples.

In the JIT-SDP study, obtaining high-quality defect data sets is costly, and it requires specialists in specific fields to thoroughly analyze defect report information and code modification logs [11]. The most existing JIT-SDP study only focuses on improving the performance of the prediction model, but ignores the cost of defect data sets acquisition. Therefore, this paper first introduces progressive sampling into the JIT-SDP scenario.

III. Progressive Sampling

A. Basic Concept of Progressive Sampling

Progressive sampling is a popular sampling strategy that has been used for various learning models [13]. Progressive sampling is an iterative process whose basic idea is to generate an array of integers $n_0, n_1, n_2, ..., n_k$. Each integer n_i indicates that the training samples with size of n_i are used to build the prediction model at the *i*th iteration. According to the calculation of the number of training samples in each iteration, progressive sampling can be classified as arithmetic progressive sampling and geometric progressive sampling [20]. The size of training samples for two progressive sampling is calculated as shown in Eq. (1) and Eq. (2), respectively, where n_0 represents the initial training sample size and a determines the growth rate of the training samples. It can be seen that the main difference between the two kinds of progressive sampling is that the geometric progressive sampling has a larger growth rate than the arithmetic progressive sampling, and is suitable for the prediction model with high algorithm complexity. Since the machine learning algorithm used in this paper is the random forest [21], the training time of the model is short, so it is suitable to use arithmetic progressive sampling.

$$n_i = n_0 + i * a \tag{1}$$

$$n_i = n_0 * a^i \tag{2}$$

Learning curve. The learning curve [19] describes the prediction performance of the prediction model at different training sample sizes and can clearly characterize the learning process of progressive sampling. Typical learning curve is shown as Fig.1, where the *x*-axis represents the training sample size and the *y*-axis represents the prediction performance of the model. A well behaved learning curve is monotonically non-decreasing and contains three regions: In the first region, the model performance increases rapidly as the training sample size increases; in the second region, the model performance increases slowly as the training sample size increases; in the third region, adding more training samples will not significantly improve the performance of the model.

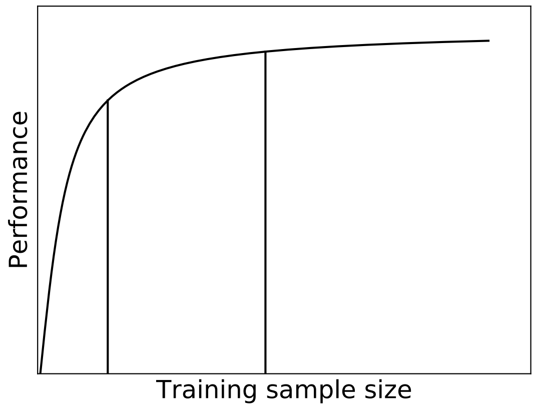


Fig. 1. Learning Curve

B. The Process of Progressive Sampling in the JIT-SDP

Progressive sampling is widely used in various software engineering related studies, such as performance prediction of configurable software [12], etc. In this paper, we first introduce progressive sampling into the JIT-SDP scenario. The detailed process is shown in Fig. 2, which involves four steps as following:

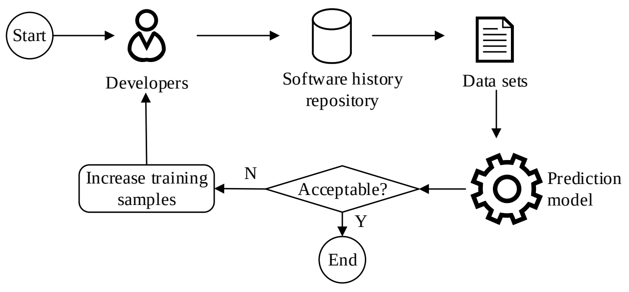


Fig. 2. The process of progressive sampling in the JIT-SDP

1) Developers mine metrics related to software changes from the software history repositories.
2) Domain experts label samples as defect-inducing or clean by analyzing defect reporting information and code modification logs in version control systems, and build defect data sets. Because of the high cost of this process, only a few number of changes are labeled.
3) Based on the existing data sets, a machine learning algorithm is used to build a defect prediction model.
4) After the prediction model is evaluated, it is required to determine whether the performance of the model is acceptable. If the performance is not acceptable, the more training samples will be collected according to the rules of progressive sampling.

We use following Algorithm 1 to describe more formally the application process of progressive sampling in JIT-SDP.

```
Algorithm 1: The progressive sampling for JIT-SDP
Input: initial sample size: $n_0$; growth factor: $a$;
       termination threshold: $threshold\_AUC$
Output: prediction model: $model$;
1  begin
2      # Build data sets with $n_0$ samples
3      $D = mining\_software\_repository(n_0)$
4      while true do
5          # Split data sets into training and test sets
6          $train\_set, test\_set = train\_test\_split(D)$
7          # Build prediction model based on machine
             learning methods
8          $model = Random\_Forest(train\_set)$
9          # Model evaluation
10         $AUC = Evaluation(model, test\_set)$
11         # Whether the model is acceptable
12         if $AUC > threshold\_AUC$ then
13             return $model$
14         end
15         else
16             $new\_D =$
                 $mining\_software\_repository(a)$
17             $D = D \cup new\_D$
18         end
19     end
20 end
```

We denote an instance of a code change as $X = \{x_1, x_2, ..., x_m\}$, where $x_1, x_2, ..., x_m$ represent the m metrics of the change X. An example of the change X is denoted as $(\mathbf{x}, Y)$, where $\mathbf{x}$ represents the values of metrics and Y represents whether the change is buggy or clean. If the change X is identified as buggy, then Y will be marked as 1, otherwise it is marked as 0. The defect data sets D for a specific project are composed of a set of examples X, where $X \subseteq D$.

In the beginning, developers need to mine n_0 samples from the software history repositories and build data sets D (Line 3). The data sets are then split into training and test sets (Line 6). The the defect prediction model is built and evaluated based on a machine learning algorithm (Line 8-10). Our experiment uses the random forest to build a prediction model and evaluate the model using AUC. If the performance of the model exceeds the threshold $threshold\_AUC$, the progressive sampling terminates the iteration (Line 12-13). Otherwise, it is necessary to further collect a samples from the software history repositories to increase the size of the data sets (Line 16-17).

IV. Experimental Setup

This paper introduces progressive sampling into the JIT-SDP problem and designs the following two research questions:

- RQ1: Whether the progressive sampling is feasible in the JIT-SDP scenario?
- RQ2: What is the optimal training sample size to establish a high performance prediction model by adopting progressive sampling?

The experimental hardware environment is *Intel(R) Core(TM)I7-7700 CPU RAM: 8G*. The programming environment used in the experiment is *python3.2*.

A. Data Sets

The data sets used in the experiment were provided by Kamei et al. [1] and are widely used in the field of JIT-SDP [7] [8] [9]. The data sets are collected from six open source projects, namely *Bugzilla(BUG)*, *Columba(COL)*, *Eclipse JDT(JDT)*, *Eclipse Platform(PLA)*, *Mozilla(MOZ)*, and *PostgreSQL(POS)*, with a total of 227417 changes. The number of defective changes, defect rate, and data collection period for each subject system are shown in Table I.

TABLE I
The basic information of data sets

Project	**Period**	**#defective changes**	**#changes**	**%defect rate**
BUG	1998/08/26∼2006/12/16	1696	4620	36.71%
COL	2002/11/25∼2006/07/27	1361	4455	30.55%
JDT	2001/05/02∼2007/12/31	5089	35386	14.38%
MOZ	2000/01/01∼2006/12/17	5149	98275	5.24%
PLA	2001/05/02∼2007/12/31	9452	64250	14.71%
POS	1996/07/09∼2010/05/15	5119	20431	25.06%

In order to accurately predict defects for software changes, Kamei et al. [1] designed 14 metrics. These metrics can be divided into five dimensions: diffusion, size, purpose, history, and experience. The specific description information is shown in Table II.

B. Prediction Model

Similar to previous research [22], the experiment uses the random forest algorithm to build prediction models [21], because previous studies have shown that random forest is highly robust, accurate and stable on JIT-SDP issues [23], and exceed other modeling techniques [24].

Random forest is an ensemble learning algorithm based on decision tree. Different from the conventional decision tree, the base learner randomly selects a subset of the attributes in each node's attribute set, and then selects an optimal attribute from the subset. Random forest algorithm is simple, easy to implement, and has low computational overhead, and is widely used in various learning tasks.

TABLE II
The description of metrics

Dimension	Metric	Description
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Distribution of modified code across each file
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	FIX	Whether or not the change is a defect fix
History	NDEV	Number of developers that changed the files
	AGE	Average time interval between the last and the current change
	NUC	Number of unique last changes to the files
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

C. Performance Indicators

The test samples can be divided into true positive(TP), false negative(FN), false positive(FP), and true negative(TN) according to the labels of the samples and the prediction results. The confusion matrix of the classification results is shown in the Table III. JIT-SDP is a binary classification problem. Common evaluation indicators include precision, recall, accuracy, etc. However, since the defect data sets are usually class-imbalanced, these threshold-based evaluation indicators are sensitive to threshold settings. Therefore, threshold independent evaluation indicators should be used [25].

The experiment uses AUC to evaluate the prediction performance of the model [1] [22]. AUC (Area Under Curve) is the area under the ROC curve. The ROC (Receiver Operating Characteristic) curve is drawn as follows: First, the test examples are sorted in descending order according to the probability that the prediction is positive; then the test examples are regarded as positive classes one by one, and the true positive rate (TPR) and false positive rate (FPR) are calculated each time; using TPR as the ordinate and FPR as the abscissa, one point of the ROC curve is obtained, and these points are connected to obtain the ROC curve.

$$TPR = \frac{TP}{TP + FN} \tag{3}$$

$$FPR = \frac{FP}{TN + FP} \tag{4}$$

TABLE III
Confusion matrix

Actual value	Prediction result	
	Positive	Negative
Positive	TP	FN
Negative	FP	TN

D. Data Preprocessing

In order to improve the prediction performance of the model, according to the recommendations of Kamei et al. [1], we conduct the following preprocessing on the data sets:

1) Remove highly correlated metrics. Since NF and ND, REXP and EXP are highly correlated, ND and REXP are excluded. Since LA and LD are highly correlated, LA and LD are normalized by dividing by LT. Since LT and NUC are highly correlated with NF, LT and NUC are normalized by dividing by NF.
2) Logarithmic transformation. Since most metrics are highly skewed, each metric(except for fix) performs a logarithmic transformation.
3) Dealing with class imbalance. The data sets used in the experiment are class-imbalanced, i.e., the number of defect-inducing changes is far more than the number of clean changes. Therefore, we perform random undersampling on the training set. By randomly removing the clean changes, the number of defect-inducing changes is the same as the number of clean changes.

V. Experimental Results and Discussion

This section answers the questions raised in Section IV through experiments.

A. Analysis for RQ1

Motivation. Progressive sampling is an effective means of determining the optimal training sample size, and is widely used in the field of software engineering [12]. This paper first introduces progressive sampling into the JIT-SDP problem to determine the optimal training sample size for the prediction model. However, in practical applications, progressive sampling requires that the learning curve of the prediction model be well behaved [19]. The basic characteristic of a well behaved learning curve is that the slope of the learning curve is monotonically non-increasing at the level of coarse granularity [19]. Therefore, we aim to verify whether progressive sampling is feasible on JIT-SDP issues through empirical research.

Approach. The experiment uses the six open source projects introduced in the Section IV as the research object to explore whether the learning curve of the JIT-SDP model is well behaved. Because the prediction model is based on a fast training random forest algorithm, the arithmetic progressive sampling method is adopted. To plot a learning curve for each data sets, we divided the data sets into two parts: 50% as the training pool for the constructing training sets and 50% as the test sets for the model evaluation. The two parameters of the arithmetic progressive sampling are as follows:

- $n_0 = |training\, pool| * 1\%$
- $a = |training\, pool| * 1\%$

Since progressive sampling requires the learning curve to be well behaved in coarse grain size, the granularity of our parameter settings is large. The initial number of training samples is 1% of the total number of training pools, and 1% of the number of training samples is added per iteration.

Findings. The experimental results are shown in Fig. 3, which contains six subgraphs, each of which represents a learning curve for a project, where the horizontal axis represents the size of training samples and the vertical axis represents the performance of the prediction model.

As can be seen from Fig. 3, the learning curve for each system is well behaved, which is generally monotonically non-decreasing. Although the learning curve fluctuates in local areas, the general trend is monotonically non-decreasing. Therefore, we can draw a conclusion that progressive sampling is feasible in the JIT-SDP problem.

B. Analysis for RQ2

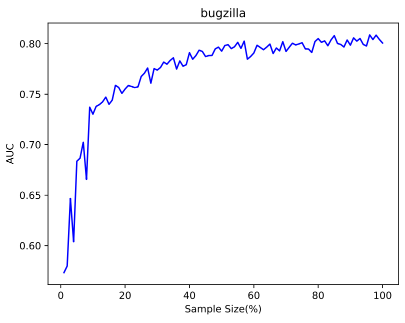
Motivation. The Section V-A has proven that progressive sampling is feasible in the JIT-SDP problem. However, what is the optimal training sample size to establish a high performance prediction model by adopting progressive sampling is a question worth studying. If a high-performance prediction model can be built with very few training samples, then only a small number of data sets need to be labelled during the construction of the defect data sets, which can greatly reduce the cost of data sets acquisition. Therefore, it is necessary to further investigate the optimal training sample size based on progressive sampling for JIT-SDP.

Approach. The experiment uses the data sets of the six projects introduced in Section IV. The prediction model is built based on the random forest, and the optimal training sample size is calculated based on the arithmetic progressive sampling. The experimental data sets are divided into two parts: 50% as a training pool for generating training sets and 50% as test sets for model evaluation. The parameters of the arithmetic progressive sampling are as follows: First, the initial sample size should be set small, so n_0 is set to 0.5% of the size of training pool. Second, since the training time of the model is short, the number of samples added at each iteration should not be too large. The experiment sets the growth rate a to 20. The threshold in the progressive sampling is used to determine whether the performance of the model is acceptable. Threshold settings are usually given by experts in a particular field. Previous studies have shown that the AUC value of the JIT-SDP model is usually not lower than 0.75 [22]. Therefore, the threshold of acceptable performance $threshold\_AUC$ is set to 0.75, i.e., once the AUC value of the prediction model is greater than or equal to 0.75, the progressive sampling terminates the iteration and returns the value of training sample size.

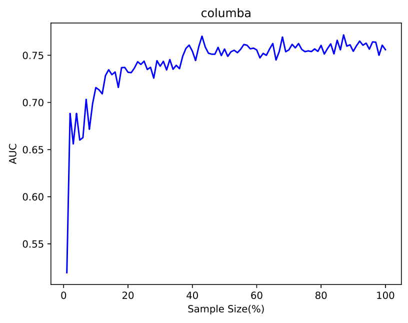
- $n_0 = |training\, pool| * 0.5\%$
- $a = 20$
- $threshold\_AUC = 0.75$

For better generalizability of experimental results, and to counter random observation bias, the entire experiment is repeated 100 times.

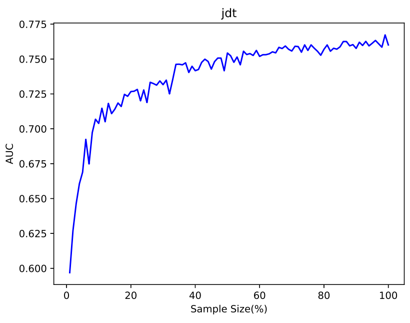
Findings. The experimental results are shown in Fig. 4, which describes box plots of optimal training sample sizes for six projects. Table IV shows the median of optimal training sample size for six projects, where the second column represents the median of optimal training sample sizes calculated from 100 experimental results, and the third column represents



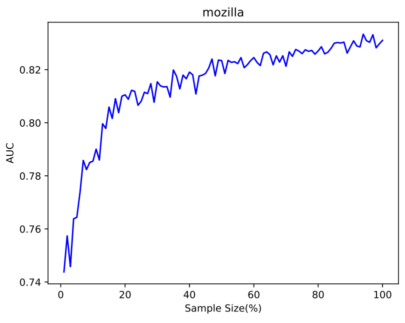
(a) BUG



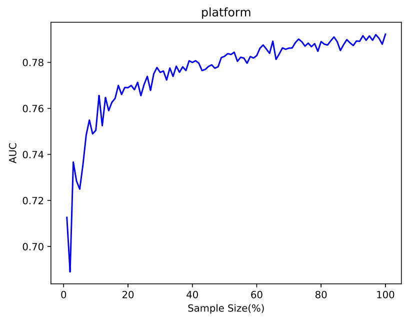
(b) COL



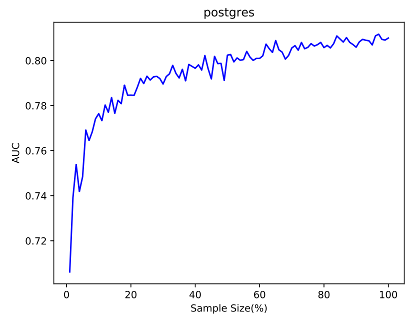
(c) JDT



(d) MOZ



(e) PLA



(f) POS

Fig. 3. The learning curves for six projects

the ratio of the optimal training sample size to the total number of data sets.

As can be seen from the Table IV, the optimal training sample size between each system has a large difference. In particular, for system MOZ, PLA, and POS, the proportion of the optimal training sample size to the total data sets is very low (less than 3%), i.e., a JIT-SDP model with high performance can be obtained by using less than 3% data sets as training sets.

Therefore, the use of progressive sampling is important for specific projects. Empirical studies have shown that using only a small number of samples can build a high performance prediction model. We recommend using progressive sampling to determine the number of training samples to reduce the cost of building defect data sets while preserving the performance of the model.

TABLE IV
THE MEDIAN OF OPTIMAL TRAINING SAMPLE SIZE FOR SIX PROJECTS

Project	Optimal training sample size#	Rate%
BUG	361	7.81
COL	931	20.89
JDT	6778	19.15
MOZ	565	0.57
PLA	1890	2.94
POS	271	1.32

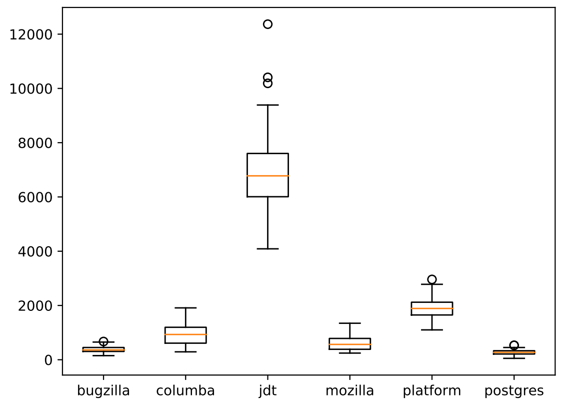


Fig. 4. Optimal training sample size

VI. Threats to Validity

External validity. Threats to external validity are mainly from the data sets used in the experiment. Although data sets are widely used in JIT-SDP research [1] [22] [7], whether the experimental conclusions can be extended to other project data sets remains to be further verified. Therefore, more data sets have yet to be mined to verify the generalization of experimental results.

Construct validity. The threats to construct validity are mainly considered whether the evaluation indicator used in our experiment can accurately reflect the prediction performance of the prediction model. The experiment uses AUC to evaluate the JIT-SDP model, which is also widely adopted by previous research [22].

Internal validity. The threats to internal validity are mainly from experimental code. Our experimental code is written in python. In order to reduce errors in the code, we used mature libraries and carefully checked the code of the experiment.

VII. Conclusions and Future Work

In this paper, we first introduce progressive sampling for the JIT-SDP. Progressive sampling is a commonly used sampling strategy that progressively increases the number of training samples to determine the optimal number of training samples.The experiment is conducted on six open source projects with 227417 changes. Our prediction model is built based on the random forest algorithm and evaluated by AUC.

Large-scale empirical studies demonstrate that progressive sampling is feasible in the JIT-SDP scenario. Moreover, experimental results show that the optimal training sample size derived by progressive sampling is very small. Especially, the proportion of training samples to the total number of data sets is less than 3% on the projects MOZ, PLA, and POS. Therefore, we suggest that progressive sampling can be used in the practical application of JIT-SDP to determine the optimal number of samples, thereby reducing the number of training samples and reducing the cost of acquiring data sets.

In the future, we plan to design a more intelligent progressive sampling method. We aim to further reduce the training sample size by selecting samples more intelligently so that progressive sampling can reach the termination condition earlier. Secondly, in order to further verify the generalization of the experimental conclusions, we hope to collect more data sets to improve the reliability of the experimental conclusions.

Acknowledgment

This work is partially supported by the NSF of China under grants No.61772200 and 61702334, Shanghai Pujiang Talent Program under grants No. 17PJ1401900. Shanghai Municipal Natural Science Foundation under Grants No. 17ZR1406900 and 17ZR1429700. Educational Research Fund of ECUST under Grant No. ZH1726108. The Collaborative Innovation Foundation of Shanghai Institute of Technology under Grants No. XTCX2016-20.

References

[1] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[2] Z. Li, X. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *IET Software*, vol. 12, no. 3, pp. 161–175, 2018.

[3] L. Cai, Y. Fan, M. Yan, and X. Xia, "Just-in-time software defect prediction:a road map," *Journal of Software*, vol. 30, no. 5, pp. 1288–1307, 2019.

[4] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[5] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE*, p. 62, 2012.

[6] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *37th IEEE/ACM International Conference on Software Engineering, ICSE*, pp. 99–108, 2015.

[7] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2823–2862, 2019.

[8] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: an end-to-end deep learning framework for just-in-time defect prediction," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR*, pp. 34–45, 2019.

[9] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code churn: A neglected metric in effort-aware just-in-time defect prediction," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM*, pp. 11–19, 2017.

[10] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.

[11] X. Chen, L. Wang, Q. Gu, Z. Wang, C. Ni, W. Liu, and Q. Wang, "A survey on cross-project software defect prediction methods," *Journal of Computer*, vol. 41, no. 1, pp. 254–274, 2018.

[12] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pp. 342–352, 2015.

[13] A. Lazarevic and Z. Obradovic, "Data reduction using multiple models integration," in *Principles of Data Mining and Knowledge Discovery, 5th European Conference, PKDD*, pp. 301–313, 2001.

[14] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan, "MULTI: multi-objective effort-aware just-in-time software defect prediction," *Information & Software Technology*, vol. 93, pp. 1–13, 2018.

[15] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[16] X. Yang, H. Yu, G. Fan, and K. Yang, "An empirical studies on optimal solutions selection strategies for effort-aware just-in-time software defect prediction," in *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE*, pp. 319–424, 2019.

[17] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *Proceedings of the 41st International Conference on Software Engineering, ICSE*, pp. 666–676, 2019.

[18] G. M. Weiss and Y. Tian, "Maximizing classifier utility when there are data acquisition and modeling costs," *Data Mining and Knowledge Discovery*, vol. 17, no. 2, pp. 253–282, 2008.

[19] F. J. Provost, D. D. Jensen, and T. Oates, "Efficient progressive sampling," in *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD*, pp. 23–32, 1999.

[20] G. H. John and P. Langley, "Static versus dynamic sampling for data mining," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, KDD*, pp. 367–370, 1996.

[21] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[22] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2072–2106, 2016.

[23] Y. Jiang, B. Cukic, and T. Menzies, "Can data transformation help in the detection of fault-prone modules?," in *Proceedings of the 2008 Workshop on Defects in Large Software Systems, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, pp. 16–20, 2008.

[24] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *26th IEEE International Conference on Software Maintenance, ICSM*, pp. 1–10, 2010.

[25] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2017.

A Code Clone Curation
— Towards Scalable and Incremental Clone Detection —

Masayuki Doi, Yoshiki Higo, and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
{m-doi, higo, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Code clones have a negative impact on software maintenance. Code clone detection on large-scale source code takes a long time, and even worse, such detections occasionally aborted due to too much target size. Herein, we consider that we detect cross-project code clones from a set of many projects. In such a detection situation, even if a project is updated, we need to detect cross-project code clones again from the whole of the projects if we simply use a clone detection tool. Therefore we need a new clone detection scheme that rapidly detects code clones from a set of many projects with incremental detection functionality. In this paper, we propose an approach that rapidly obtains code clones from many projects. Our approach includes a strategy of multi-stage code clone detection unlike other clone detection techniques: in the first-stage detection, code clones are detected from each of the projects, respectively; then in the second-stage detection, the code clones detected in the first-stage are unified by using our clone curation technique. This multi-stage detection strategy has a capability of incremental clone detection: if the source code of a project is updated, the first-stage detection is applied only to the project and then the second-stage curation is performed. We constructed a software system based on the proposed approach. At present, the system utilizes an existing detection tool, CCFinder. We have applied the system to the large-scale source code (12 million LoC) of 128 projects. We also detected clones from the target source code by simply using CCFinder and compared the detection time. As a result, the clone detection with our system was 17 times shorter than CCFinder's detection. We also compared detection time under an assumption that each project of the targets is updated once. The experimental results showed that our incremental clone detection shortened the detection time by 91% compared to non-incremental clone detection.**

***Index Terms*—code clone detection, large-scale, incrementability**

I. Introduction

A code clone (hereafter, clone) is a code fragment which is identical or similar to another code fragment in source code. Clones may lead to software maintenance problems [1], [2] and bug propagation [3], [4]. Thus, researchers have been actively studying techniques for clone detection. For example, a number of tools for automatic clone detection from source code have been developed and released [5].

The amount of source code is steadily increasing, and large-scale clone detection has become a necessity. Large-scale clone detection can be used for detecting similar mobile applications [6], finding the provenance of components [7], and code search [8]. Thus, scalable clone detectors have been developed [9]. However, such clone detectors seek to detect all present clones in the target software, so when large-scale clone detection is conducted with such a clone detector, the clone detector outputs a huge number of clones as the results of the clone detection. It is impractical to check manually whether or not each of all detected clones is worth more than a glance. Furthermore, if a new project is added to the target projects of clone detection or a part of the target projects is updated, the clone detector needs to detect clones from all the target projects again. Such a clone detection takes a too long time. In order to solve those problems, a new technique for large-scale clone detection is required.

The results of clone detections may include many negligible clones. Negligible clones are worthless when dealing with clone information in software development and maintenance. For example, language-dependent clones are negligible clones [10]. When a specific programming language is used, a programmer cannot help writing some similar code fragments which cannot be merged into code fragments due to language limitations. A language-dependent clone consists of such code fragments, so that language-dependent clones are negligible. Many techniques have been proposed to remove the negligible clones from the clone detection results. The techniques classify clones according to whether code fragments are high cohesion and low coupling [11], according to the ratio of repetitive structures included in code fragments [10], and according to the machine learning using some metrics [12]. By using those techniques, negligible clones can be filtered out from the clone detection results. We thought that incorporating such filtering technique in clone detection process should make it possible to detect clones on a larger scale more efficiently.

In this paper, we propose a clone curation approach which rapidly detects clones from many projects. Our approach detects clones in three stages: (1) detecting intra-project clones, (2) filtering out negligible clones from the detection results, and (3) consolidating different sets of similar intra-project clones into a set of cross-project clones. The above process realizes a scalable clone detection because

- traditional clone detection, which requires a high computational complexity, is performed on small-size source code (source code of a single project)
- negligible clones are filtered out before generating cross-project clones.

We implemented the proposed technique and evaluated the scalability of the proposed technique. As a result, when the proposed technique and CCFinder [13] detected clones from 128 pseudo projects, each of which includes 100 KLoC, the proposed technique was able to detect clones up to 17 times faster than CCFinder. In addition, in order to evaluate whether the proposed technique can detect clones rapidly in the case of updating a part of the target projects, we measured the execution time required to detect clones again.As a result, our technique was able to detect clones 11 times faster than CCFinder in such a situation. Finally, we demonstrated that adjusting the parameters of our technique can detect clone faster.

The remainder of this paper is structured as follows. Section II describes the definitions of the terminology used in this paper. Section III presents our approach. Section IV describes the implementation of our approach. Section V describes the design of the evaluation experiments. Section VI describes the datasets used in the evaluation experiments. Section VII describes the results of the experiments and the discussions. Section VIII describes the threats of validity. Section IX describes the conclusion and future work.

II. PRELIMINARIES

A. Code clone

Given two identical or similar code fragments, a **clone relation** holds between the code fragments. A clone relation is defined as an equivalence relation (i.e., reflexive, transitive, and symmetric relation) on code fragments.

If a clone relation is established in given two code fragments, the code fragment pair is called a **clone pair**. An equivalence class of a clone relation is called a **clone set**. That is, a clone set is a maximal set of code fragments where a clone relation exists between any pair of them. A code fragment within a clone set is called a **code clone** or just a **clone**.

B. RNR

Many negligible clones are included in the detection results. In our previous research, we proposed a metric RNR to automatically filtering out such negligible clones [10]. $RNR(S)$ means the ratio of non-repeated code sequence in a clone set S. Here, we assume that a clone set S includes a code fragment f. $LOS_{whole}(f)$ represents the length of the whole sequence of fragment f, and $LOS_{repeated}(f)$ represents the length of the repeated sequence of f, then metric $RNR(S)$ is calculated by the following formula:

$$RNR(S) = 1 - \frac{\sum_{f \in S} LOS_{repeated}(f)}{\sum_{f \in S} LOS_{whole}(f)}$$

For example, we assume that we detect clones from following two source files (F_1, F_2). Each source file consists of the following five tokens.

$F_1 : a\ b\ c\ a\ b$

$F_2 : c\ c^*\ c^*\ a\ b$

where the superscript "$*$" indicates that its token is in a repeated code sequence.

We use label $C(F_i, j, k)$ to represent the fragment. The fragment $C(F_i, j, k)$ starts at the j-th token and ends at the

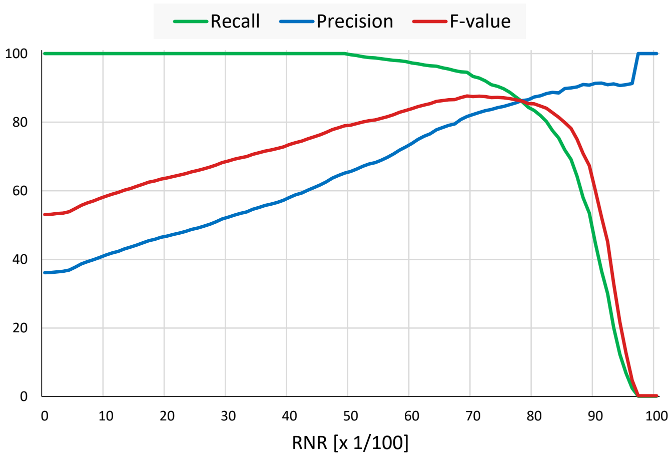


Fig. 1. Transitions of Recall, Precision, and F-value by RNR

k-th token in source file F_i(j must be less than k). In this case, the following two clone sets are detected from the source files.

$S_1 : C(F_1, 1, 2), C(F_1, 4, 5), C(F_2, 4, 5)$

$S_2 : C(F_2, 1, 2), C(F_2, 2, 3)$

A RNR value of each clone set is the following:

$$\begin{aligned} RNR(S_1) &= 1 - \frac{0+0+0}{2+2+2} = 1.0 \\ RNR(S_2) &= 1 - \frac{1+2}{2+2} = 0.25 \end{aligned}$$

The value 0.25 of $RNR(S_2)$ represents that most of the tokens in S_2 are in the repeated code sequence. Our previous research reported that low-RNR clone sets are typically negligible clones such as consecutive variable declarations, consecutive method invocations, and case entries of switch statements.

In our previous research, we examined how well the RNR filtering worked. We calculated precision, recall, and f-value of the filtering. The definitions of the values are the followings:

$$\begin{aligned} Recall(\%) &= 100 \times \frac{|S_{negligible} \cap S_{filtered}|}{|S_{filtered}|} \\ Precision(\%) &= 100 \times \frac{|S_{filtered}|}{|S_{negligible}|} \\ F - value &= \frac{2 \times Recall \times Precision}{Recall + Precision} \end{aligned}$$

where $S_{filtered}$ is clone sets filtered out by RNR and $S_{negligible}$ is all real negligible clone sets.

Figure 1 illustrates transitions of recall, precision, and f-value when the RNR threshold varies between 0 and 1. Figure 1 means 65% of negligible clone sets are filtered out with no false positive by using 0.5 as the RNR threshold.

III. PROPOSED TECHNIQUE

The entire process of our approach is summarized in Figure 2. It is composed of three steps: clone detection on each project, negligible clones exclusion, and clone curation cross projects. The following subsections describe the design of each step.

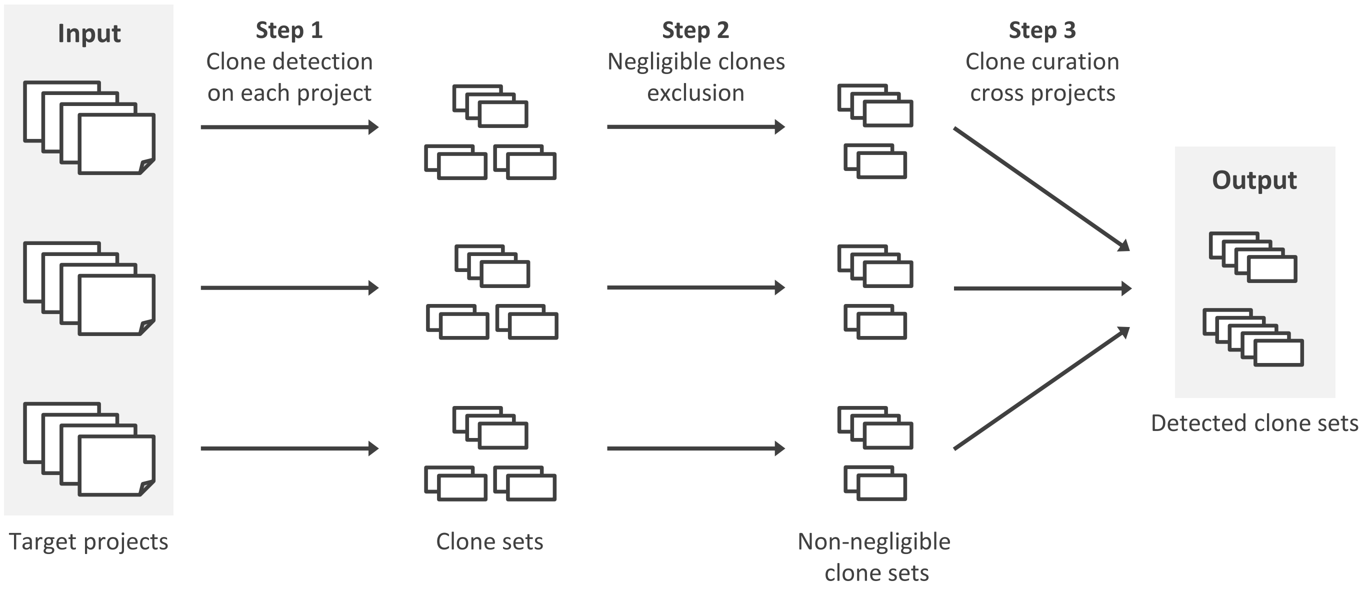

Input
Step 1
Clone detection on each project
Step 2
Negligible clones exclusion
Step 3
Clone curation cross projects
Output
Detected clone sets
Target projects
Clone sets
Non-negligible clone sets

Fig. 2. The Overview of Our Approach

A. Clone detection on each project

In this step, metric RNR is calculated from each of the detected clone sets after they are detected from each of the target projects. This process realizes a scalable clone detection because

- the size of the source code to be detected is reduced, and
- clone detection can be capable of parallel execution in each of the target projects.

In the case of detecting clones from a large project, our approach can detect clones faster by dividing the project into a set of smaller pseudo projects. However, over-division may lead to finding fewer clones to be detected because our approach cannot detect clone pairs that exist only between the projects.

We utilize an existing detection tool. Many detectors output a text file as the detection results. Consequently, our approach can be applied to many detectors such as token-based or PDG-based ones because the clone sets, which are the output of this step, can be obtained by parsing the text file outputted by detecting clones from each of the target projects. Moreover, if a clone detector has a function to calculate RNR as well as detecting clone sets, this step can be performed efficiently. For example, CCFinder [13] has a function to calculate RNR. In this case that a clone detector does not have the function, we need to calculate RNR as a post-process of clone detection.

B. Negligible clones exclusion

It is ineluctable that negligible clones are included in clone detection results. The presence of negligible clones has negative impacts from the following two viewpoints: making it more difficult to analyze clone detection results, and taking a longer time to finish clone detections. Hence, our approach excludes the negligible clones from candidates of cross-project clones. Clone sets to be excluded are satisfying $RNR < 50$ based on the previous research [10]. There are two reasons to exclude low-RNR clones: (1) users can avoid spending their time to analyze negligible clones; (2) the execution time of clone detection gets shortened. Hence the number of clone sets that are the targets of similarity calculation gets reduced, and then similarity calculation can be performed more efficiently.

C. Clone curation cross projects

The curation of cross-project clone sets is performed based on the judgment whether the similarity between clone sets is equal to or greater than the threshold.

Consequently, our technique omits similarity calculation between a pair of clone sets if the pair satisfies both the following conditions.

- The RNR value of a clone set in the pair is largely different from the one of the other clone set of the pair.
- The RNR value of either clone set in the pair is sufficiently large.

RNR is based on the code fragment structure. Similar clone sets may be the almost same RNR values. Thereby the clone detection should get faster by omitting similarity calculation between clone sets which have the big difference between each RNR values and are the one of the clone set pair's RNR is sufficiently large. More completely, RNR_m, RNR_n values of given to clone sets and two parameters θ_{diff} and θ_{max}, our approach determinates whether the similarity between clone sets are satisfying the following formula.

$$\begin{aligned} Omit(\theta_{diff}, \theta_{max}) \quad = \quad & (|RNR_m - RNR_n| > \theta_{diff}) \\ & \wedge (max(RNR_m, RNR_n) > 100 - \theta_{max}) \end{aligned}$$

where max is a function to return the maximum value in the parameters. Figure 3 shows the area of omitting similarity

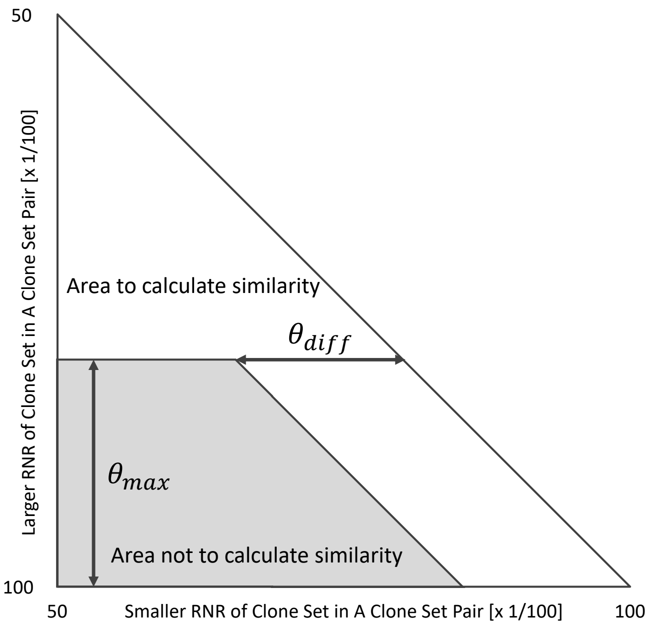

50
100
50
100
Larger RNR of Clone Set in A Clone Set Pair [x 1/100]
Smaller RNR of Clone Set in A Clone Set Pair [x 1/100]
Area to calculate similarity
θ_{diff}
θ_{max}
Area not to calculate similarity

Fig. 3. Area to Calculate Similarity Between A Clone Set Pair

calculation by the formula. The first expression is represented that RNR value of a clone set in the pair is largely different from the one of the other clone set of the pair. The θ_{diff} parameter means the threshold of the difference. The second expression is represented that the RNR value of either clone set in the pair is sufficiently large. The θ_{max} parameter means the threshold of the RNR value. For the relationship between the appropriate values of the parameters required for the similarity calculation and the detection accuracy and the speed based on them, the evaluation experiment is described in Section V-C.

IV. IMPLEMENTATION

A. Clone detection tool

Our proposed approach detects clones using a clone detection tool. Our implementation utilizes CCFinder [13] .

B. Coefficient of similarity

We define the similarity between a pair of clone sets. Hereafter we call a pair of clone sets *clone set pair*.

$$Sim(S_m, S_n) = \frac{|N(S_m) \cap N(S_n)|}{max(|N(S_m)|, |N(S_n)|)}$$

where S_m and S_n are a clone set, respectively. N is a function to return a set of n-grams created from the tokens included in a given clone set. The target tokens are extracted by a lexical analysis on the text representation of the clone set and a normalization of the identifiers and literals included in the text. N-gram size of our implementation is 5, which is on Myles et al. [14]. max is a function to return the maximum value in the parameters. The reason why we use n-gram is that we want to calculate text similarity without considering whether each instruction is repeated or not. If two clone sets have a higher similarity than a threshold θ, they are regarded as a single clone set. Otherwise, they are repeated as different clone sets as they are. At this moment, we use 0.9 as a default value. The value of 0.9 is the evaluated in Sec.VII.

V. Experimental Design

This research evaluates our proposed approach by answering the following research questions.

A. ***RQ1: How fast does our approach detect cross-project clones?***

One primary goal is to detect cross-project clones as fast as possible. In this evaluation, we create same-size pseudo projects from BigCloneBench dateset [15]. Then, we compare the execution time of two clone detections: our proposed approach and CCFinder. In the formal detection, in-project clones are detected by CCFinder from each pseudo project. And then, cross-project clones are generated by the curation. In the latter detection, cross-project clones are directly detected from the whole of the pseudo projects by using CCFinder.

B. ***RQ2: How fast does our approach follow update of target projects?***

Our proposed approach utilizes the results of the clone detection per project. Herein, we assume that the source code of a project updated and we need to obtain cross-project clones again. If we use the proposed approach, we only need to detect clones again from the updated project. And then, the detection results of the updated project and the detection results of the other projects are input to the curation. If we do not use the proposed approach, we need to run CCFinder for the whole of the target projects. In this evaluation, we compare the two kinds of detections: our proposed approach and CCFinder.

C. ***RQ3: How effective is parameters adjustment to improve clone detection performance?***

As described in Section III-C, our proposed approach has two parameters θ_{diff} and θ_{max} that specify the area not to calculate similarity. We seek for appropriate parameters by executing the proposed approach with different parameters.

VI. Data Sets

In this research, we utilized two datasets, Apache [16] and BigCloneBench [15] datasets.

The Apache dataset is composed of 84 Java projects in Apache Software Foundation[1]. Unfortunately, CCFinder reported encoding errors on nine projects. This is because the nine projects included at least a file that is encoded with a character code that CCFinder cannot treat. Consequently, we excluded the nine projects from our target ones.

The BigCloneBench dataset is one of the largest clone benchmarks available to date. It was created from IJaDataset 2.0 [2], which is composed of 25,000 Java systems. The benchmark includes 2.9 million source files with 8 million manually-validated clone pairs. The BigCloneBench dataset was used for

[1]http://www.apache.org

[2]https://sites.google.com/site/asegsecold/projects/seclone

clone evaluations and scalability testing in several large-scale clone detection and clone search studies [8]. In this research, we constructed 256 pseudo projects from the BigCloneBench dataset. The 256 pseudo projects are composed of two groups, 128 projects of 10 KLoC and 128 projects of 100 KLoC. Each pseudo project consists of source files that were randomly selected from the BigCloneBench dataset.

VII. Evaluation

We conducted an evaluation of our proposed approach on a single workstation, which has the following specification.

- Windows 10
- 1.6GHz processor (6 cores)
- 32 GB Memory
- 512 GB SSD

The proposed approach curates clone sets if the similarity between the different clone sets exceeds a threshold. Hence, the choice of the similarity threshold affects the curation performance . To find the optimal threshold for curation, we measured the execution time and the number of the curated clone sets when we curate clone sets from the Apache dataset described in Sec. VI with different thresholds. Table I shows the results.

As the threshold value was reduced, The execution time increased significantly. On the other hand, as the threshold value was increased, the execution time was shortened, but the number of clone set pairs whose similarity exceeds the threshold value decreases. Hence, the optimal value is 0.9. Therefore, we utilized 0.9 as the threshold value for subsequent evaluations.

A. RQ1: How fast does our approach detect cross-project clones?

To answer RQ1, we compared the execution time of two clone detections: our proposed approach and CCFinder. We used the following thresholds in the application of our proposed technique.

- θ_{diff}: 0
- θ_{max}: 0 (Similarity calculation is not omitted.)

We utilized the dataset, which had constructed from BigCloneBench dataset described in Section VI. Figure 4 shows the overview of how to construct the dataset. In order to detect clones with changing the number of the target projects, we extracted seven sets of the pseudo projects randomly from each of the 10 KLoC group and the 100 KLoC group, respectively. The number of pseudo projects in a set i, $i \in [1, 7]$, is 2^i.

The execution time of each clone detection is shown in Figure 5. Our approach can detect clones rapidly in the both

TABLE I
Execution Time and Curated Clone Sets with Different Parameterization

Threshold	0.85	0.9	0.95
Execution Time	42m 28s	26m 55s	21m 40s
#Curated Clone Set Pairs	80,104	52,785	29,762

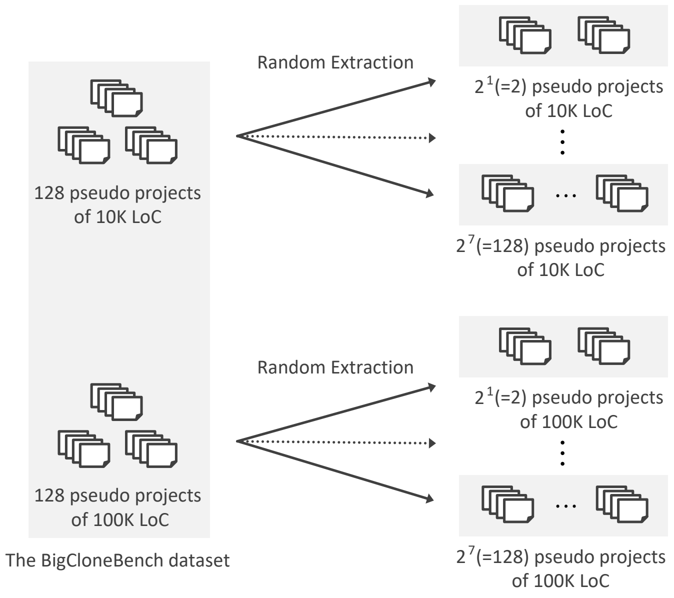
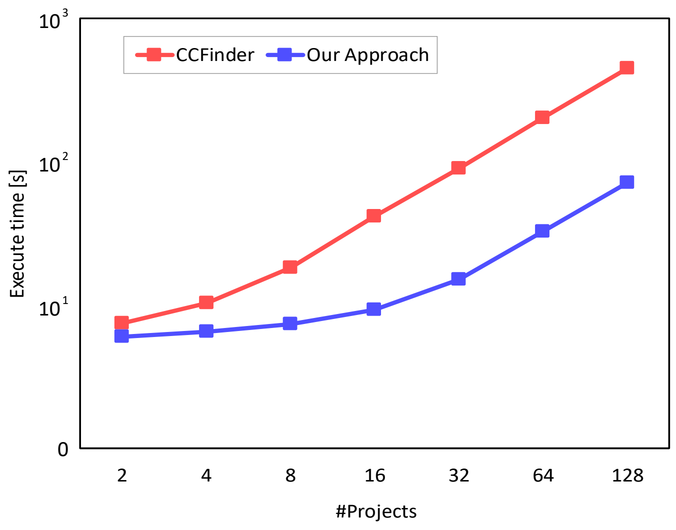

Random Extraction
128 pseudo projects of 10K LoC
2^1(=2) pseudo projects of 10K LoC
2^7(=128) pseudo projects of 10K LoC
Random Extraction
128 pseudo projects of 100K LoC
2^1(=2) pseudo projects of 100K LoC
2^7(=128) pseudo projects of 100K LoC
The BigCloneBench dataset

Fig. 4. The Overview of How to Construct The Dataset

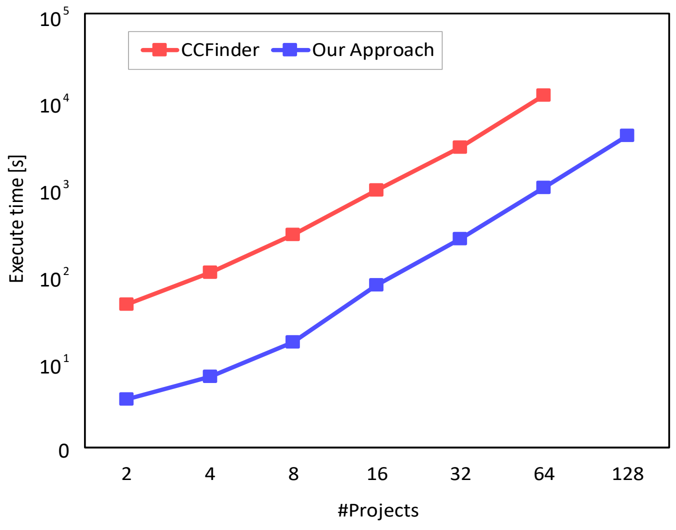
cases of 10 KLoC and 100 KLoC. The 1.24 times was the minimum value while the maximum value was 6.24 times in 10 KLoC. The greater the number of target projects is, the higher the ratio of the execution time between our proposed approach and CCFinder one is. We consider there are two reasons for that we obtained such results. Our proposed approach detects clones in parallel and it detects within-project clones from each project instead of cross-project clones from the whole dataset.

The 11.4 times was the minimum value while the maximum value was 17.1 times in 100 KLoC. Our proposed approach was able to detect clones from the 128-project dataset of 100 KLoC. On the other hand, CCFinder was not able to finish detecting clones from the same dataset due to out of memory error. Detecting clones from vast source code requires huge memory, which is the reason why CCFinder failed to detect clones from the dataset. In our approach, CCFinder takes only a small amount of source code in a single clone detection and CCFinder is executed many times in parallel. Consequently, in our approach, out of memory error did not occur. Our approach was able to detect clones from the large dataset where CCFinder was not able to finish detecting clones due to insufficient memory.

We also evaluated the ratio of clones that were not detected by our proposed approach. The approach cannot detect clone pairs that exist only between different projects because the approach curates clones that are detected from each of the projects. Hence, we measured the ratio of clones that the proposed approach could detect. We utilized the 64 pseudo projects of the 100 KLoC as the target projects, and we compared the number of clones that could be detected by the proposed approach and the number of clones that could be detected by CCFinder. The results are shown in Fig.6. It can be seen that the ratio of clones that can be detected decreases



(a) Execution Time for Projects of 10 KLoC



(b) Execution Time for Projects of 100 KLoC

Fig. 5. Execution Time for Each Project Group

as the number of projects increases. That is because clones that exist only between the different projects increase as the number of projects increases. As a result, in the case of 64 projects of 100K LoC, the 38 percent of clones were not be detected, but the remaining 62 percent of clones can be detected 17 times faster than CCFinder.

Our answer to RQ1 is that our proposed approach can detect clones up to 17 times faster than CCFinder. Our approach can detect clones from the large target that cannot detect clones by CCFinder due to insufficient memory.

B. RQ2: How fast does our approach follow update of target projects?

In this evaluation, we assumed a situation: the source code of each target project is updated once in different dates; a user wants to detect cross-project clones as rapid as possible just after any project is updated. Under this assumption, we utilized the 64 pseudo projects of the 100 KLoC, which had been

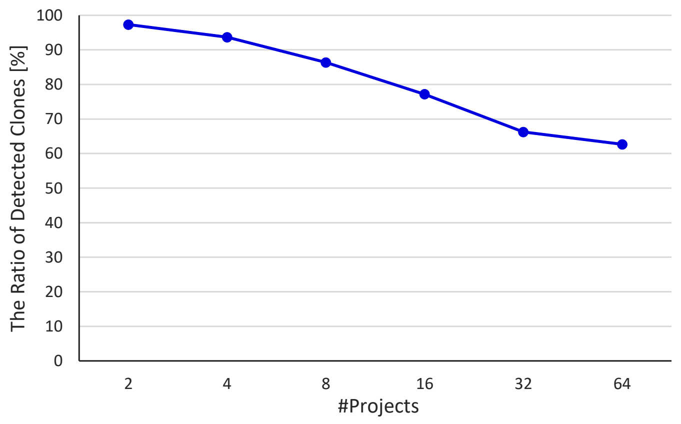


Fig. 6. The Ratio of Detected Clones

constructed in RQ1. We did not use the dataset of 128 projects because CCFinder cannot detect clones from the dataset. Thus, in the case that we directly detect cross-project clones with CCFinder, a clone detection from the whole of the 64 projects is executed 64 times. On the other hand, in the case that we apply the proposed approach, a clone detection from the updated project and a clone curation for the whole of 64 projects are executed 64 times. We measured the execution time of the both cases.

The measurement results are shown in Table II. CCFinder took 11,207 seconds while our approach took only 1,006 seconds to detect clones on average. Therefore, our approach was able to detect clones 11 times faster than CCFinder. We investigated the ratio of clone detection and clone curation in our approach and found that 97% of the execution time spent on clone curation.

Our answer to RQ2 is that the proposed approach took only 9% execution time of detecting cross-project clones from the whole of 64 pseudo projects with CCFinder.

C. RQ3: How effective is parameters adjustment to improve clone detection performance?

In the process of answering RQ1 and RQ2, we found that the clone curation step accounts for the majority of the execution time. This is because our approach calculates the similarities between all clone sets. To curate clones more rapidly, our approach needs to reduce the number of similarity calculations. We considered that omitting similarity calculation between some clone sets, which have little effect on the curation results, can curate clones more rapidly. We also considered that we could identify such clone set pairs by utilizing RNR, which is described in Section III-C. Thus, to seek for such clone set pairs, we investigated the distribution of similar clone set pairs in the 75 projects of Apache dataset described in Section VI. In Figure 7, the color of each cell

TABLE II
EXECUTION TIME OF UPDATING TARGET PROJECT INCREMENTALLY

	CCFinder	Our Approach
Average Time	11,207s	1,006s

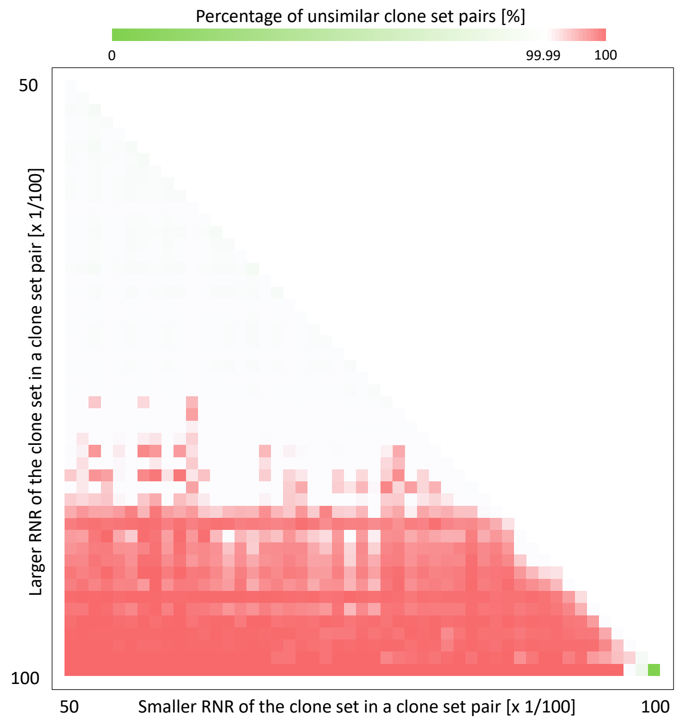


Fig. 7. The Percentage of Similar Clone Set Pairs

indicates a percentage of the clone set pairs whose X-axis and Y-axis clone sets are not similar than 0.9. The while cells are the percentage is 99.99%. The red cell shows the percentages are higher than 99.99% while the green cell means that the percentage is lower than 99.99%. We considered that clone sets in a clone set pair are not similar when the RNR values of the clone sets are largely different from each other and the RNR value of either clone set in the pair is close to large. Therefore, our approach omits the similarity calculations of such clone set pair using two parameters, θ_{diff} and θ_{max}.

In this research question, we seek for appropriate values of two parameters, θ_{diff} and θ_{max} to curate clones rapidly. We attempted to compare the execution time and the precision of clone curation for each of the parameter combinations. However, it takes too long time to curate clones for all combinations of parameters. Hence, we measured the number of similarity calculations instead of the execution time. Namely, by comparing the number of similarity calculations and the precision of clone curation, we evaluated the effectiveness of our approach to omit similarity calculations.

Figure 8(a) is a heatmap which shows the percentage of clone set pairs curated for each parameter. The while cells indicate that the percentage of curated clone set pairs is 90%. The green cell shows that the percentages are higher than 90% while the red cell means that the percentage is lower than 90%. Figure 8(b) is another heatmap which shows how much similarity calculations are omitted for each pair of parameters, θ_{diff} and θ_{max}. Each cell shows that the percentage of the similarity calculations with the given two parameters against the similarity calculations without omitting any calculations. The while cells indicate that the percentage of similarity calculations is 60%. The green cell shows that the percentages are higher than 60% while the red cell means that the percentage is lower than 60%. We can see that there are several value pairs of two parameters where their cells in both Figure 8(a) and Figure 8(b) are white. Namely, there are parameters which can reduce the number of similarity calculations to 60 percents with keeping the 90 percents of curated clone set pairs. Thus, we sought for the optimal pair of parameters using the heatmaps and we measured the number of similarity calculations that could be omitted when the optimized parameters were utilized.

Figure 9 shows the relationship between the ratio of curated clone set pairs and the ratio of similarity calculation reduction. According to Figure 7, we can see that there are multiple pairs of the two thresholds that yield the same ratio of curated clone set pairs. Herein, we focus on the pair of the two parameters that reduces similarity calculation at a maximum, and we investigated the relationship between the two ratios. As a result, we found that 41% of similarity calculations can be omitted by sacrificing 5% of the curations.

Our answer to RQ3 is that using appropriate values of parameters can reduce 41% of similarity calculations only by sacrificing 5% of the clone curations.

VIII. Threats of Validity

A. Clone detector

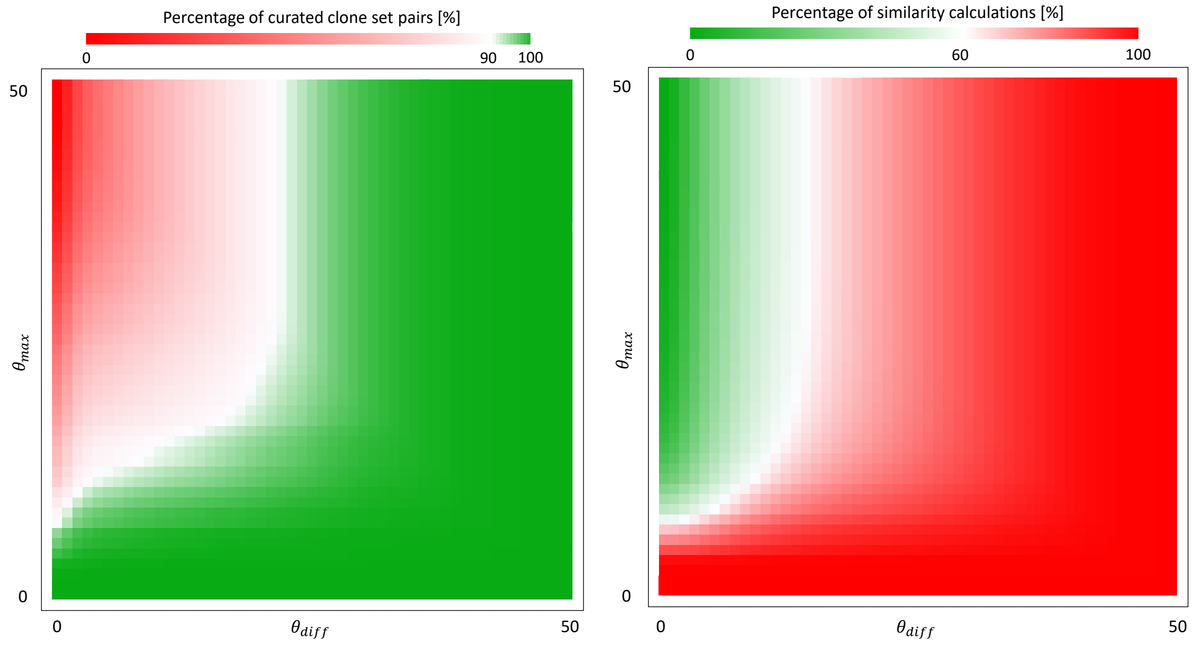
In the evaluations, we utilized CCFinder as a clone detector. However, there are many clone detectors besides CCFinder. Applying other clone detectors in our approach does not necessarily get more rapid than conventional clone detections unlike CCFinder. We are planning on evaluating whether or not other clone detectors work well in our approach.

B. Experimental target

In the evaluations, we succeeded in reducing similarity calculations by 41% on the Apache projects dataset. However, if we use other datasets, we may obtain different ratios of similarity calculations.

IX. Conclusion

In this research, we proposed a technique to detect clones rapidly. The proposed technique curates the results of clone detections for each of the projects. As a result of evaluating the performance of our technique, it was possible to detect clones up to 17 times faster with CCFinder. In addition, the execution time of incremental updates was able to be reduced to nine percents of previous ones. Furthermore, we showed that using appropriate values of parameters can reduce 41% of similarity calculations only by sacrificing 5% of the clone curations. We plan to improve our technique so that it can be applied to clone detectors other than CCFinder such as SourcererCC [9] or NiCAD [17] in the future.



(a) The Percentage of Curated Clone Set Pairs

(b) The Number of Similarity Calculations

Fig. 8. Heatmap of Curated Clone Set Pairs and Similarity Calculations for Pairs of Each Parameter

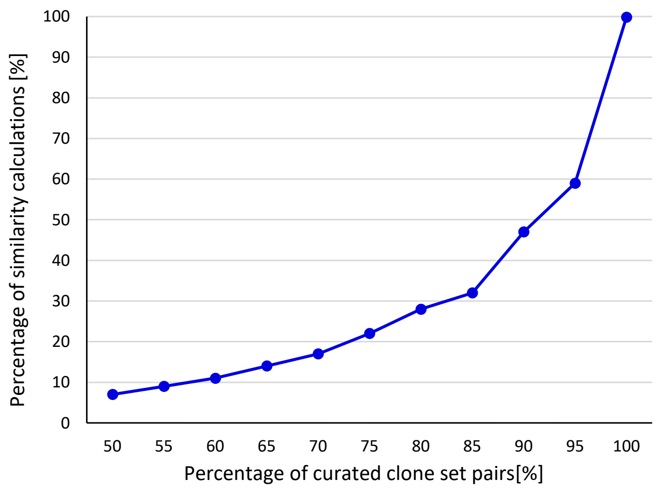


Fig. 9. Transitions of Similarity Calculations and Curated Clone Set Pairs

References

[1] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *2008 IEEE International Conference on Software Maintenance*, 2008, pp. 227–236.

[2] M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, and K. A. Schneider, "An empirical study of the impacts of clones in software maintenance," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 242–245.

[3] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?" in *Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 72–81.

[4] T. Zhang and M. Kim, "Automated transplantation and differential testing for clones," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 665–676.

[5] A. Sheneamer and J. Kalita, "A survey of software clone detection techniques," *International Journal of Computer Applications*, vol. 137, no. 10, pp. 1–21, 2016.

[6] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 175–186.

[7] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle, "Software bertillonage: finding the provenance of an entity," in *Proceedings of the 8th working conference on mining software repositories (MSR)*, 2011, pp. 183–192.

[8] C. Ragkhitwetsagul and J. Krinke, "Siamese: scalable and incremental code clone search via multiple code representations," *Empirical Software Engineering*, pp. 1–49, 2019.

[9] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 1157–1168.

[10] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Method and implementation for investigating code clones in a software system," *Information and Software Technology*, vol. 49, no. 9, pp. 985–998, 2007.

[11] A. Goto, N. Yoshida, M. Ioka, E. Choi, and K. Inoue, "How to extract differences from similar programs? a cohesion metric approach," in *2013 7th International Workshop on Software Clones (IWSC)*, May 2013, pp. 23–29.

[12] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler, "Automatic clone recommendation for refactoring based on the present and the past," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 115–126.

[13] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[14] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proceedings of the 2005 ACM Symposium on Applied Computing*, ser. SAC '05, 2005, pp. 314–318.

[15] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *In Proceedings of the Early Research Achievements track of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014)*, 2014, pp. 476–480.

[16] Y. Higo and S. Kusumoto, "How should we measure functional sameness from program source code? an exploratory study on java methods," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 294–305.

[17] J. Cordy and C. Roy, "The nicad clone detector," *IEEE International Conference on Program Comprehension*, pp. 219–220, June 2011.

An Experiment to Compare Combinatorial Testing in the Presence of Invalid Values

Konrad Fögen
Research Group Software Construction
RWTH Aachen University
Aachen, NRW, Germany
foegen@swc.rwth-aachen.de

Horst Lichter
Research Group Software Construction
RWTH Aachen University
Aachen, NRW, Germany
lichter@swc.rwth-aachen.de

***Abstract*—Robustness is an important property of software that should be thoroughly tested. Combinatorial testing (CT) is an effective black-box test approach. When using it for robustness testing, input masking can prevent faults from being detected. However, the impact is not yet clear. Therefore, we conducted a controlled experiment to understand how input masking affects the fault detection effectiveness of CT and how effective CT is in the presence of error-handling and invalid values.**

***Keywords*-Software Testing, Combinatorial Testing, Robustness**

I. Introduction

Robustness is an important property of software systems that describes "the degree to which a system or component can function correctly" in the presence of external faults like invalid inputs [1]. External faults can have a severe impact on the system's robustness because they can propagate to system failures resulting in abnormal behavior or system crashes [2]. To improve robustness, systems implement error-handling to appropriately react to external faults [3]. Oftentimes, the external fault cannot be resolved by the system internally [4]. Then, the system is terminated by the error-handling procedure that returns an error-message to the client without executing the normal procedure. This is also referred to as the *error-propagation strategy* [5]. Unfortunately, error-handling procedures have a fault density that is up to three times higher compared with normal procedures [6]. Therefore, testing is important to check error-handling.

The purpose of testing is to reveal failures by stimulating a system under test (SUT) with test inputs and observing the results via test oracles [2]. To reveal a failure, a fault must be triggered to produce an error and the error must propagate to a failure of the SUT [7]. Assuming that test oracles reveal all propagated failures, the important factor in testing is the selection of test inputs such that the faults are triggered.

Combinatorial testing (CT) is a black-box approach for test input selection [8]. A test model describes the SUT via input parameters and input values. Using a combination strategy, test inputs are generated such that they satisfy a combinatorial coverage criterion like *t-wise* that is satisfied if all value combinations of t parameters appear in at least one test input.

Combinatorial robustness testing (CRT) is an extension to CT that incorporates robustness testing [5]. It is argued that CRT is necessary because of the *input masking effect* [5], [9]–[11]: The first invalid value that is evaluated by the SUT triggers error-handling and the normal control-flow is left. When the error-propagation strategy is used, the SUT returns with an error-message and the normal control-flow is not resumed. Then, the other values and value combinations of the test input remain untested because they are *masked*.

CRT avoids input masking by separating the testing with valid test inputs, that do not contain any invalid value, from testing with strong invalid test inputs, that contain exactly one invalid value. Therefore, the test model must be enriched with additional semantic information about invalid values. Previous experiments [5] have shown that CRT is an effective approach in the presence of error-handling.

Despite the presence of error-handling, CT can also reveal failures without requiring additional semantic information. However, the extent to which the input masking effect can impact the fault detection effectiveness is is yet unknown. Therefore, our aim is to answer the following research question: **How effective is CT in triggering faults when error-handling and invalid values are present?** To answer the research question, we apply the t-factor fault model, derive influencing factors and conduct a controlled experiment.

The paper is structured as follows. First, an example is introduced. Then, Section III and IV summarize background and related work. In Section V, the t-factor fault model is applied to the context of error-handling and factors that influence fault triggering are identified. Afterwards, the experiment design is discussed in Section VI and the results are presented in Section VII. Afterwards, threads to validity are discussed and we conclude with a summary of our work.

II. Example

To illustrate the impact of error-handling, we use a customer registration service as an example. The example consists of three validity checks to ensure that the entered data is not invalid. Since the service cannot correct the data itself, an error code is returned to the client asking to correct the data.

A test model for CT is depicted in Figure 1 with `123` representing some invalid value. Each test input that contains an invalid value like `[Title:123]` should yield an error code.

$$\begin{array}{ll} p_1 : Title & V_1 = \{Mr, Mrs, 123\} \\ p_2 : FamilyName & V_2 = \{Miller, Davis, 123\} \\ p_3 : Address & V_3 = \{UK, US, 123\} \end{array}$$

Figure 1. Example Test Model

Further, assume that the implementation of the service contains a fault in the validity check for family names that returns a wrong error code (title instead of name error). It is triggered whenever an invalid family name, e.g. `[FamilyName:123]`, is evaluated. An implementation is illustrated in Listing 1 with `INV_xxx` string literals representing specific error codes.

```
String register(String title, family, addr){
 if(isInvTitle(title)) return INV_TITLE;
 if(isInvFamilyName(family)) return INV_TITLE;
 if(isInvAddress(addr)) return INV_ADDRESS;
 ...
}
```

Listing 1. Example of an Input Validity Check

A test input like `[Title:Mrs, FamilyName:123, Address:UK]` would trigger the fault. In contrast, a test input like `[Title:123, FamilyName:123, Address:UK]` would yield `INV_TITLE` because of the invalid title. But, it would not trigger the name check fault because of input masking.

To satisfy 1-wise coverage, a minimal set of three test inputs is generated with exactly one test input that contains `[FamilyName:123]`. In total, nine test inputs with `[FamilyName:123]` exist and the combination strategy must select one. Of the nine test inputs, three contain `[Title:123]` causing input masking. Therefore, the probability of triggering the fault is $\frac{6}{9} = 0.66\%$.

Increasing the testing strength to $t > 1$ will also increase the fault triggering probability. However, finding a minimal set of test inputs that satisfies t-wise coverage for $t \geq 2$ is in general NP hard [12]. Heuristics are used as combination strategies which produce small but not always minimal sets of test inputs. Then, some t-sized value combinations appear more than once and the probability cannot be simply calculated.

III. Background

A. Combinatorial Testing

CT is a well-known approach to black-box testing where test inputs are selected based on a test model [8]. The **test model** TM describes the input space of a program as a set of n parameters $P = \{p_1, ..., p_n\}$ and the domain of each parameter p_i is a finite nonempty set of m_i values $V_i = \{v_1, ..., v_{m_i}\}$. A **combination** τ is a set of $0 < d \leq n$ parameter-value pairs (p_i, v_j) for d distinct parameters p_i with $v_j \in V_i$. A **test input** is a combination of size $d = n$. Combination τ_a covers another combination τ_b if every parameter-value pair of τ_b is included in τ_a which we denote as $\tau_b \subseteq \tau_a$.

In CT, coverage criteria and combination strategies depend on a specific test model TM [8]. A **coverage criterion** C describes requirements of TM that must be met by a set of test inputs T and a **combination strategy** describes how to select test inputs T such that C is satisfied.

The **t-wise coverage criterion** is a common criterion that is satisfied if all value combinations of t parameters appear in at least one test input [8]. In addition, all smaller combinations ($d < t$) are covered and also some larger combinations of size $t' = (t + k)$ with $k > 0$ and $t' \leq n$ are covered [13]. This so-called **collateral coverage** can potentially help triggering additional faults [14].

The input domains of real-world systems are typically restricted. As a consequence, certain values or value combinations are not of any interest or may prevent a test from being executed. **Exclusion-constraints** are commonly used to exclude irrelevant value combinations from test input selection [5]. Every test input that satisfies the exclusion-constraints is a **relevant** test input.

B. Combinatorial Robustness Testing

CRT is an extension to CT that incorporates robustness testing [5]. It explicitly considers the input masking effect that is caused by error-handling. To avoid input masking, CRT separates the generation of valid and invalid test inputs.

Therefore, additional semantic information is required to model invalid values [5]. The additional information can be modelled via **error-constraints** which denote a second set of constraints [5]. Then, relevant test inputs are further partitioned as follows. A relevant test input is a **valid test input** if it satisfies all exclusion- and all error-constraints. A relevant test input is **invalid** if it satisfies all exclusion-constraints but at least one error-constraints remains unsatisfied. An invalid test input is denoted a **strong invalid test input** if exactly one error-constraint is unsatisfied.

Then, valid test inputs and invalid test inputs are generated separately such that they satisfy different coverage criteria. The **valid t-wise coverage criterion** is satisfied if each valid parameter value combination of size t appears in at least one test input τ of which all other values and value combinations are also valid [5], [8]. In addition, the **single error coverage criterion** is satisfied if each invalid value appears in at least one strong invalid test input τ of which all other values and value combinations are valid [5], [8].

When satisfying the aforementioned coverage criteria, both normal procedures and error-handling are tested without input masking. However, in comparison to CT, additional effort is required to model the error-constraints.

IV. Related Work

Cohen et al. [9] first described the input masking effect caused by error-handling [8]. They also noted the need to separate valid and invalid test inputs to avoid input masking. An evaluation of combination strategies by Grindal et al. [11] discussed an example where input masking prevented a fault from being triggered. A case study by Wojciak and Tzoref-Brill [15] reported on CT including testing with invalid inputs.

The CT tools AETG [9], ACTS [16] and PICT [10] allow to mark individual values as invalid and also generate separate sets of test inputs. However, invalid value combinations are not directly supported.

In previous work [5], [17], we introduced error-constraints as a modelling technique that allows to directly model both invalid values and invalid value combinations. We also conducted experiments that compared CRT with CT. But, they focused on configuration-dependent faults where the error-handling depends on a certain configuration of valid parameter combinations. Further, we discussed techniques to identify and explain over-constrained test models [18] and discussed a technique to semi-automatically repair over-constrained test models [19].

In a case study [20], we analyzed bug reports of a software for life insurances. 51 out of 212 analyzed bug reports describe robustness faults. Many of them were triggered by invalid value combinations and we concluded that it is not sufficient for a CT tool to only consider invalid values.

Despite the conclusion, we only consider invalid values in this experiment because it allows a clearer separation between valid and invalid values when extending a given test scenario.

Other empirical studies also compared the efficiency of CT. A recent study summarizes previous comparisons [21]. However, none of these studies focused on error-handling.

V. Applying the t-factor Fault Model in the Presence of Error-Handling

A. Overview

The idea of the t-wise coverage criterion is based on the corresponding **t-factor fault model** which is formally introduced by Dalal and Mallows [22]. In general, a fault model is a description of hypothesized faults. The t-factor fault model relies on a transformational model of the SUT where the output is defined in terms of its input [23]. The input is modelled as a set of parameters $p_1, ..., p_n$ with each parameter p_i having a domain D_i consisting of a potentially infinite number of values.

The faults are defined in terms of the SUT's input. It is assumed that faults are caused by the interaction of t parameters and a t-factor fault is triggered by a combination of t parameter values. A t-factor fault can be described by a condition over t parameters which must be satisfied by an input to the SUT in order to trigger the fault. Each input that satisfies the condition triggers the t-factor fault.

The t-factor fault model is researched empirically [20], [24]–[29] where bug reports for different types of software are analyzed. An interaction rule is derived from the empirical findings which states that "only a few factors are involved in failure-inducing faults in software. Most failures are induced by single factor faults or by the interaction of two factors; progressively fewer failures are induced by interactions between three, four, or more factors. The maximum degree of interaction in actual faults so far observed is six" [30].

Since the t-wise coverage criterion is defined relative to the test model, the SUT model and test model share the same set of input parameters. While the domain D_i of a SUT input parameter p_i is potentially infinite, the domain $V_i \subseteq D_i$ of a test model parameter p_i is a finite nonempty subset of values that contains all values a tester is interested in.

When testing with a test suite that satisfies the t-wise coverage criterion, all parameter value combinations of size t appear in some test input. Testing should fail for each SUT that contains t'-factor faults with $t' \leq t$ if the values of the test model are selected properly such that the condition of the t'-factor faults can be satisfied. Therefore, CT is also called *pseudo-exhaustive testing* implying that t-wise testing is as good as exhaustive testing for a particular class of software with faults of factor t or smaller [26].

A test input τ that triggers a t-factor fault contains a combination c that is a **failure-inducing combination** (FIC). Each test input τ that covers $c \subseteq \tau$ triggers a fault [31]. A FIC c is **minimal** (MFIC) if no proper subset $c' \subset c$ triggers a fault. The size of a MFIC is predetermined by the t-factor fault and its condition.

When applying the t-factor fault model to faults in the presence of error-handling, characteristics that affect the capability of triggering faults can be derived. The characteristics either affect the t-factor faults or the FICs that trigger them. They are discussed in the following two subsections.

Recall the fault of the example implementation (Figure 1) that is triggered whenever an invalid family name is evaluated. To describe it as a t-factor fault, the error-handling must be taken into account. It is not a 1-factor fault because not every input that satisfies `isInvFamilyName(family)` triggers the fault. Consequently, `[FamilyName:123]` is not a FIC. Triggering the fault requires a valid title because the error-handling `isInvTitle(title)` propagates otherwise. Therefore, the fault can be modelled as a 2-factor fault using a conjunction over `title` and `family`: $\neg$(`isInvTitle(title)`) $\wedge$ (`isInvFamilyName(family)`).

For the given test model, the combinations `[Title:Mr, FamilyName:123]` and `[Title:Mrs, FamilyName:123]` are minimal failure-causing.

B. Characteristics affecting Size of t-Factor Faults

1) Number of Parameters involved in Error-Handling: In the presence of error-handling, the condition to trigger a fault can be formulated as a conjunction of two sub-conditions. First, the location of an incorrect error-handler must be *reached* [7] by ensuring that no prior error-handler terminates the SUT. We denote this as the **prevention sub-condition**. Second, an invalid value must cause an error-handler to produce an incorrect program state (*infection* [7]) that can propagate to a failure. We denote this as the **infection sub-condition**. For the example, $\neg$(`isInvTitle(title)`) is used for prevention and (`isInvFamilyName(family)`) is used for infection.

The size of a t-factor fault increases with the number of parameters involved in prevention and infection sub-conditions. To guarantee that a t-factor fault is triggered, the test input set must satisfy t-wise coverage of the same size.

2) Priority of Error-Handlers: Each error-handler with an earlier position in the control-flow has the potential to terminate the SUT before the incorrect error-handler is reached. Therefore, each prior error-handler increases the prevention sub-conditions.

A fault in the first error-handler of the example would be modelled by an empty prevention sub-condition. In contrast, a fault in the third error-handler would be modelled by a prevention sub-condition that includes both prior error-handlers, e.g. $\neg$(`isInvTitle(title)` $\vee$ `isInvFamilyName(family)`).

However, the modelled fault depends on a specific implementation whereas CT is a black-box approach which depends on the SUT's specification instead. While testing with 2-wise coverage is sufficient to detect a fault in error-handling that can be modelled as a 2-factor fault, it requires the location of the incorrect error-handler to be known beforehand in order to determine the appropriate testing strength of $t = 2$.

The specification does typically not impose a specific order of error-handling. For instance, an implementation that checks the validity of the address first is as correct as the implementation shown in Listing 1 where the address is checked last. Then, the fault in the validity check of the family name becomes a 3-factor fault.

To make the determination of testing strength t independent from the location of an incorrect validity check within the control-flow, all error-handlers in every possible order must be taken into account. Then, t would grow with the number of parameters checked by error-handlers.

Using a testing strength of $t = 3$ ensures that the fault is triggered for all possible orderings of the error-handlers. Thereby, the prior knowledge about the incorrect error-handler is also abandoned. By deriving the testing strength from all parameters that are involved in any error-detection condition, it is ensured that each error-handler is reached and potential faults are triggered.

While this testing strength denotes the lower limit to ensure that potential faults are triggered independently from the ordering of error-handlers, testing is still conducted for a specific implementation with a specific ordering. Therefore, we distinguish the *effective* prevention sub-condition which is sufficient for a specific implementation from the *general* prevention sub-condition that is sufficient for all orderings. On average, the effective prevention sub-condition considers fewer error-handlers which improves the likelihood of triggering a fault when using a testing strength that is not sufficient for the general prevention sub-condition.

C. Characteristics affecting the Number of Minimal Failure-inducing Combinations

1) Number of Valid Values: Given a test model and a t-factor fault f, a parameter p is involved if the condition that describes f includes p. Otherwise, p is not involved. For the example, parameters `Title` and `FamilyName` are involved in the condition $\neg$`(isInvTitle(title))` $\wedge$ `(isInvFamilyName(family))` while parameter `Address` is not involved.

In the example, two MFICs of size $t = 2$ exist that trigger the same fault. A test suite that satisfies 2-wise coverage includes each MFIC at least once and guarantees that the fault is triggered. Since two MFICs trigger the same fault, the probability of selecting one of them is increased when testing with $(t' < 2)$-wise collateral coverage. A set of test inputs that only satisfies 1-wise coverage has a probability of at least $\frac{2}{3} = 0.66\%$ to trigger the fault, i.e. at least one test input covers `[FamilyName:123]` and there is a $\frac{2}{3}$ chance that `[Title:123]` is not covered by the same test input.

The effect of values can be distinguished depending on whether or not the value's parameter is involved in the infection or prevention sub-condition.

A valid value or valid value combination that is involved in the infection sub-condition does not affect the failure-inducing combinations because satisfying the infection sub-condition and being valid are mutually exclusive. For instance, adding another valid family name `[FamilyName:Smith]` to the example test model does not affect the FICs because it cannot satisfy `(isInvFamilyName(family))`.

But, valid values and valid value combinations of parameters that are involved in the prevention sub-condition increase the number of MFICs. For instance, adding another valid value `[Title:Sir]` to the example results in another MFIC `[Title:Sir, FamilyName:123]`. For 1-wise testing, the probability to trigger the fault increases to $\frac{3}{4} = 0.75\%$.

Valid values and valid value combinations of parameters that are not involved in the prevention and infection sub-condition of a t-factor fault do not directly affect the MFICs. They contribute to the set of t-sized parameter values combinations that must be covered by some test input to satisfy t-wise coverage, though.

For our example, nine test inputs can be created that cover `[FamilyName:123]`, i.e. $\{Mr, Mrs, 123\} \times \{UK, US, 123\}$. Since three of them cover `[Title:123]` and do not satisfy the effective prevention sub-condition, the probability of selecting a test input that covers a MFIC are $\frac{6}{9} = 0.66\%$. When another valid address is added, 12 test inputs that cover `[FamilyName:123]` can be created and four of them do not satisfy the effective prevention sub-condition. The probability of triggering the fault remains the same.

It has to be noted that additional values may affect the overall selection of test inputs. Maybe a combination strategy cannot find a small test suite for the given values or another reason that is inherent to the combination strategy increases or decreases redundancy and thus affects the fault triggering probability. These effects are beyond the scope of this paper.

2) Number of Invalid Values: Invalid values of parameters that are involved in the condition of a t-factor fault f can increase or decrease the probability of triggering f. It depends on whether the parameters are involved in the prevention or infection sub-condition.

When the parameter of the invalid value is involved in the prevention sub-condition, the probability of input masking is increased. For the example, adding another invalid value `[Title:456]` decreases the probability of selecting a test input that covers one of the two MFICs to $\frac{2}{4} = 0.50\%$.

When the parameter of the invalid value is involved in the infection sub-condition, the probability of input masking is decreased. For instance, adding another invalid value `[FamilyName:456]` that triggers the same fault as `[FamilyName:123]` adds two more MFICs `[Title:Mr, FamilyName:456]` and `[Title:Mrs, FamilyName:456]` to the example.

Invalid values of parameters that are not involved in t-factor fault f only exist for effective prevention sub-conditions because general prevention sub-conditions consider all error-handlers. As an example, consider an additional invalid address `[Address:456]`. From the perspective of general prevention sub-conditions, they can be treated as above invalid values that increase the probability of input masking. From the perspective of effective prevention sub-conditions, the invalid values do not affect the MFICs because the evaluation would happen after the evaluation of the incorrect error-handler.

VI. Experiment Design

A. Test Scenarios

The objective of our experiment is to evaluate the effectiveness of CT when generating test inputs in the presence of error-handling. Especially, we want to determine in which cases CT is sufficient enough such that CRT and its additional effort can be avoided. Therefore, we generated test inputs with different characteristics and executed them in test scenarios. The source code and the results of the experiment are available at our companion website[1].

For an experiment, it is important to control the factors that can influence the results. Therefore, we designed artificial test scenarios and changed different characteristics in a controlled and traceable way. Each test scenario contains exactly one faulty error-handler which always propagates to a failure when triggered and the failure is always revealed by the test oracle. Thus, the selection of test inputs is the only factor that influences the results of test execution.

The implementation of a test scenario is illustrated in Listing 2. A test scenario accepts input values for a number of parameters and includes a sequence of error-handlers with one error-handler for one parameter implemented by an `if` statements. The result of a test scenario execution is `INV_INPUT` if an error-handler correctly identifies an invalid value and terminates the execution. If an invalid value is identified by a faulty implemented error-handler, `NULL` is returned instead; `VAL_INPUT` is returned if all error-handlers are passed because all values are valid.

```
String checkInput (Object a, b, c){
  if(isInvalid(a)) return INV_INPUT;
  else if(isInvalid(b)) return NULL;
  else if(isInvalid(c)) return INV_INPUT;
  else return VAL_INPUT;
}
```

Listing 2. Illustration of a Test Scenario Implementation

[1]https://github.com/coffee4j/quasoq-2019

Table I
Test Suite Sizes used for the Experiment

P	V	t=1	t=2	t=3	t=4	t=5
6	2	2	6	14	26	42
6	3	3	15	49	140	318
6	4	4	24	109	442	1377
6	5	5	38	209	1059	4195
6	6	6	52	361	2165	10407
12	2	2	8	19	47	105
12	3	3	20	71	262	885
12	4	4	31	170	835	3854
12	5	5	48	329	2068	11889
12	6	6	67	564	4295	29645
18	2	2	10	24	60	144
18	3	3	20	88	336	1241
18	4	4	36	202	1098	5458
18	5	5	54	401	2725	16843
18	6	6	75	689	5650	42102
24	2	2	10	26	71	175
24	3	3	21	98	396	1513
24	4	4	40	232	1298	6629
24	5	5	59	454	3203	20482
24	6	6	83	786	6662	51287
30	2	2	11	28	80	200
30	3	3	23	106	446	1715
30	4	4	41	255	1448	7555
30	5	5	63	495	3589	23369
30	6	6	86	859	7473	58468

Based on the application of the t-factor fault model, we describe each test scenario S in terms of (1) the number of parameters, (2) the number of parameters that are involved in error-handling, (3) the number of valid values, (4) the number of invalid values per parameter, and (5) the index of the position of the incorrect error-handler that contains a fault.

The illustrated test scenario uses 3 parameters where the second error-handler(index $i = 1$) is incorrect. The number of valid and invalid values per parameter is implicitly encoded by the test model that is used to generate test inputs.

By varying the index of the incorrect error-handler, three different test scenarios for our example can be created, where either the first, second or third error-handler is incorrect. A set of test scenarios, that shares the same parameters and values but differs in the index of the incorrect error-handler is called a **test scenario family** S^*.

As a notation, we use `P-V-I-E` where `P` refers to the number of parameters involved in error-handling, `V` refers to the number of valid values per parameter, `I` refers to the number of invalid values per parameter, and `E` refers to the number of parameters that are involved in error-handling.

The experiment starts with a *root* test scenario family `6-1-1-6` representing a simple application of CT. The root test scenario family is then extended as follows. The number of parameters is increased by six up to 30 parameters. The number of error-handlers in a test scenario family either remains at six or is equal to the number of parameters. The total number of number of values per test scenario family is extended up to six with one to five valid and invalid values.

B. Test Input Generation

The test models used in this experiment share the same set of parameters with the test scenario and define the number of valid and invalid values per parameter. To avoid any bias, we use test suites from the NIST Covering Array Tables[2]. They are publicly available and contain many of the smallest known test suites [32]. Table I depicts the sizes of the test suites. Column *P* refers to the number of parameters and column *V* refers to the total number of values per parameter. The test suites are reused for different ratios of valid and invalid values.

The order of parameters and values in a test model has no impact on whether or not a generated test suite satisfies t-wise coverage. However, it has an impact on which t-wise parameter value combinations are combined in a single test input. To reduce the effect of accidental fault triggering that is caused by ordering, the parameters and values of a test suite are randomly reordered and 100 different variants of each test suite are generated. The set of all test suite variants is called a **test suite family** T^*.

The testing strengths used in the experiment range from $t = 1$ to $t = 5$ because most failures are induced by this range, according to the interaction rule.

C. Evaluation Metrics

A common metric to evaluate combination strategies is called **Fault Detection Effectiveness** (FDE) [8], [14].

A test suite T is denoted as **failing** for a test scenario S if at least one of the test inputs $\tau \in T$ triggers the t-factor fault $f \in S$ and the test suite consequently fails.

$$\textbf{failing}(T, S) = \begin{cases} 1 \text{ if } \exists \tau \in T \text{ that fails for } S \\ 0 \text{ otherwise} \end{cases} \quad (1)$$

Using the failing function, FDE is defined as the ratio between the number of test suites $T \subseteq T^*$ of a family that fail for a test scenario S and the number of all test suites in a family T^* that is used to test S.

$$\textbf{FDE}(T^*, S) = \frac{\sum_{T \in T^*} \text{failing}(T, S)}{|T^*|} \quad (2)$$

In other words, the FDE is based on randomized variants of a test suite that all satisfy the same testing strength. They all are used to test the same test scenario S which has a fixed incorrect error-handler. While this metric can be used to identify characteristics that may influence the FDE, the information cannot be used in practice because one must know which error-handler is incorrect.

Therefore, we introduce the **average fault detection effectiveness** (AFDE) which is the average FDE over a family of test scenarios S^*. Thus, AFDE represents the effectiveness of a test scenario family when knowing that one error-handler is incorrect but without knowing its index.

$$\textbf{AFDE}(T^*, S^*) = \frac{\sum_{S \in S^*} \text{FDE}(T^*, S)}{|S^*|} \quad (3)$$

[2]https://math.nist.gov/coveringarrays/

VII. Results & Discussion

A. Overview

The overall results of the experiment are consistent with the application of the t-factor fault model. Whenever the first error-handler is incorrect, the prevention sub-condition is empty and one parameter is involved in the infection sub-condition. The resulting 1-factor fault is triggered in each test scenario by all test suites that satisfy the testing strength $t = 1$.

Whenever an error-handler at a higher index is incorrect, the prevention sub-condition includes the parameters checked by all error-handlers with lower indices. Since one parameter is involved in the infection sub-condition, the faults can be described as $(\text{index} + 1)$-factor faults, where the index of the first error-handler is 0. For all considered testing strengths $(1 \leq t \leq 5)$, all $(\text{index} + 1)$-factor faults are triggered by all test suites that satisfy the corresponding testing strength.

Beyond that, collateral coverage causes higher $(\text{index} + 1)$-factor faults to be repeatedly triggered by test suites that satisfy lower testing strengths.

B. Fault Detection Effectiveness

Table II depicts an excerpt of FDEs computed for test scenario families with six to 30 parameters consisting of two valid values and one invalid value. Each test scenario family contains six error-handlers. The *I* column denotes the index of the incorrect error-handler, *t* denotes the testing strength that is satisfied by the family of test suites and the remaining columns denote the computed FDE values.

According to the t-factor fault model, a testing strength of $t = 2$ is required to guarantee that an incorrect error-handler with index = 1 is detected. Among all depicted test scenario families, the fault is on average also triggered by 67.6% test suite families that only satisfy $t = 1$. The exact numbers are depicted in the second row of Table II.

Testing with higher testing strengths is even more effective. On average, test suite families that satisfy a testing strength of $t = 2$ detects incorrect error-handlers at index = 2 in 99.2% of all cases. As expected, a family of test suites that satisfies $t = 3$ always detects the incorrect error-handlers at index = 2. However, incorrect error-handlers at indices 3 and 4 are always detected as well. The incorrect error-handlers at index 5 are detected by 98.6% of all test suite families.

When increasing the number of parameters while the number of error-handlers remains at six, the data indicates no general trend for the FDE. In many cases, the FDE improves slightly. Although, there are cases where the FDE deteriorates. For instance, the FDE for detecting an incorrect error-handler at index 1 with a test suite family that satisfies only testing strength $t = 1$ improves from 64% for the test scenario family with six parameters to 74% for the test scenario family with 12 parameters. But, it deteriorates to 65% for the test scenario family with 18 parameters.

Table III depicts the FDE for test scenario families with six to 30 parameters and an equal number of error-handlers. Increasing the number of error-handlers such that one error-handler exists for each parameter has no direct impact on the

Table II
FDE for Test Scenario Families with six to 30 Parameters and six Error-Handlers (Excerpt)

I	t	6-2-1-6	12-2-1-6	18-2-1-6	24-2-1-6	30-2-1-6
0	1	100	100	100	100	100
1	1	63	74	65	66	70
1	2	100	100	100	100	100
2	1	43	34	38	35	52
2	2	98	99	100	99	100
2	3	100	100	100	100	100
3	1	30	35	33	29	31
3	2	90	96	95	95	99
3	3	100	100	100	100	100
4	1	22	22	20	19	27
4	2	72	82	84	85	91
4	3	100	100	100	100	100
5	1	12	14	11	13	18
5	2	61	61	69	63	69
5	3	95	99	100	100	99
5	4	100	100	100	100	100

Table III
FDE for Test Scenario Families with Six to 30 Parameters and an Equal Number of Error-Handlers (Excerpt)

I	t	6-2-1-6	12-2-1-12	18-2-1-18	24-2-1-24	30-2-1-30
0	1	100	100	100	100	100
1	1	63	63	66	77	66
1	2	100	100	100	100	100
2	1	43	48	55	39	49
...	...	...	...	...	...	...
7	3		78	86	91	87
9	3		52	52	61	55
9	4		95	96	98	99
11	3		34	29	30	34
11	4		67	78	88	83
11	5		99	100	100	100
17	3			2	2	2
17	4			13	16	16
17	5			38	36	44
23	4				2	2
23	5				3	5

FDE. Over all indices, testing strengths and parameter sizes, the difference between the FDE for six error-handlers and the FDE for $|P|$ error-handlers is 1.25 percentage points on average. For instance, the difference for $I = 1$, $t = 1$ and 12 parameters is 11 percentage points with an FDE of 74% for 6 error-handlers (Table II) and an FDE of 63% for 12 error-handlers (Table III).

The biggest deviations are noticed for test scenario families with 30 parameters. For $I = 1$ and $t = 1$, the difference between the FDE for six error-handlers (57%) and the FDE for 30 error-handlers (42%) is 15 percentage points. For $I = 5$ and $t = 2$, the difference between the FDE for six error-handlers (40%) and the FDE for 30 error-handlers (60%) is -20 percentage points.

Analysing higher indices of incorrect error-handlers ($I > 5$) emphasizes that the required testing strength to detect a fault increases as well. Although, the testing strength grows slower compared to lower indices ($I \leq 5$). For instance, $t = 3$ is sufficient to detect incorrect error-handlers at index $I = 4$ in 100% of all cases and it is almost sufficient (98.6% on average) to detect all incorrect error-handlers at index $I = 5$. For $I = 7$, the average FDE decreases to 85.5%. For $I = 17$, only 2% of all incorrect error-handlers are detected. In comparison, $t = 4$ is sufficient to detect an incorrect error-handler at index $I = 7$ in 100% of all cases. $t = 5$ is sufficient to detect a fault for index $I = 10$ in 100% of all cases and even a fault for index $I = 11$ is detected in 99.75% of all cases. Afterwards, the FDE for $t = 5$ decreases to an average of 39.3% for index $I = 17$ and to 4% for $I = 23$.

Overall, the findings are consistent with the application of the t-factor fault model. Changing the number of parameters of a test scenario family has no clear effect on the FDE as well as changing the total number of error-handlers. In contrast, increasing or decreasing the specific index of the incorrect error-handler has the biggest effect on the FDE.

Following the t-factor fault model, a testing strength t guarantees to detect incorrect error-handlers up to index $I = t - 1$ because one parameter is involved in the infection sub-condition. Beyond that, the collateral coverage effect of CT causes a reliable detection of incorrect error-handlers with higher indices. Although for index $I = 11$ and higher, even $t = 5$ is not sufficient to detect incorrect error-handlers for the test scenario families depicted in Table III.

To use the knowledge acquired from FDE metric, knowledge about the index of an incorrect error-handler is required which makes it hard to use the information in practice. In contrast, the AFDE values allow to draw conclusions regarding the effectiveness of CT assuming that one error-handler is incorrect when only the number of checked parameters as well as the number of valid and invalid values is known. Therefore, all further analyzes are based on the AFDE metric.

C. Average Fault Detection Effectiveness

Table IV and Table V list the AFDE values for all test suite families and the testing strengths from $t = 1$ to $t = 5$. Column P denotes the number of parameters, column E denotes the number of error-handlers, and column t denotes the testing strength. *Avg.* depicts the average AFDE among all testing strengths. The remaining columns follow the pattern `V-I` with `V` describing the number of valid values and `I` describing the number of invalid values. Table IV contains the AFDE values for test scenario families with six error-handlers. In Table V, the number of error-handlers equals the number of parameters.

As discussed in the prior subsection, increasing the number of parameters only has no clear effect on the FDE. The AFDE metric reflects this as depicted by Table IV.

Increasing the total number of error-handlers had no impact on the FDE of existing error-handling indices. But, the FDE for the additional indices is worse because more parameters belong to the prevention sub-condition. Since AFDE represents the average FDE among all indices of incorrect error-handlers, the worse FDE of additional error-handlers decreases the AFDE value. This is depicted in Table V.

Changing the number of values per parameter has a great effect on the AFDE. Depending on whether the additional values are valid or invalid, the AFDE increases or decreases.

Table IV
AFDE OF ALL TEST SCENARIO FAMILIES WITH SIX ERROR-HANDLERS

P	**E**	**t**	**1-1**	**1-2**	**1-3**	**1-4**	**1-5**	**2-2**	**3-3**	**2-1**	**3-1**	**4-1**	**5-1**
6	6	1	36	33.17	34.83	34.17	33.83	47.5	53.33	45	53	63.33	65.83
6	6	2	55.33	53.83	53	53.33	52.33	79.5	94.67	86.83	97	99.5	99.83
6	6	3	71.83	71.5	73.33	70.83	70.67	98	99.83	99.17	100	100	100
6	6	4	86.67	86.67	86.5	85.33	86.33	100	100	100	100	100	100
6	6	5	95	96	96.33	97.33	98.33	100	100	100	100	100	100
6	6	*Avg.*	68.97	68.23	68.8	68.2	68.3	85	89.57	86.2	90	92.57	93.13
12	6	1	33.33	32.17	31.5	33	34	46.5	57.5	46.5	55.83	64	67.5
12	6	2	60	60.67	57.33	54.67	56	84.5	92.83	89.67	99	99.67	100
12	6	3	76.67	74.83	74.33	75	71.33	99.33	100	99.83	100	100	100
12	6	4	94.17	92.5	90.5	91	88.83	100	100	100	100	100	100
12	6	5	97.83	99.33	99.83	99.33	100	100	100	100	100	100	100
12	6	*Avg.*	72.4	71.9	70.7	70.6	70.03	86.07	90.07	87.2	90.97	92.73	93.5
18	6	1	33.33	32.83	33.33	32.83	34	43.83	52.17	44.5	57	58.33	65.83
18	6	2	62.33	58.83	57	56.67	53.5	86	94.5	91.33	97.5	100	100
18	6	3	80.83	79.67	77.33	74	73.83	99.5	100	100	100	100	100
18	6	4	94	94.17	94.17	92.17	90.5	100	100	100	100	100	100
18	6	5	99.67	100	100	100	100	100	100	100	100	100	100
18	6	*Avg.*	74.03	73.1	72.37	71.13	70.37	85.87	89.33	87.17	90.9	91.67	93.17
24	6	1	31.33	33.33	33	33.5	32.67	46.17	52.33	43.67	53.67	60.33	66.5
24	6	2	63.17	59.83	58.17	57.33	55.5	85.83	94.17	90.33	98.33	100	100
24	6	3	80.67	77.67	77.5	75.67	75	99.67	100	100	100	100	100
24	6	4	94.83	94.67	94.33	91.5	91.83	100	100	100	100	100	100
24	6	5	100	99.83	100	100	100	100	100	100	100	100	100
24	6	*Avg.*	74	73.07	72.6	71.6	71	86.33	89.3	86.8	90.4	92.07	93.3
30	6	1	35.33	34.5	33.17	33.83	32.33	45.83	51.67	49.67	54.17	60.83	69.17
30	6	2	65.17	61.67	57.17	57.17	56.67	84.83	95	93.17	98.83	100	100
30	6	3	84.33	83.5	79	77	72.5	100	100	99.83	100	100	100
30	6	4	95.83	96.67	93.5	91.33	93	100	100	100	100	100	100
30	6	5	99.83	99.83	100	100	100	100	100	100	100	100	100
30	6	*Avg.*	76.1	75.23	72.57	71.87	70.9	86.13	89.33	88.53	90.6	92.17	93.83

Table V
AFDE OF ALL TEST SCENARIO FAMILIES WITH INCREASING NO. OF ERROR-HANDLERS

P	**E**	**t**	**1-1**	**1-2**	**1-3**	**1-4**	**1-5**	**2-2**	**3-3**	**2-1**	**3-1**	**4-1**	**5-1**
6	6	1	36	33.17	34.83	34.17	33.83	47.5	53.33	45	53	63.33	65.83
6	6	2	55.33	53.83	53	53.33	52.33	79.5	94.67	86.83	97	99.5	99.83
6	6	3	71.83	71.5	73.33	70.83	70.67	98	99.83	99.17	100	100	100
6	6	4	86.67	86.67	86.5	85.33	86.33	100	100	100	100	100	100
6	6	5	95	96	96.33	97.33	98.33	100	100	100	100	100	100
6	6	*Avg.*	68.97	68.23	68.8	68.2	68.3	85	89.57	86.2	90	92.57	93.13
12	12	1	17	16	16.58	16.5	16.67	23.08	26.33	25.42	32.08	38.75	44.5
12	12	2	29.5	30	27.83	26.83	26.92	46.58	54.17	54.08	74.75	89.58	95.83
12	12	3	41.92	39.5	38.08	36.75	36.08	64.92	78.33	79.67	98.25	100	100
12	12	4	52	49.5	46.83	45.67	45.17	81.42	96.75	95.58	100	100	100
12	12	5	61.33	57.83	55.33	55.08	54.75	94.67	100	99.92	100	100	100
12	12	*Avg.*	40.35	38.57	36.93	36.17	35.92	62.13	71.12	70.93	81.02	85.67	88.07
18	18	1	11	11.17	11.28	11	10.72	15.78	18.61	17.11	23.22	27.94	32.78
18	18	2	22.33	19.72	19.28	18.5	17.5	31.83	36.67	39.22	58.94	72.67	84.72
18	18	3	28.61	27.72	26	24.83	24.22	44.78	54.78	58.11	84.56	98.78	99.83
18	18	4	36.39	33.83	32.44	31	30.83	58.56	70.94	74.33	98.5	100	100
18	18	5	42.67	40.11	38.72	37.56	36.56	70.28	86.5	90.17	100	100	100
18	18	*Avg.*	28.2	26.51	25.54	24.58	23.97	44.24	53.5	55.79	73.04	79.88	83.47
24	24	1	8.25	8.38	8.46	8.04	8.63	11.79	13.38	11.75	16.71	21.08	32.06
24	24	2	16.46	14.96	14.08	14.58	13.75	23.29	29.33	31.04	47.88	63.25	87.5
24	24	3	22.08	19.67	19.88	19.29	19.13	34.46	41.33	49	80.75	98.08	99.94
24	24	4	27.83	26.54	24.71	24.38	23.67	45.04	54.71	72.79	99.33	100	100
24	24	5	33.92	31.21	29.21	28.79	28.58	54.75	69.29	92.58	100	100	100
24	24	*Avg.*	21.71	20.15	19.27	19.02	18.73	33.72	40.79	43.37	64.3	73.78	79.32
30	30	1	6.3	6.93	6.3	6.47	6.83	9.07	10.8	10.3	13.5	17	19.73
30	30	2	13.57	12.1	11.8	11.47	10.93	19.53	22.53	23.57	35.73	48.87	60.5
30	30	3	18.03	16.67	15.6	15.73	15.07	27.8	34.17	36.17	57.87	77.53	92.93
30	30	4	23.1	20.77	19.77	19.3	19.03	35.77	44	47.47	75.2	96.6	100
30	30	5	27.13	24.97	23.67	23.07	22.8	44.3	53.7	58.57	91.87	100	100
30	30	*Avg.*	17.63	16.29	15.43	15.21	14.93	27.29	33.04	35.21	54.83	68	74.63

The AFDE always improves when adding only valid values. For the largest test scenario $P = 30$ and $E = 30$ with four and five valid values, the testing strengths $t = 4$ and $t = 5$ are sufficient to detect an incorrect error-handler almost always. Although, testing with $t = 5$ can be perceived as impractical as it would require the execution of 23369 (`4-1`) and 58468 (`5-1`) test cases (See Table I).

This result is also consistent with the characteristics of valid values when applying the t-factor fault model. The FDE is improved because additional MFICs are created by the additional valid values.

Besides the two favorable cases with (`4-1`) and (`5-1`), $t = 5$ is not sufficient to detect all incorrect error-handlers reliably. For 2 valid values (`2-1`) and $E = 18$, the AFDE is 90.17%. A higher testing strength would be necessary which further increases the time for test input generation and the time for test execution due to the larger the test suite size.

There is also a trend indicating that the AFDE decreases when adding only invalid values. For instance, the average AFDE for $P = 30$ and $E = 6$ decreases from 76.1% for one valid and one invalid value (`1-1`) to 70.9% for five invalid values (`1-5`). However, this trend is not as clear as for valid values. One exception exists for $P = 6$ and $E = 6$ where the average AFDE improves by 0.57 (`1-3`) and 0.1 (`1-5`) percentage points.

But, the trend is also consistent with the characteristics of invalid values when applying the t-factor fault model. The parameter of one invalid value belongs to the infection sub-condition and improves the probability of triggering the fault. But, all other invalid values either deteriorate the probability because their respective parameter belongs to the effective prevention sub-condition or the invalid value has no effect.

When adding valid and invalid values equally (`2-2` and `2-2`), the AFDE always increases in comparison to `1-1`. The AFDE is also always higher when comparing it to test scenario families with more invalid than valid values. But, the AFDE is always lower compared to test scenario families with more valid than invalid values.

This finding is still consistent with the t-factor fault model. Although, it cannot be directly derived from it. The numbers indicate that the additional MFICs introduced by additional valid values have a stronger effect on the AFDE than the prevention sub-conditions increased by additional invalid values.

To summarize the findings, CT triggers all t-factor faults as guaranteed by the t-wise coverage criterion. Furthermore, even more faults are triggered via collateral coverage. Test suites that satisfy higher testing strengths $t \geq 3$ trigger many incorrect error-handlers with higher indices.

Adding valid values to the test suite increases the AFDE. But, the required test suite size increases as well. Conversely, adding invalid values decreases the AFDE. Additional parameters with error-handling have no impact on FDE for existing error-handlers. But, the faults in the additional error-handlers are harder to detect. Therefore, AFDE deteriorates with an increasing number of parameters involved in error-handling.

Overall, the experiment shows that CT can be an effective approach to detect incorrect error-handlers. Although, not all incorrect error-handlers can be detected reliably. Depending on the number of error-handlers and the distribution of valid and invalid values, a high testing strength is necessary which requires the execution of large test suites.

VIII. Threats to Validity

We compared the effectiveness of CT in the presence of error-handling and invalid values. Therefore, test inputs are generated and executed on test scenarios. Publicly available test suites are used to avoid bias in the test input generation.

The used test scenarios are artificial and do not necessarily represent realistic scenarios. In addition, it is possible that we unconsciously designed the test scenarios in a way that their pre-established characteristics are supported. However, the considered characteristics are explicit and all information is available online[1] so that it is comprehensible and repeatable. In addition, the AFDE metric is designed to derive knowledge and to apply it to real-world scenarios.

To prevent falsified results due to accidental fault triggering, the parameters and values of each test suite are randomized and 100 variants of each test suite are combined to a test suite family. All presented numbers are average numbers.

IX. Conclusion

CT is a generally effective approach to black-box test input generation. When considering invalid values to also test for robustness, the input masking effect can prevent faults from being triggered. CRT is an extension to CT for robustness testing that avoids the input masking effect. But, CRT imposes extra costs since it requires additional semantic information in the test model. The implications of input masking on the effectiveness of CT are unclear beyond the general idea. Therefore, it is unclear when CT can be used and when CRT should be used despite the extra costs.

In this paper, we designed and conducted a controlled experiment to measure the effectiveness of CT in different test scenarios. Therefore, we applied the t-factor fault model and discussed characteristics that are specific to error-handling and invalid values. Based on these characteristics, artificial test scenarios are designed and tested using publicly available small test suites.

The results of the experiment show that CT triggers all t-factor faults as guaranteed by the t-wise coverage criterion. Even more faults are triggered via collateral coverage. Test suites that satisfy higher testing strengths $t \geq 3$ trigger many incorrect error-handlers with higher indices.

Valid values increase FDE and AFDE and invalid values decrease them. Additional parameters with error-handling have the highest impact which deteriorates AFDE the most. While $t = 4$ is sufficient in favourable cases with many valid values and few parameters involved in error-handling, not even $t = 5$ is sufficient in unfavourable cases with many invalid values and many parameters involved in error-handling.

Overall, the experiment shows that CT can be effective in the presence of error-handling and invalid values. But in many cases, an advantageous distribution of valid and invalid values, a high testing strength and thus very large test suites are required.

Despite the good performance of CT in many cases, CRT is a promising approach that requires potentially fewer test executions. Although, a direct comparison is necessary to decide if the execution of very large test suites required by CT outweighs the additional modelling effort.

In future work, we will extend the experiment to consider invalid value combinations and configuration-dependent faults as well. As they potentially increase the number of parameters involved in the prevention and infection sub-conditions. Further, we will compare not only the effectiveness but also the efficiency of CT with CRT.

References

[1] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std*, vol. 610.12-1990, 1990.

[2] A. Avizienis, J. Laprie, B. Randell, and C. E. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Sec. Comput.*, vol. 1, no. 1, pp. 11–33, 2004.

[3] M. Young and M. Pezze, *Software Testing and Analysis: Process, Principles and Techniques*. USA: John Wiley & Sons, Inc., 2005.

[4] Y. Li, S. Ying, X. Jia, Y. Xu, L. Zhao, G. Cheng, B. Wang, and J. Xuan, "Eh-recommender: Recommending exception handling strategies based on program context," in *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*, 2018, pp. 104–114.

[5] K. Fögen and H. Lichter, "Combinatorial robustness testing with negative test cases," in *2019 IEEE International Conference on Software Quality, Reliability and Security, QRS 2019, Sofia, Bulgaria, July 22-26, 2019*, 2019, pp. 34–45.

[6] P. Sawadpong, E. B. Allen, and B. J. Williams, "Exception handling defects: An empirical study," in *14th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2012, Omaha, NE, USA, October 25-27, 2012*, 2012, pp. 90–97.

[7] N. Li and J. Offutt, "Test oracle strategies for model-based testing," *IEEE Trans. Software Eng.*, vol. 43, no. 4, pp. 372–395, 2017.

[8] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," *Softw. Test., Verif. Reliab.*, vol. 15, no. 3, pp. 167–199, 2005.

[9] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatiorial design," *IEEE Trans. Software Eng.*, vol. 23, no. 7, pp. 437–444, 1997.

[10] J. Czerwonka, "Pairwise testing in real world: Practical extensions to test case generators," in *24th Pacific Northwest Software Quality Conference*, vol. 200, 2006.

[11] M. Grindal, B. Lindström, A. Offutta, and S. Andler, "An evaluation of combination strategies for test case selection," Department of Computer Science, University of Skövde, Tech. Rep. HS-IDA-TR-03-001, 2003.

[12] Y. Lei and K. Tai, "In-parameter-order: A test generation strategy for pairwise testing," in *3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE '98), 13-14 November 1998, Washington, D.C, USA, Proceedings*, 1998, pp. 254–261.

[13] B. Chen and J. Zhang, "Tuple density: a new metric for combinatorial test suites," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 876–879.

[14] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection," *IEEE Trans. Software Eng.*, vol. 41, no. 9, pp. 901–924, 2015.

[15] P. Wojciak and R. Tzoref-Brill, "System level combinatorial testing in practice - the concurrent maintenance case study," in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, 2014, pp. 103–112.

[16] L. Yu, Y. Lei, R. Kacker, and D. R. Kuhn, "ACTS: A combinatorial test generation tool," in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, 2013, pp. 370–375.

[17] K. Fögen and H. Lichter, "Combinatorial testing with constraints for negative test cases," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, 2018, pp. 328–331.

[18] ——, "Repairing over-constrained models for combinatorial robustness testing," in *2019 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS Companion 2019, Sofia, Bulgaria, July 22-26, 2019*, 2019, pp. 177–184.

[19] ——, "Semi-automatic repair of over-constrained models for combinatorial robustness testing," in *26th Asia-Pacific Software Engineering Conference (APSEC), Putrajaya, Malaysia, Dec 2-5, 2019 (to be published)*, 2019.

[20] ——, "A case study on robustness fault characteristics for combinatorial testing - results and challenges," in *Proceedings of the 6th International Workshop on Quantitative Approaches to Software Quality co-located with 25th Asia-Pacific Software Engineering Conference (APSEC 2018), Nara, Japan, December 4, 2018.* CEUR-WS.org, 2018, pp. 22–29.

[21] H. Wu, N. Changhai, J. Petke, Y. Jia, and M. Harman, "An empirical comparison of combinatorial testing, random testing and adaptive random testing," *IEEE Transactions on Software Engineering*, 2018.

[22] S. R. Dalal and C. L. Mallows, "Factor-covering designs for testing software," *Technometrics*, vol. 40, no. 3, pp. 234–243, 1998.

[23] D. Harel and A. Pnueli, "On the development of reactive systems," in *Logics and Models of Concurrent Systems*, K. R. Apt, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 477–498.

[24] D. R. WALLACE and D. R. KUHN, "Failure modes in medical device software: An analysis of 15 years of recall data," *International Journal of Reliability, Quality and Safety Engineering*, vol. 08, no. 04, pp. 351–371, 2001.

[25] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.*, Dec 2002, pp. 91–95.

[26] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 418–421, 2004.

[27] K. Z. Bell and M. A. Vouk, "On effectiveness of pairwise methodology for testing network-centric software," in *2005 International Conference on Information and Communication Technology*, Dec 2005, pp. 221–235.

[28] D. R. Kuhn and V. Okun, "Pseudo-exhaustive testing for software," in *30th Annual IEEE / NASA Software Engineering Workshop (SEW-30 2006), 25-28 April 2006, Loyola College Graduate Center, Columbia, MD, USA*. IEEE Computer Society, 2006, pp. 153–158.

[29] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Estimating t-way fault profile evolution during testing," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, June 2016, pp. 596–597.

[30] R. N. Kacker, D. R. Kuhn, Y. Lei, and J. F. Lawrence, "Combinatorial testing for software: An adaptation of design of experiments," *Measurement*, vol. 46, no. 9, pp. 3745 – 3752, 2013.

[31] P. Arcaini, A. Gargantini, and M. Radavelli, "Efficient and guaranteed detection of t-way failure-inducing combinations," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2019, Xi'an, China, April 22-23, 2019*, 2019, pp. 200–209.

[32] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and R. D. Kuhn, "Refining the in-parameter-order strategy for constructing covering arrays," *Journal of Research of the National Institute of Standards and Technology*, vol. 113, no. 5, pp. 287–297, 09 2008.

Studying Multifaceted Collaboration of OSS developers and its impact on their bug fixing performance

Amit Kumar
Department of Information Technology
Indian Institute of Information Technology
Allahabad, India
Email: *amitchandramunityagi@gmail.com*

Mahen Gandhi
Area of Computer Science and Engineering
NIIT University
Neemrana, India
Email: *mahenm.gandhi@st.niituniversity.in*

Yugandhar Desai
Area of Computer Science and Engineering
NIIT University
Neemrana, India
Email: *yugandhard.desai@st.niituniversity.in*

Sonali Agarwal
Department of Information Technology
Indian Institute of Information Technology
Allahabad, India
Email: *sonali@iiita.ac.in*

***Abstract*—Developers often collaborate to fix complex bugs, even in open source software systems (OSS) where collaboration largely occurs through discussions in the bug tracker. The implicit Developer Social Networks (DSN) are created as a result of these discussions. Past research has investigated the usefulness of such DSNs in addressing many Software Engineering problems (e.g. Defect Prediction, Evolution of collaboration patterns, etc.). However, the multifaceted nature of DSNs constructed from bug reports data has been ignored in most of the past studies. That is, in most of the past studies, the link among developers exist only if they comment on the same bug report while in reality, the developers may be connected indirectly (e.g. pair of developers are connected even if they comment on two different bug reports which are associated with the same software component). Such unexplored relationships among developers can be used in defining new measures to identify important developers in the OSS system which otherwise is not trivial to do. In this paper, we study this implicit multifaceted nature of collaborations among developers by extending single layer DSN to Multi-layer DSN (MDSN). Our experiments performed on bug data of Eclipse and NetBeans show that structure of DSNs and their evolution at various layers differ significantly and performance of developers in bug fixing process is not only significantly correlated (Pearson correlation coefficient up to 0.74) with their network centrality scores but also vary across various layers of MDSN signifying their usefulness in determining the crucial and important developers in the software systems.**

***Index Terms*—Developer Social Network, Multidimensional Developer Social Network, Multilayered Developer Social Network, Multifaceted Developer Social Network**

I. Introduction

0pt5.5ex plus 1ex minus .2ex4.3ex plus .2ex Issue Tracking Systems are not only used to archive bug reports and the related information but also to help developers to collaborate and have a discussion on issues (bugs or features). Developers typically interact by commenting on bug reports. These interactions form an implicit developer social network (DSN).

Due to the readily available data from issue trackers, researchers have started investigating DSNs to solve software development problems. For example, DSNs have been used to study community structures of software developers and their evolution [1], to categorize bug reports [2], and to help in defect prediction [3].

However, most of these studies explore only one type of links among the developers (e.g. In DSN constructed from bug report data, the developers are connected if they have worked together to fix the same bug report) while they are indirectly connected through various other avenues. For instance, developers who have not commented on the same bug report but have commented on two different bug reports found in the same component of a software product, are indirectly connected. In this paper, we consider many such indirect connections among the developers and build the Multi-layered / Multi-faceted Developer Social Network (**MDSN**). In our Multi-Layered DSN, each layer represents a different DSN which shows the links among developers capturing different types of proximity among them. We believe that a holistic view of these different kinds of proximities among the developers and investigation of Multi-faceted Developer Social Network (MDSN) can elucidate more on the nature of developer collaborations on issue tracking systems.

Towards our goal of investigating MDSN, we first attempt to answer the fundamental question if the structure of DSN at various layers vary significantly from each other. Network Structure of DSN has been characterized by many global social network properties in past studies [4] [5]. We also use global social network properties to characterize and investigate DSN of various layers and hence ask the following research question:

RQ1: How significantly the global network properties of DSN vary across the layers of the MDSN?

Past studies have reported that DSN does not remain invariable and evolve. Studying such evolution of DSN is important as it allows us to comprehend how relationships

among developers evolve. MDSN has many layers of DSN and hence it is important to study and compare the evolution of each DSN. This sheds light on the dynamics of DSNs at each layer. In particular, we pose our second research question as follows:
RQ2: How does the evolution of DSNs differ at each layer? Do some DSNs evolve faster than others?

The first two research questions are posed to investigate if the Multifaceted/Multilayered approach of studying DSN adds some value to the understanding of developer communication structure or not.

However, past studies have used DSN to characterize the traits, performance, and importance of the developers [6] [7] [8] [9]. Node centrality measures in DSN and entropy of developers contributions have been used widely to characterize the importance of developers in the collaboration network of developers [10]. In MDSN, the collaboration happens at various levels and hence it will be interesting to see how the importance of nodes in DSN is associated with their bug fixing performance. To measure the importance of the node(developer), we compute the graph entropy-based measures along with various node centrality measures at each layer of DSN. In particular, we ask our third research question as follows:
RQ3: How significantly various metrics measuring the importance of the developers in DSNs correlate with their bug fixing performance? How significantly these correlations differ across different layers of MDSN?

To answer these research questions, we first construct the Multilayered Developer Social Networks from the bug report data of two popular Java IDE projects-Eclipse and NetBeans. Then we use many global network properties(characterizing the properties of the entire network), node importance based measures and measures to characterize the bug-fixing performance of developers to answer our research questions.

II. Related Work

Leveraging archival data to facilitate ongoing software development is a key tenet in software engineering research. One such data that researchers have started using is the implicit social networks that are created because of developer interactions: when they work on the same file or task or communicate regarding an issue or task. Here we sample a subset of research involving DSNs which are related to our work. For example, Canfora et al. [9] mine data from the mailing lists to identify experienced developers who actively interact with newcomers to identify mentors. Bird et al. [1] on the other hand, have used the DSNs from mailing lists to investigate the social status of OSS participants based on the network structure and the relationship between email and commit activities via these networks. Hong et al. [4] have investigated how a DSN evolves and compared it to the evolution in other social networks like Facebook, Twitter, etc. The network structure properties of DSNs have also been studied to identify the structures that correlate with efficiency in the bug fixing process [5].

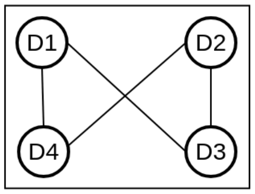


Fig. (1) Single-layered DSN

Zanetti et al. [2] found that centrality of users in a communication network between bug reporters and developers to be indicative of the quality of a bug report. Cataldo and Herbsleb [6] observed that the core developers in the communication structure of the organizations are top contributors. Meneely et al. [3] and Wolf et al. [11] used developer social network for failure prediction. More recently, Wang and Nagappan [12] studied the distribution of collaboration patterns and used them to see the impact of such patterns on the quality of the project from the security point of view.
Our study is similar to the work described above as we also study developer social network. However, instead of using single layer DSN, we study multilayered(Multifaceted DSN). Each layer in our Multifaceted DSN represents different sort of relationship among developers making it a richer framework for depicting more complex proximities among them. Our model of MDSN is inspired by Kazienko et al. [13]. However, to the best of our knowledge, we are first to calibrate and use it in the Software Engineering domain. We also investigate how the positions of developers in networks represented by various layers in proposed Multi-faceted DSN convey about their performance in bug fixing activities. This investigation also adds to the novelty of the proposed work in this paper.

III. Background and Methodology

A. Developer Social Network (DSN) and Multi-layered DSN (MDSN)

Issue Tracking Systems have been used as a communication platform by developers while they work on the bugs reported by users and fellow developers. Developers usually do so by commenting on the bug reports. A typical issue report in ITS of large software ecosystem such as NetBeans or Eclipse has many fields which provides details about the issue e.g. short and long description of the issue, product and component of

TABLE (I) Example Bug Reports

Bug Id	Assigned To	Severity	Priority	Product Id	Component Id	Reporter	Operating System	Commenters
B1	D1	S1	P1	2	3	R1	Linux	D1, D4
B2	D1	S1	P3	1	11	R2	Windows XP	D2, D3
B3	D1	S2	P2	2	3	R3	Linux	D2
B4	D2	S1	P1	2	4	R1	Mac OS X	D2, D4
B5	D3	S3	P3	3	11	R3	Mac OS X	D1, D3

TABLE (II) Dataset

	Duration	#Bug Report's	#Assignees	#Comments	#Commenters	#Reporters	#OS	#Products	#Components	#Versions
Eclipse	2001-2005	27213	759	149340	2767	1911	23	59	325	88
NetBeans	2001-2005	21345	374	134040	1905	1559	16	36	328	21

the given product where the issue is likely to be present, operating system (the product is used/tested with), priority of the issue, severity of the issue, reporter who reported the issue, assignee (whom the issue is assigned to) etc. The discussion on Issue Reporting Systems has been leveraged to construct the Developer Social Network among developers. However, the DSN can be modeled as a single-layered DSN as well as a multilayered DSN. Most of the past studies [4] [5] [8] consider DSN as single-layered, where developers are nodes and edges between the pair of developers exist if they have commented on the same bug report.

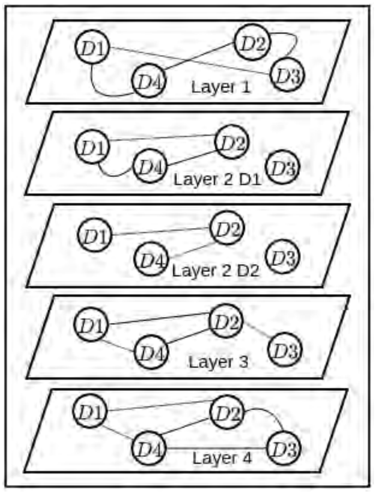


Fig. (2) Multi-layered DSN

We model our DSN as Multi-layered DSN so that multiple facets of the collaboration among developers can be investigated. In our MDSN, each layer represents a different kind of relationship among the developers. In the case of single-layer DSN (DSN considered by past studies), the developers are connected with the edge between them if they have commented on the same bug report. In MDSN, developers are also connected even if they comment on different bugs/issues which are found in the same product, the same component of the product, reported by the same reporter or if the bugs were discovered while software product was used with the same operating system. In total, we have five layers in our MDSN. To illustrate further the difference between single-layered DSN and MDSN, let us consider a toy example data set shown in Table I. There are five bug reports with some (relevant to our study) of their attributes in this table. The single-layered DSN constructed out of this dataset is shown in Figure 1 while MDSN consisting of five layers can be constructed as shown in Figure 2. It can be noted that single-layered DSN is contained in the MDSN as one layer (L1) in it, making it a richer network framework to depict the deeper relationships among the developers. Other layers in MDSN of Figure 2 can be understood easily i.e. L2-D1 represents the network where developers are connected with an edge if they have commented on different bug reports found in the same product. L2-D2, L3, and L4 represent the similar semantics i.e. developers in L2-D2, L3 and L4 are connected with the edge between them if they have commented on the different bug reports found in same product as well as same component, two bug reports reported by same reporter, two bug reports associated with same operating system respectively. It should be noted that we deliberately avoided the trivial links in layers L2-D1, L2-D2, L3, and L4 (replication of links in layers L2-D1, L2-D2, L3, L4 due to L1) by connecting only those developers commenting on different bug reports. We did it to analyze the exclusive nature and power of DSN at each layer. We used DSN at each layer to answer our research questions by exploring the global network properties and various node importance measures of it.

B. Global Network Properties

To investigate the difference in the nature and evolution of various DSNs at every layer, we used similar global network properties as used by Hong et al. [4]. We compared the DSNs at each layer based on the following global network properties:

1) Network density: Network density is defined as the ratio of number of edges present in the network and the maximum possible edges which can exist in the network(excluding self-loops). Higher density of network indicates higher levels of inter-developer communication.

2) Modularity: Modularity of network is important measure as higher value of modularity denotes the higher community structure present in the network. We used the same modularity definition as defined by Newman [14].

$$M = \sum_{i=1}^{n} \left(x_i - y_i^2\right)$$

where, x_i denotes the proportion of the edges between the vertices of the community i while y_i denotes the proportion of the edges that are not part of the community.

3) Average Path Length (APL): It is the average length of shortest paths between each pair of nodes in the network. Shorter APL shows that Developers are well connected to each other in the network and are easily accessible to each other.

4) Average Clustering Coefficient: The Clustering Coefficient in a graph is the degree of clustering by a node with its neighboring nodes. The clustering coefficient of a vertex can be defined as follows in an undirected graph:

$$P_i = \frac{2x_i}{y_i\left(y_i - 1\right)}$$

Here x_i is the number of edges between neighbors of node i and y_i is the number of node i's neighbors. Average

Network Clustering Coefficient is the average of all network nodes clustering coefficients. A significantly higher average clustering coefficient indicates that network follows small word phenomenon [15].

5) Average Degree (AD): The degree of a node in the graph is total number of edges incident on that. Since each edge has two vertices and counts in the degree of both vertices, the average degree of the undirected graph is defined as :

$$AD = 2 \times \frac{|E|}{|V|}$$

C. Node Importance Measures

Past studies have leveraged the position of developers in DSN to investigate their importance in the developer community. In particular, various node based centrality measures in DSN have been used to measure the importance of the developers in DSN. To answer our third research question, we used following node importance measures defined for DSN:

1) Eigenvector Centrality: In graph theory, eigenvector centrality (also called eigencentrality) is a measure of a node's importance in a network. The idea is to assign proportional score values to all network nodes. Let $G(V, E)$ be a graph, consisting of vertices V and edges E. Let A = (a_v, t) be the adjacency matrix, i.e $a_{v,t}$ = 1 if vertex v is linked to vertex t, and $a_{v,t}$ = 0 otherwise. The relative centrality score of vertex v as defined by Phillip Bonacich [16] is:

$$x_v = \frac{1}{\lambda} \sum_{t \epsilon M(v)} x_t = \frac{1}{\lambda} \sum_{t \epsilon G} a_{v,t} x_t$$

where $M(v)$ is a set of the neighbours of v and λ is a constant. Mathematically, this can be written in vector notation as the famous eigenvector equation,

$$Ax = \lambda x$$

The principal eigenvector of the above equation denotes the centrality of all network nodes (here, node is the developer).

2) Betweenness Centrality: The Betweenness Centrality of the graph is determined by the propensity of a single vertex to be more central than any other vertex in the graph. In other words, it measures how often a node appears on shortest paths between nodes in the network. The following is the standard measure given by Freeman [17]:

$$C_B(V) = \sum_{s \neq v \neq t \epsilon V} \frac{\sigma_{st}(V)}{\sigma_{st}}$$

where, σ_{st} is the number of shortest paths from $s \epsilon V$ to $t \epsilon V$.

3) Closeness Centrality: A node's closeness centrality in a connected graph is a measure of centrality in a network, measured as the reciprocal sum of the shortest path length between the node and all other nodes in the graph. Therefore, the more central the node, the closer it is to all the other nodes. The closeness centrality as defined by Sabidussi [18] is as follows:

$$C_C(V) = \frac{1}{\sum_{t \epsilon V} d_G(v, t)}$$

4) Entropy based measure: The entropy of a system measures the randomness or uncertainty in it. The entropy of a graph measures the diversity of edges incident on its nodes. Higher the entropy of the graph, more uniform is the distribution of its edges on its nodes. Dehmer and Mowshowitz [19] provides good survey on graph entropy. Graph entropy can be used to measure the importance of the nodes in the graph. For instance, if entropy of a graph is significantly changed after removing a node from it, node is considered important. Though many definitions are available for graph entropy in literature, we define entropy of a graph as follows: Let $G = (V, E)$ be an undirected graph. The entropy of the graph G, denoted as $H(G)$ is defined as:

$$H(G) = \sum_{i \epsilon V} -p_i \log p_i$$

where, p_i is the degree of node i divided by the sum of the degrees of all nodes in the graph. The graph entropy based importance $H(G_i)$ of node i is defined as:

$$H(G_i) = |E_2 - E_1|$$

where, E_1 is the entropy of the graph G with node i and E_2 is the entropy of the graph G without node i. This helps us in determining how important a node is in the graph. Higher the value of $H(G_i)$, more is the importance of the developer i.

D. Performance of developers in bug fixing process

To answer our third research question, we require two set of measures - Node importance measures as defined in previous sub section and measures to quantify the performance of the developers in bug fixing process. We used following measures to measure the performance of developers in bug fixing process.

1) Average fix time: The Average Fixed Time for an assignee **a** is estimated over a certain period of time using the equation below. To calculate the developer's efficiency in certain time period, we only consider the bugs that are opened and fixed during that time period.

$$AFT = \frac{\sum_{i=1}^{n} t2_{b_i} - t1_{b_i}}{n}$$

where,

b_i = i^{th} Bug in set of bugs assigned to the assignee **a**.

n = Total bugs assigned to the assignee **a**.

t1 = Time when the bug was assigned to the assignee.

t2 = Time when the **FIXED** label was added to the bug report for the first time.

2) Aggregate Priority Points: This metric is used to measure the importance of the developer with respect to the type of bugs he/she fixes. The developer who fixes the bugs with higher priority is considered to be more important.We assign priority points to each developer based on the types of bugs he fixes. First we assign the weightage to each priority type as follows:

TABLE (III) Weightage of Priorities

Priority	P1	P2	P3	P4	P5
Points	5	4	3	2	1

Then we calculate the priority points for each developer. Higher value of priority points for the developer signifies that he fixes the bugs with relatively higher priorities making him more important developer than those with lower priority points.

The equation for estimating the aggregate priority points of developer over a certain period of time is as follows:

$$APP = \frac{\sum_{i=1}^{n} p_{\mathrm{i}} \times c_{\mathrm{p}}}{n}$$

where,

n = Total number of bugs assigned to assignee **a**.

$\mathbf{p_i}$ = Priority of bug.

$\mathbf{c_p}$ = Points allocated to priority *p*.

3) Aggregate Severity Points: This is very similar to the Aggregate Priority Points. The points allocated for each severity are as follows.

TABLE (IV) Weightage of Severities

Severity	trivial	minor	normal	major	critical	blocker
Points	1	2	3	4	5	6

Note that NetBeans and Eclipse allow users to demand new features that are not technically real bugs. Therefore, we do not consider those bug reports where the severity attribute is set for enhancement because this category is reserved for feature requests or improvements to the product. The formula for calculating the aggregate severity points is as follows:

$$ASP = \frac{\sum_{i=1}^{n} s_{\mathrm{i}} \times c_{\mathrm{s}}}{n}$$

where,

n = Total number of bugs assigned to assignee **a**

$\mathbf{s_i}$ = Severity of bug

$\mathbf{c_s}$ = Points allocated to severity *s*

4) Total Components Developer Works Upon: This measure is the total number of modules/components that the assignee has worked on during certain time period. This denotes the diversity in the work profile of the developer.

E. Experimental Set up and Dataset

To carry out our work,we performed our experiments with bug reports of two common open-source software projects- Eclipse and NetBeans. We chose these projects because they are very popular among the community of software engineering, developed around a similar time as OSS projects and have similar functionalities that make them a good choice to test our proposed Multi-layered Developer Social Network. In total, Our dataset has 283380 comments made upon 48258 bug reports between 2001 and 2005. We chose this period as both the projects during this time were in their initial phase making them ideal to study their evolution. The complete details of the dataset are shown in Table II.

In Issue Tracking System like Bugzilla(used by Eclipse and NetBeans), though the issue is assigned to one person i.e. Assignee, it is fixed collaboratively by OSS contributors. The collaboration happens through comments made on the bug report. Hence in answering our research questions, we constructed single-layered and Multi-layered DSN out of the comments of the bug reports in our dataset. However, it should be noted that to answer our RQ3, the node importance measures and performance metrics are calculated and analyzed only for assignees as they are most responsible for fixing the assigned bug/issue. We conjecture that the assignees with good node importance measure in MDSN are good performers. In particular, node importance measures of an assignee in different layers influences her performance differently. For computing various global network-based metrics and node importance based measures we used Gephi Network Analysis tool [20].

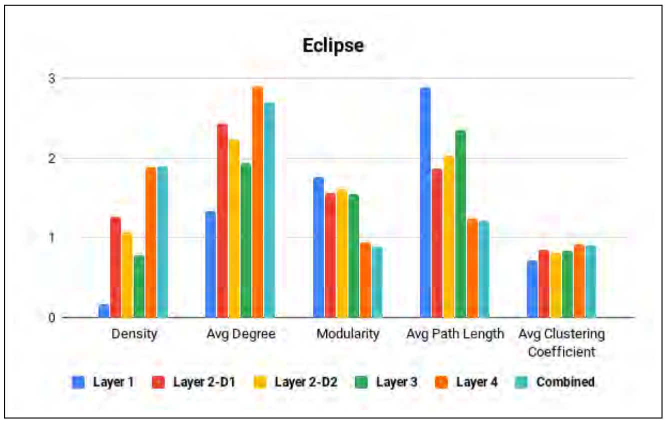
IV. Results and Discussion

In this section, we discuss our results and their implications with respect to our research questions.

RQ1: *How significantly the global network properties of DSN vary across the layers of the MDSN?*

To answer this research question, we computed all the network measures defined in section III-B with the help of Gephi [20] and compared the global network properties of DSNs formed at each layer of MDSN. Past research [4] [5] has also used the similar approach to compare various networks.The difference in network measures across various layers of MDSN signifies the importance of individual layers in MDSN making it more useful framework to study the collaboration patterns among developers. Our results are shown in Figure 3. It can be seen from the figure that while density, average path length, modularity of the DSN vary significantly across different layers of MDSN, the clustering coefficient remains relatively stable across the layers. The modularity of the network describes the community structure of the network and past studies have leveraged modularity of DSN for community detection and discovering team structures in OSS projects [1] [21] [4]. Variation in modularity across different layers suggest that the different community structure and team structure could be discovered using our MDSN approach improving the knowledge about the team structure in OSS maintenance activities. Furthermore, other network measures e.g. network density, average path length etc. have been used to predict the defects in software modules [11] and hence it would be interesting to investigate if the accuracy of defect prediction models could also be improved by incorporating network measures computed based on MDSN. In nutshell, it is clear from our results shown in Figure 3 that network structure of DSNs formed at various layers is significantly different from each other and encourages to leverage MDSN to investigate their usefulness in solving popular research problems e.g. community detection, defect prediction etc.

RQ2: *How does the evolution of DSNs differ at each layer? Do some DSNs evolve faster than others?*



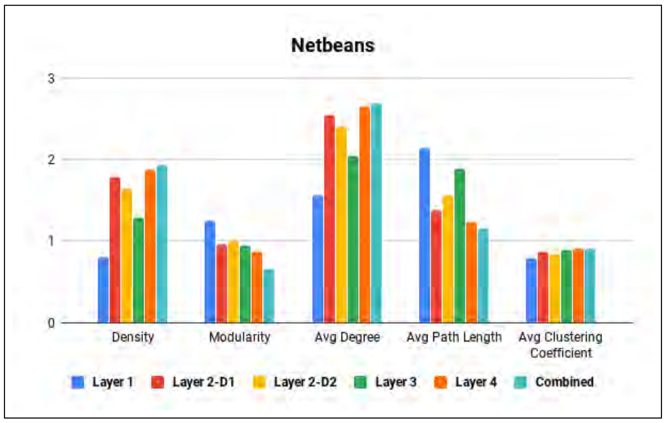


Fig. (3) Variation in Network Structure across various layers of MDSN

To answer this research question, we first split 5 years bug report data of Eclipse and NetBeans (2001-2005) into chunks of 6 months data. This way, we have 10 samples of bug report data for each of the projects. Then we compute the global network properties for each sample to see their evolution over time. Figure 4 shows the evolution of network properties of DSN over time for both Eclipse and NetBeans. It can be seen easily from the figure that the evolution of almost all the network properties at layer-L2-D1(edges between developer nodes if they comment on different bug reports associated with the same product) and L2-D2 (edges between developer nodes if they comment on different bug reports associated with the same product as well as same component) evolve faster than properties at other layers. This out-performance of these layers is observed for both NetBeans as well as Eclipse. Graph-based metrics such as density, modularity and their evolution have been used to study the software evolution and predicting some of its important aspects [22]. Our study extends the past study on DSN based software evolution and suggests that studying the evolution of DSN at various layers of MDSN can provide new insights to the software evolution research. Variation in evolution of DSNs at various layers also suggests that predicting future aspects of some DSNs is more difficult than others. For instance, predicting the developers leaving the project might be much more difficult in layer L2-D1 and L2-D2 in comparison of other layers where evolution is relatively smoother. Overall, the answer to this research question is affirmative based on the results shown in Figure 4 adding the value to our study.

RQ3: *How significantly various metrics measuring the importance of the developers in DSN correlate with their bug fixing performance? How significantly these correlations differ across different layers of MDSN?*

TABLE (V) Correlation Analysis

Layers	Metric	Avg Fixed Time		Total Components		Aggregate Priority Points		Aggregate Severity Points	
		Eclipse	NetBeans	Eclipse	NetBeans	Eclipse	NetBeans	Eclipse	NetBeans
L1	Betweenness Centrality	-0.079	-0.161	0.209**	0.506**	0.742**	0.563**	0.744**	0.578**
	Closeness Centrality	-0.195**	-0.434*	0.181**	0.646**	0.336**	0.636**	0.338**	0.648**
	Eigenvector Centrality	-0.161*	-0.443*	0.315**	0.656**	0.482**	0.656**	0.475**	0.651**
	Entropy Based Measure	-0.162*	-0.222	0.409**	0.005	0.635**	-0.087	0.631**	-0.087
L2 - D1	Betweenness Centrality	-0.059	-0.171	0.282**	0.506**	0.269**	0.358	0.261**	0.396*
	Closeness Centrality	-0.141*	-0.386*	0.092	0.646**	0.162*	0.472*	0.157*	0.489**
	Eigenvector Centrality	-0.096	-0.424*	0.063	0.451*	0.032	0.442*	0.03	0.448*
	Entropy Based Measure	0.162*	0.026	-0.097	0.045	-0.146*	-0.061	-0.139*	-0.029
L2 - D2	Betweenness Centrality	-0.064	-0.119	0.297**	0.48**	0.275**	0.294	0.269**	0.326
	Closeness Centrality	-0.141*	-0.434*	0.106	0.521**	0.175**	0.464*	0.172*	0.473**
	Eigenvector Centrality	-0.092	-0.384*	0.086	0.48**	0.035	0.452*	0.034	0.485**
	Entropy Based Measure	-0.156*	0.022	0.10*	-0.212	0.191**	-0.093	0.185**	-0.092
L3	Betweenness Centrality	-0.11	-0.18	0.223**	0.468*	0.74**	0.642**	0.745**	0.647**
	Closeness Centrality	-0.271**	-0.314	0.274**	0.641**	0.364**	0.651**	0.366**	0.654**
	Eigenvector Centrality	-0.242**	-0.523**	0.311**	0.553**	0.357**	0.526**	0.356**	0.525**
	Entropy Based Measure	-0.232**	-0.212	0.365**	0.08	0.474**	-0.01	0.472**	-0.012
L4	Betweenness Centrality	-0.25**	-0.631**	0.246**	0.524**	0.428**	0.45*	0.436**	0.466*
	Closeness Centrality	-0.347**	-0.656**	0.329**	0.442*	0.285**	0.416*	0.287**	0.422*
	Eigenvector Centrality	-0.393**	-0.788**	0.279**	0.385*	0.204**	0.371*	0.203**	0.373*
	Entropy Based Measure	-0.39**	-0.365	0.301**	0.164	0.251**	0.213	0.252**	0.222
Combined DSN	Betweenness Centrality	-0.263**	-0.656**	0.255**	0.438*	0.32**	0.402*	0.325**	0.407*
	Closeness Centrality	-0.358**	-0.711**	0.277**	0.397*	0.242**	0.379*	0.243**	0.38*
	Eigenvector Centrality	-0.424**	-0.743**	0.247**	0.376*	0.192**	0.36	0.192**	0.36
	Entropy Based Measure	-0.418**	-0.243	0.26**	0.125	0.216**	0.115	0.218**	0.137

$*p < .05, **p < 0.01, ***p < .001$

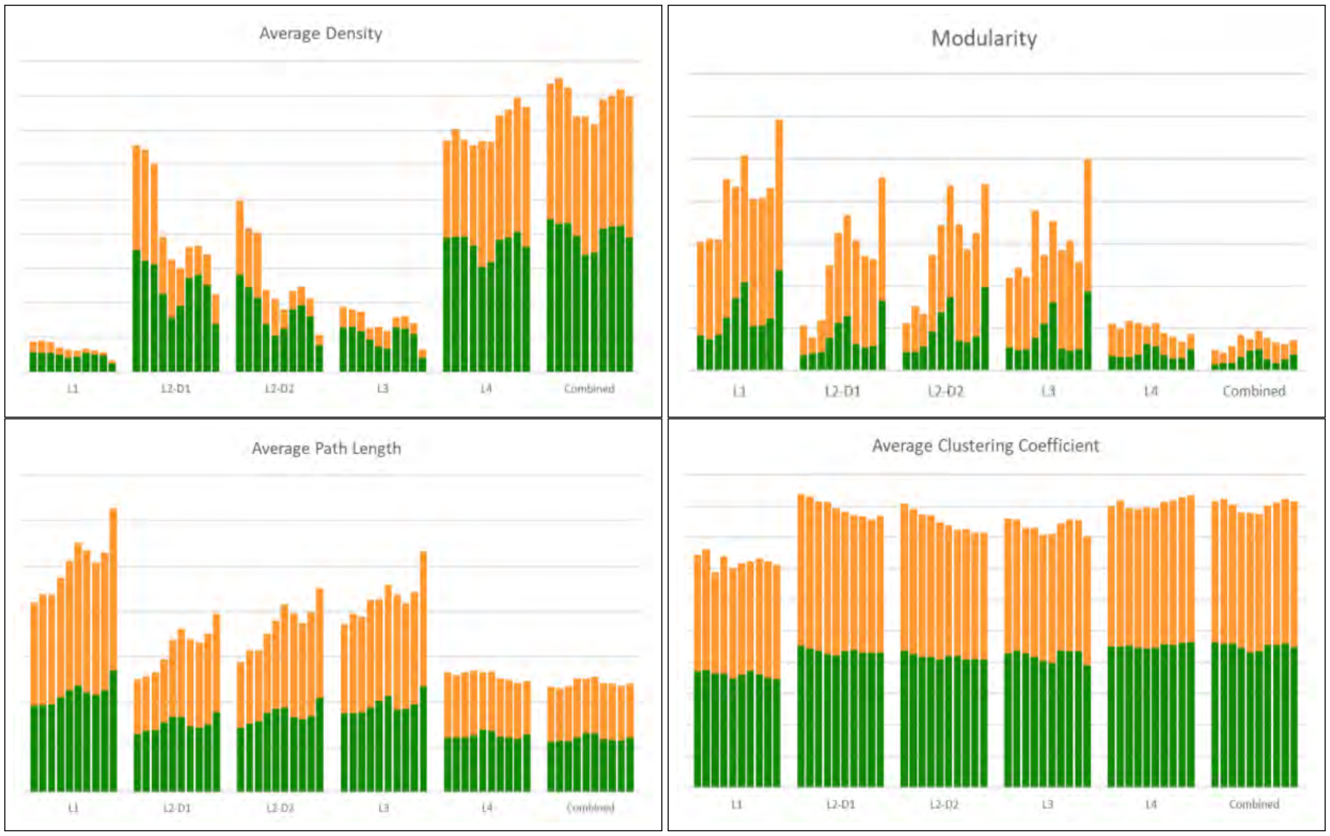


Fig. (4) Layer-wise evolution of global network properties of DSN (🟧 Eclipse, 🟩 NetBeans) from 2001 to 2005.

The main findings of this paper is the answer to this research question. To answer this research question, we first computed various node importance measures for the DSN at each layer as defined in section III-C and the measures to characterize the performance of the developers as defined in section III-D. Then we used the Pearson correlation coefficient to see how effectively and strongly these two sets of measures correlate with each other. We chose Pearson Correlation Coefficient to see the strength of association between two types of measures as it has been found useful by many past studies [23] [24]. Table V shows a summary of our results. The values of correlation coefficients where the p-value is less than 0.05 are specifically highlighted. To see the cumulative effect of node importance measures on the performance of the developers we merged the DSNs of all the layers into one. In this merged integrated DSN, a node exists between the pair of the developers if there exists a link between them in any of the layers - L1, L2-D1, L2-D2, L3, L4. In general, our results are encouraging. For instance, the Eigenvector centrality value of a node in DSN is negatively correlated with the average fix time suggesting that developers who enjoy good eigenvector centrality value fix the issue faster than their peers with lower eigenvector centrality. The value of the correlation coefficient between Average fix time and other node importance measures is also significant. It can also be seen that the correlation coefficient between Average fix time and other node importance measures is maximum for layer-4 out of all the individual layers. This is interesting because past studies show that predicting the fix-time of a bug in OSS is hard. Bhattacharya and Neamtiu [25] reported that many of the features/attributes considered by researchers to build the predictor in predicting the average fix time of the bug are not found relevant. Our results suggest that node importance based measures for assignee can prove to be good features for such predictors. Second, most of the past studies considered the node-based centrality at layer-1 only while our study suggests that similar centrality measures perform better if we leverage MDSN instead of single-layered DSN.

A closer look at Table V shows that node importance measures are also significantly correlated with the total number of components the developer worked upon in fixing the issues. This suggests that developers with high node importance measures gain diverse expertise in fixing the issues making them more crucial/important for the organization. This suggests that our node importance measures can identify the developers with diverse expertise in fixing issues. A significant correlation between node importance measures of developers and aggregate priority points as well as aggregate severity points of the bugs fixed by them shows that node importance measures selected in our study are good measures to identify crucial and important developers (developers who are good to fix the bugs with high priority and high severity).

Interestingly, MDSN approach of investigating the impact of various node importance measures on their bug fixing performance provides new insight as many of the results are counter-intuitive, e.g. best correlation is found for layer-4 (where developers are connected if they have commented on two different bug reports associated with the same operating system). Though the correlation is found significant for both the projects making the finding general enough, the value of correlation coefficients are found to be higher with NetBeans data.

Overall, there are two takeaways from our results-first, significant correlation between various node-based measures and performance of developers suggests that these measures can be used to identify important developers. Second, the measures based on different layers of MDSN are differently correlated with the performance of developers, making the MDSN framework worthy enough to try for identifying the crucial and important developers in OSS.

V. Conclusion and Future Work

In this research, we proposed MDSN to investigate the multifaceted nature of collaboration among the developers while they fix the bugs and collaborate through the Issue Reporting System. There are many takeaways from our research. First, since the structure of networks varies significantly across various layers of MDSN, replicating the past studies on community detection and identifying team formation in OSS on the MDSN framework may provide new insights. Second, Our results show that many node importance measures i.e. node centrality based metrics and graph entropy-based measures have a significant correlation with the performance of the developers in the bug fixing process. Further, such correlations vary significantly across the layers suggesting that MDSN could be more useful to identify important and crucial developers in the developer community of OSS. Though our results are consistent with both the case studies, we selected for our research, there are few threats to its validity. First, comments are not made only by developers on ITS (Issue Tracking Systems) and hence considering all commenters as developers could be a threat to the validity of our results. Second, ITS is not the only platform where developers collaborate. Past research has also used version control data to study collaboration among developers. Hence, performing our study on version control data can complement our study. We plan this in our future work.

References

[1] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 137–143, ACM, 2006.

[2] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "Categorizing bugs with social networks: a case study on four open source software communities," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 1032–1041, IEEE Press, 2013.

[3] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 13–23, ACM, 2008.

[4] Q. Hong, S. Kim, S. C. Cheung, and C. Bird, "Understanding a developer social network and its evolution," in *2011 27th IEEE international conference on software maintenance (ICSM)*, pp. 323–332, IEEE, 2011.

[5] A. Kumar and A. Gupta, "Evolution of developer social network and its impact on bug fixing process," in *Proceedings of the 6th India Software Engineering Conference*, pp. 63–72, ACM, 2013.

[6] M. Cataldo and J. D. Herbsleb, "Communication networks in geographically distributed software development," in *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, pp. 579–588, ACM, 2008.

[7] M. Joblin, S. Apel, and W. Mauerer, "Evolutionary trends of developer coordination: A network approach," *Empirical Software Engineering*, vol. 22, no. 4, pp. 2050–2094, 2017.

[8] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 25–35, IEEE, 2012.

[9] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "Who is going to mentor newcomers in open source projects?," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, p. 44, ACM, 2012.

[10] Q. C. Taylor, J. E. Stevenson, D. P. Delorey, and C. D. Knutson, "Author entropy: A metric for characterization of software authorship patterns," in *Third International Workshop on Public Data about Software Development (WoPDaSD08)*, p. 6, 2008.

[11] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 1–11, IEEE Computer Society, 2009.

[12] S. Wang and N. Nagappan, "Characterizing and understanding software developer networks in security development," *arXiv preprint arXiv:1907.12141*, 2019.

[13] P. Kazienko, K. Musial, and T. Kajdanowicz, "Multidimensional social network in the social recommender system," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 41, no. 4, pp. 746–759, 2011.

[14] M. E. Newman, "Modularity and community structure in networks," *Proceedings of the national academy of sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[15] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *nature*, vol. 393, no. 6684, p. 440, 1998.

[16] P. Bonacich, "Power and centrality: A family of measures," *American Journal of Sociology*, vol. 92, no. 5, pp. 1170–1182, 1987.

[17] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977.

[18] G. Sabidussi, "The centrality index of a graph," *Psychometrika*, vol. 31, pp. 581–603, Dec 1966.

[19] M. Dehmer and A. Mowshowitz, "A history of graph entropy measures," *Information Sciences*, vol. 181, no. 1, pp. 57–78, 2011.

[20] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: an open source software for exploring and manipulating networks," in *Third international AAAI conference on weblogs and social media*, 2009.

[21] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 24–35, ACM, 2008.

[22] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 419–429, IEEE, 2012.

[23] Z. Zhang, W. K. Chan, T. Tse, P. Hu, and X. Wang, "Is non-parametric hypothesis testing model robust for statistical fault localization?," *Information and Software Technology*, vol. 51, no. 11, pp. 1573–1585, 2009.

[24] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pp. 9–9, IEEE, 2007.

[25] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: can we do better?," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp. 207–210, ACM, 2011.

Prioritization of EA Debts Facilitating Portfolio Theory

Yoon Chow Yeong
Universiti Teknologi Petronas
Perak Darul Ridzuan, Malaysia
yeongyoonchow@gmail.com

Simon Hacks
Division of Network and Systems Engineering
KTH Royal Institute of Technology
Stockholm, Sweden
shacks@kth.se

Horst Lichter
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
lichter@swc.rwth-aachen.de

***Abstract*—Implementing an enterprise architecture (EA) project might not always be a success due to uncertainty and unavailability of resources. Hitherto, we have proposed a new metaphor –Enterprise Architecture Debt (EAD)–, which makes bad habits within EAs explicit. We anticipate that the accumulation of EAD will negatively influence EA quality, also expose the business into risk.**
Recognizing the importance of business-IT alignment in enterprise architecture context, this paper proposes an application of portfolio-based thinking and utility theory for EAD prioritization. For proof-of-concept purpose, we develop synthetic data using coarse-grained estimates to demonstrate the application of the proposed portfolio-based approach which helps to determine the optimum selection of EAD to be resolved. The results show that our approach can help EA practitioners and management to reason their EA investment decisions based on the EAD concept, with adjustable enterprises risk tolerance level.

***Index Terms*—Enterprise Architecture Management, Enterprise Architecture Debt (EAD), Portfolio Theory, EA Portfolio Optimization, Utility Theory**

I. Introduction

Technical debt is a metaphor that had been introduced by Cunningham [1]. In the software development industry, technical debt is regarded as a critical issue in terms of the negative consequences such as increased software development cost, low product quality, decreased maintainability, and slowed progress to the long-term success of developing software [2]. Technical debt describes the delayed technical development activities for getting short-term payoffs such as a timely release of a specific software [3]. Seaman et al. [4] described technical debt as a situation in which software developers accept compromises in one dimension to meet an urgent demand in another dimension and eventually resulted in higher costs to restore the health of the system in future.

Furthermore, technical debt is explained as the effect of immature software artifacts, which requires extra effort on software maintenance in the future [5]. The concept of technical debt reflects technical compromises that provide short-term benefit by sacrificing the long-term health of a software system [6]. In view of the original idea of technical debt that focused on the code level in software implementation, the concept had been extended to software architecture, documentation, requirements, and testing [7]. While the technical debt metaphor has further extended to include database design debt, which describes the immature database design decisions [8], the context of technical debt is still limited to the technological aspects.

Over the years, technical debt becomes increasingly important when organizations invest huge amounts of money in IT to stay competitive, effective, and efficient. However, it is vital to align IT and business in order to realize the full benefits and potentials of those IT investments [9]. From there, the concept of Enterprise Architecture (EA) has evolved as a method to facilitate the alignment of IT systems and business strategies within dynamic and complex organizations [10]. Consequently, the huge interest in EA resulted in vast scientific contributions that address a broad thematic spectrum [11], including EA frameworks, EA management, and EA tools. However, there is a lack of insight into the application of the debt concept to include not only the technological aspects addressed by technical debt, but also the business aspects. Adapting the concept of technical debt in the EA domain, hitherto we have proposed a new metaphor “Enterprise Architecture debt (EAD)” to provide a holistic view [12].

In the real world, debt is not necessarily a negative thing to incur, same goes to EA debt to be held in an enterprise. The danger of debt comes into place when there is no proper debt management approach to prioritize, which debt should be repaid as soon as possible. We predict that managing EA debt will be one of the critical success factors of EA implementation and, thus, there is tremendous need to allocate resource effectively to maintain the current level of profitability by properly managing EA debts that exist in an enterprise.

Numerous studies have been dealing with the approaches to prioritize technical debt in the domain of software engineering [3]–[5], [8], [13]–[16], and yet these studies do not address the business aspects as a whole EA. To fill the research gap, this study aims to extend the application of portfolio theory into the concept of EA debt. This will be achieved by focusing on the following research questions:

(*RQ1*) *How can a given set of EA debt items be prioritized based on a portfolio approach?*

The following list of research sub-questions are emerged from the main research question which is mentioned as above:

(*RQ1.1*) *What attributes of EA debt should be contained in a portfolio-based prioritization model?*

(*RQ1.2*) *What are the process steps required to prioritize EA debt items based on a portfolio thinking?*

This study proposes a portfolio-based approach to prioritize EA debt that exists in EA implementation by incorporating the portfolio thinking and utility theory into EA. This proposed approach contributes to the theoretical body of knowledge by providing a fundamental understanding on how EA debt items can be conceptualized and measured for decision-making. It is strongly believed that this approach can measure, manage, and prioritize debt on an enterprise-wide level, which can be valuable to EA stakeholders by avoiding massive interests on EA debt. In light of the novel introduction of the EA debt metaphor, it is foreseen that EA debt decision-making would be a worthwhile subject for future research in the EA field.

The rest of this paper is structured as follows: First, we introduce the facilitated key concepts of modern portfolio theory and utility theory. Second, we present in Section III how we apply the concepts of portfolio theory and utility theory (Section III-A), and depict a process, which guides the prioritization (Section III-B). Next, we demonstrate our approach on a fictitious case study in Section IV and present related work (Section V). Last, we conclude our work in Section VI.

II. Key Concepts

A. Modern Portfolio Theory

In the finance domain, Modern Portfolio Theory (MPT) was originally developed by Markowitz [17]. The goal of this theory is to develop an approach to determine an efficient portfolio with the maximum return at a given level of risk or the minimum risk at a given level of return. Based on this, decisions can be made of which types and amounts of financial assets in a portfolio should be invested or divested [17], [18]. The investments can be stocks, bonds, or other financial products that are characterized by a return at a certain level of risk. The development of MPT was based on the rule that investors should consider expected or anticipated return as a desirable thing, whereas variance of return as an undesirable thing. In MPT, a portfolio is a weighted combination of assets in which each asset's return and variance of the return are used to measure the portfolio performance [17].

The fundamental concept behind MPT emphasizes the importance of evaluating the relationship between price changes in each asset and price changes in every other asset in the portfolio [19]. Each individual financial asset generates different level of return and risk and, thus, the introduction of MPT seeks to minimize the total variance of the investment portfolio's return through the concept of diversification. The diversification allows investors to combine different assets whose returns are not perfectly positively correlated. By wisely deciding on the proportions of various financial assets, the advantage of diversification can be achieved through the portfolio return maximization for a given level of portfolio risk, or the portfolio risk minimization for a given level of portfolio return.

The expected return of a portfolio is expressed by the following equation [17]:

$$E = \sum_{i=1}^{N} w_i \mu_i \tag{1}$$

where E is the portfolio's return, w_i is the weight of asset i in the portfolio, the sum of all weights w has to be 1, and μ_i is the expected return of asset i. On the other hand, the portfolio variance of return is calculated as follows [17]:

$$V(R) = \sum_{i=1}^{N} w_i^2 \delta_i^2 + 2\sum_{i<j}^{N} w_i w_j \rho_{ij} \tag{2}$$

where $V(R)$ is the portfolio's return variance, δ_i is the return variance of individual asset i, and ρ_{ij} is the covariance between the assets i and j. The portfolio's standard deviation, δ_p can then be computed as follows:

$$\delta_p = \sqrt{V(R)} \tag{3}$$

Often, MPT relies on historical variance of financial assets' returns to measure the risk. Unfortunately, projects that involve non-financial assets commonly do not have well-defined historical variance for absolute objective measurement [19]. However, Omisore et al. [19] asserted that this does not eliminate the possibility of applying MPT to non-financial assets because the concept is transferable to a wide range of investments as long as the "risk" is expressed in terms of uncertainty about expectations and possible losses on forecasts. Therefore, in EA debt context, we express risk in term of "chance of interest growth", which brings the risk of an increased amount of required effort to resolve an EA debt item in future phase as well as the negative impacts on the EA value.

B. Utility Function and Risk Aversion

In general, most investors require a greater return as compensation for taking a greater risk [19]. Nevertheless, investors differ in their level of risk tolerance, which means that they are risk averse to varying degrees and eventually leads to different utility functions [20]. The concept of utility function provides a way to select the optimal portfolio that yields the best trade-offs of return and risk, and gives the most satisfaction (utility) to the investors, taking their risk tolerance level into consideration [21]. This can be applied in the enterprise context where each profit-seeking enterprise differs in the amount of risk it is willing to accept at a given level of return.

To address the differences in enterprises' risk tolerance level, our approach proposes to prioritize EA debt portfolio for repayment based on the portfolio theory along with the principle of expected utility maximization. The utility maximization principle states that a rational investor acts to choose an investment that maximizes the expected utility of wealth among a set of feasible investment alternatives [20]. The similar concept can be applied in the context of EA debt in which an enterprise should act to invest in paying off the EA debt portfolio that maximizes the expected utility of resources among a set of existing EA debt portfolios.

Since risk aversion is not an objectively measurably quantity that aims for absolute measure, there is no unique utility function, which comes into place. Carlsson et al. [21] reported that one of the commonly employed utility function is:

$$U(P) = R_p - 0.005 * A * \delta_p^2, \tag{4}$$

where R_p is the expected return of a portfolio, A is an index of the investor's risk aversion coefficient (a higher index indicates a higher level of risk averseness), and δ_p^2 is the variance of the portfolio's expected rate of return, which is the square of standard deviation, δ_p, a measure of portfolio risk. The risk aversion coefficient is meant to be positive for all risk-averse investors whereas a negative index indicates a risk-loving investor [18].

The factor of 0.005 in Equation (4) is a scaling convention and normalizing factor that allows us to express the R_p and δ_p^2 as a percentage value instead of decimals. Adhering the positive affine transformation property of a utility function, we are allowed to scale a utility function by translating it with the addition and/or subtraction of any constant [20]. To further scale down the size of variance for easy interpretation in our study, we reduce the factor of 0.005 in Equation (4) to 0.001.

In this work, we apply the altered Equation (4) to plot risk-indifference curves (also known as utility curves) which allows us to select the attainable and optimum debt portfolio by combining the curves with the risk-return trade-off plots. Having to say that the single point where one of the curves intersects the efficient frontier is the debt portfolio that provides the best combination of risk-return for the risk level that is acceptable for the organization.

III. Applying Modern Portfolio Theory to Enterprise Architecture debt

In the field of Information Systems (IS), it is common to apply theories that originates from a diverse set of disciplines such as psychology, sociology, economics, finance, and computer science for problem-solving at the intersection of people, information technology, and organizations [22]. Based on the definition of EA debt presented by Hacks et al. [12], this work suggests the application of portfolio thinking into EA debt context with the aim of expanding the visibility and understanding of the newly introduced metaphor.

Technically, EA is responsible for translating the organizations strategy into projects that result in the achievement of a target state of the enterprise [10]. The gap between the target (to-be) architecture and the current (as-is) architecture is to be filled by identifying and implementing projects, programs, or initiatives. However, the required resources to achieve the goal of a project, program, or initiative are limited in terms of budget, time and performance specifications [23].

With inadequate resources and other forms of constraints, there are situations where we need to compromise certain principles, goals etc. in the first EA life cycle phase. Any omission of business and IT aspects inevitably leads to an incomplete view of the EA concerned and may result in an EA debt. As described by TOGAF [24], the Architecture Development Cycle (ADM) is a method to develop an EA in a continuous and iterative manner. From there, we anticipate that the existing EA debt somehow needs to be repaid in the future EA life cycle phase and, thus, EA practitioners, IT representatives, and management are accountable to make decisions on which EA debt item needs to be repaid first in order to avoid higher future cost.

It is expected that the accumulation of EA debt in an EA project will significantly affect the quality of an EA such as its maintainability and agility in responding to the rapidly changing business environments. Oppositely, if the EA debt is managed effectively, it is expected to increase the EA value. In other words, EA debt is analogous to a financial asset that generates return at a certain amount of risk. We expect that EA debt prioritization helps to effectively pay off the EA debt and, in turn, the EA can be adapted towards the new business requirements.

In view of the similarity between financial investment and incurring EA debt, we realize a potential of mapping the concept of financial portfolio management to the EA debt context.

A. *How to Measure EA debts*

An organization, which intends to implement EA, consists of numerous EA projects for enterprise transformation, e.g. adding new business processes or retiring applications. This study reasons that EA debt items inevitably exist in each project. EA debt items could be the failure of removing outdated elements in diagrams, a missing implementation standard, undefined business role definitions, outdated technological structure, etc. EA debt prioritization is a decision-making process that involves determining, which EA debt portfolio is optimal to pay off in the current phase. By examining the EA debt items across business and IT layers, an enterprise can gain a better understanding of EA debt embedded along the process of EA implementation. Having EA debt properly assessed and being paid off in line with the long-term business mission and goals, we can ensure that the resources are utilized efficiently. In short, EA debt measurement urges a new way of thinking of technical debt as an integrated part of the organization's business aspects.

In order to apply financial portfolio theory to EA debt, we need to quantify EA debt for measurement. Therefore, we derive a set of operational definitions in line with the financial definitions in portfolio theory that conform to the common assumptions of existing technical debt studies [5], [8]. Based on existing technical debt literature [5], [8], [14], [15], [25], each EA debt item has its associated principal estimate, interest estimate, and expected return. Despite the unit used for technical debt measurement in dollars, hours,

TABLE I
LIST OF MEASUREMENT ATTRIBUTES

Measurement attribute	Operational definition
Principal	The number of hours required to resolve the EA debt in the current EA phase.
Interest amount	The extra hours that will be required in the future phase if the EA debt is not resolved in the current phase.
Interest variance	The likelihood that the interest amount will increase at the point of repayment in future phase.
Expected return	Net benefit of resolving the EA debt in the current phase and hold the debt to the future phase.
Risk aversion coefficient	The degree of enterprise's risk averseness.

people, or work units, these figures are easy to interpret and to handle, because they serve as a common language, allowing trend monitoring as well as historical data comparison [16]. This study opts to use working hours as measurement unit. Each measurement attribute is summarized in Table I.

Referring to Table I, we give an instance to provide an insight into how each measurement unit can be represented. Once an EA debt item is incurred in an EA project, a certain amount of working hours is required to resolve the EA debt, which is denoted by principal (X). For instance, due to an enterprise architect's careless examination, one of the outdated elements in use case diagram of System A was not removed. This incurs an EA debt which requires a principal of 0.2 hours to update the use case diagram immediately, at time 0 (t_0). However, if this particular debt item is held to a future phase (t_{n+1}), this will cause faultiness in clarifying system requirements being developed and eventually affects progress of system development.

As such, this EA debt item carries interest (IA) of 5 working hours, which is the extra hours required in the future to identify and correct the faultiness, which already brings negative impacts. Unfortunately, this interest amount is uncertain in such a way that it is assumed to fluctuate over time depends on the scope, complexity, and impact of the components that the EA debt item associated with at the time of repayment. In this case, if the EA debt item is held until time 4, t_4, the interest amount, is very likely to increase from $IA_{t_1} = 5$ to $IA_{t_4} = 12$, because the EA debt item is now not only bringing negative impacts to the planning phase, but also the development phase. This uncertainty of interest growth rate in the future represents the risk level of an EA debt item, because the EA debt item with high interest growth rate accumulate interest faster, which brings a higher future cost, which can be expressed as interest standard deviation (δ_d). If the interest growth rate of a particular EA debt item is not likely to grow or its growing rate is much lower than other EA debt items, this indicates that the debt payment can be deferred in a way that it carries lower risk.

On the other hand, the expected return of an EA debt item can be understood in which the number of working hours that can be saved up by paying the debt at t_0, which is calculated using the following equation:

$$R_d = \mid X - (X + IA) \mid, \tag{5}$$

where R_d is the individual EA debt item's expected return, X is the principal, and IA is the interest amount of the EA debt item.

To fit in the MPT model, we need to determine the "weight" of each EA debt item (w_d) and "correlations with other EA debt items" for each of the identified EA debt items. We assume that an EA debt portfolio contains all EA debt items in equal proportions, in a way that, $\sum_{w \in W} w_{d_i} = 1$. On the other hand, we adapt the idea of Guo and Seaman [5] to use correlation coefficients to represent the correlation between two debt items, where COR_{ij} expresses the correlation between d_i and d_j. Since an EA debt portfolio is made up with multiple EA debt items across multiple architectural layers, determining the correlations between EA debts requires analysis of all EA entities embedded in EAM activities as well as interoperability between architecture entities and architecture domains. A reliable estimation of correlations could be done through dependency analysis [5]. For simplicity, we consider that the correlation coefficient would be either 1 (two debt items are related to each other) or 0 (two debt items are unrelated to each other). With the value of correlation coefficients, a co-variance matrix can be created by computing:

$$\rho_{ij} = \delta_{d_i} \delta_{d_j} COR_{ij}, \tag{6}$$

where ρ_{ij} is the co-variance between debt items i and j.

With all the measurement attributes required in the MPT model, the expected return, variance, and standard deviation of an EA debt portfolio can be computed using the equations (7), (9), and (10), respectively. The expected return of an EA debt portfolio is the weighted sum of its EA debt items' expected returns:

$$R_P = \sum_{i=1}^{n} w_{d_i}(R_{d_i}), \tag{7}$$

with one constraint as presented in the following equation:

$$\sum_{i=1}^{n} w_{d_i} = 1. \tag{8}$$

On the other hand, the variance of the portfolio's return, which indicates the probabilities that the set of EA debt items will return different levels of benefits, is expressed as:

$$\delta_P^2 = \sum_{i=1}^{n} w_{d_i}^2 \delta_{d_i}^2 + 2 \sum_{i<j}^{n} w_{d_i} w_{d_j} \rho_{ij} \tag{9}$$

The EA debt portfolio's standard deviation, δ_P can then be computed as follows:

$$\delta_P = \sqrt{\delta_P^2} \tag{10}$$

As in practice, it is difficult to estimate and accurately model the exact amount of consequences of an EA debt item. Initially, when an EA debt item associated with each layer is identified, the principal, interest amount, and interest growth rate is estimated subjectively according to the enterprise architects experience. These rough estimations can then be adjusted using historical data that was collected throughout the EA life cycle. The more accurate and detailed the data is, the more reliable the estimation. The following section demonstrates how the proposed approach can be applied to reason about prioritization decisions.

B. Application Process

To apply portfolio theory and utility function to prioritize EA debt items for debt repayment, we need to ensure that the considerations mentioned in Section III-A are taken into account in order to map the portfolio model to EA debt measurement. Following, we propose a series of steps to identify the optimal EA debt portfolio based on portfolio model (cf. Figure 1). On top of that, with the application of utility function, enterprise architects can reason and justify about EA debt repayment decisions based on the enterprises risk tolerance level. The basic steps of our proposed prioritization approach are stated as follows:

1) Identify a project involved in EA implementation to achieve the target (to-be) architecture and let P be its EA debt portfolio.
2) Identify the associated EA debt items, $d_i \mid \forall i \in \{1, 2, ..., n\}$. This step is important, as not only debt items in the IT domain, but also in the business domain are identified.
3) For each EA debt item, d_i, estimate the principal (X_{d_i}), interest amount (IA_{d_i}), interest growth rate/interest standard deviation (δ_{d_i}), weights (w_{d_i}) and the correlations with other debt items ($COR_{d_{i,j}}$).
4) For each EA debt item, d_i, determine the values of portfolio model, which are the expected return (R_{d_i}) and the covariance matrix (ρ_{ij}) using Equation (5) and Equation (6), respectively.
5) Run the portfolio model on the available data to determine the expected return (R_P), variance (δ_P^2) and standard deviation (δ_P) of the EA debt portfolio.
6) Repeat steps 1-5 for all EA projects.
7) Identify the efficient EA debt portfolios. The efficient debt portfolios are the ones that lie on the efficient frontier and give the best return-risk trade-off if the debt is repaid at the current EA phase.
8) Determine the enterprise's risk aversion coefficient. For simplicity, this study ranges risk aversion coefficients from 1.0 to 5.0, with the lower number representing higher tolerance to risk.
9) Apply the utility function (Equation (4)) to calculate the risk-indifference curves.
10) Identify and prioritize the optimum portfolio where the utility curve intersects at the efficient frontier.

IV. Case Study

For proof-of-concept purpose, we applied a synthetic case study and artificial data was generated accordingly based on the proposed steps described in Section III-B. Coarse-grained estimates of EA debt items' properties have been made and we acknowledge that it is sufficient for measuring the EA debt items for preliminary prioritization decision-making. Estimates that are more detailed can be made when more real-world information is available upon which to base the estimates.

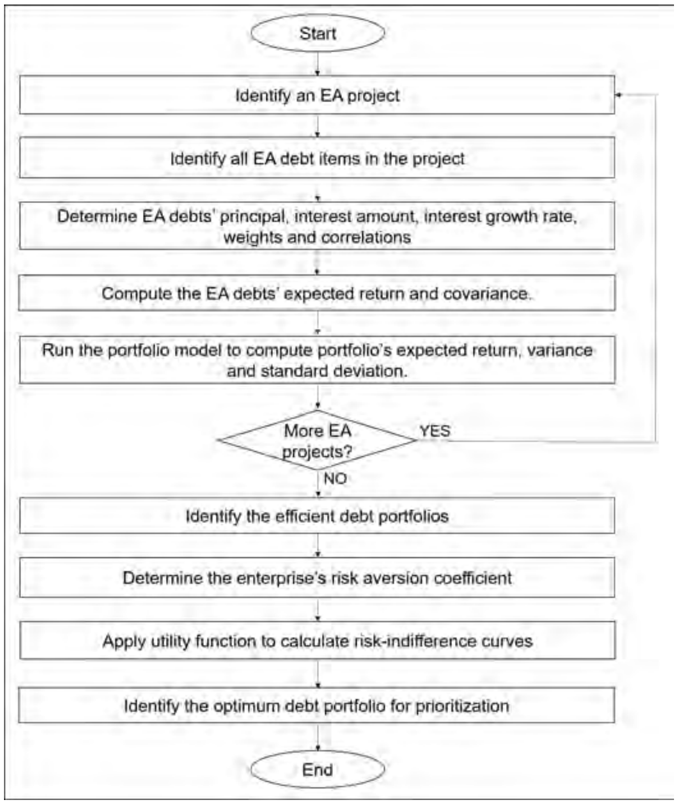

Start
Identify an EA project
Identify all EA debt items in the project
Determine EA debts' principal, interest amount, interest growth rate, weights and correlations
Compute the EA debts' expected return and covariance.
Run the portfolio model to compute portfolio's expected return, variance and standard deviation.
More EA projects?
YES
NO
Identify the efficient debt portfolios
Determine the enterprise's risk aversion coefficient
Apply utility function to calculate risk-indifference curves
Identify the optimum debt portfolio for prioritization
End

Fig. 1. Proposed Process Steps

We considered that a Company ABC can choose from five projects to support the enterprise transition from a current EA to a target EA in order to improve business-IT alignment. Along the EA implementation life cycle, various types of EA debt items incurred in each EA project.

To provide a better understanding of our proposed approach, the following data demonstrates the application of portfolio theory and utility function in the context of EA debt prioritization to show how an optimal EA debt portfolio can be identified and prioritized for decision-making.

Step 1: Identify an EA project: Project A (also known as EA debt portfolio A).

Step 2: Identify the associated EA debt items across the four architectural layers. See Table II.

Step 3: Estimate the principal, interest amount and interest growth rate of each EA debt item. See Table III.

Step 4: Compute the expected return and covariance matrix for each EA debt item. See Table IV.

Step 5: Run the portfolio model to compute the expected return, variance, and standard deviation of the EA debt portfolio. See Table V first row.

Step 6: Identify other EA projects and repeat steps 2-5.

TABLE II
LIST OF EADS IN DEBT PORTFOLIO A

PROJECT A		
ID	**EAD domain**	**EAD description**
A1	Business Architecture	Fail to remove the outdated elements in a use case diagram
A2	Application Architecture	Fail to document interface descriptions
A3	Data Architecture	Lack of data model for an application
A4	Technology Architecture	Missing implementation standard

TABLE III
PRINCIPAL, INTEREST AMOUNT AND INTEREST GROWTH RATE ESTIMATES OF DEBT PORTFOLIO A

PROJECT A			
ID	**Principal**	**Interest amount**	**Interest growth rate**
A1	20 mins	8 mins	80 %
A2	15 mins	10 mins	70 %
A3	23 mins	15 mins	60 %
A4	8 mins	5 mins	50 %

TABLE IV
PORTFOLIO MODEL VALUES OF DEBT PORTFOLIO A

PROJECT A							
ID	**Weight**	**Expected return**	**Standard deviation**	**Covariance matrix**			
				A1	A2	A3	A4
A1	0.25	8 mins	0.8	0.64	0.56	0.48	0.4
A2	0.25	10 mins	0.7	0.56	0.49	0.42	0.35
A3	0.25	15 mins	0.6	0.48	0.42	0.36	0.3
A4	0.25	5 mins	0.5	0.4	0.35	0.3	0.25

TABLE V
RISK AND EXPECTED RETURN OF 5 DEBT PORTFOLIOS

Project	Portfolio risk	Portfolio expected return
A	0.65	9.5
B	0.467	9.333
C	0.4	12.75
D	0.567	15.67
E	0.2	8.333

Table V displays the computed portfolio's expected return and risk of five projects.

Step 7: Identify the efficient EA debt portfolios. As shown in Figure 2, debt portfolio A and B are inefficient portfolio because other portfolios can offer higher return at the similar level of risk or a lower risk at the similar level of return.

Step 8: Define the enterprise's risk aversion coefficient. Company ABC has a risk aversion coefficient value of 2.

Step 9: For visualization, calculate and plot the risk-indifference curves. See Figure 3 for exemplary curves for the utility values of 4,6,8, and 10.

Step 10: Identify the optimum portfolio for prioritization by solving Eq. 4 for every portfolio. The risk-return scatter plot in Figure 4 indicates that EA debt portfolio C is the optimum portfolio that provides best risk-return trade-offs and maximum satisfaction, as it (almost) based on the utility curve of 10.

Our synthetic case study shows that our approach is applicable in general. However, further research is necessary to enable enterprises to apply our approach in practice. Especially, steps 3 and 4 might be extremely challenging, due to missing experience in the field. To tackle step 3, we suggest collecting and documenting possible EA debt items and provide them as catalogs to the community. These catalogs can help to identify possible debt items and serve as a discussion basis to get a deeper understanding of the domain.

Step 4 requires the determination of the measurement attributes, which are needed to calculate the optimal portfolio. However, those attributes usually will be not obvious as for financial assets. Therefore, future research should elaborate on methods that enable practitioners to assess this attributes in an easy manner.

V. RELATED WORK

Despite the vast attention have been paid to technical debt, to our best knowledge, there is no existing approach to prioritize EA debt items as this metaphor is recently proposed by us [12]. Therefore, existing prioritization approaches have been studied in the context of technical debt.

Technical debt management (TDM) is composed of a sequence set of activities to prevent technical debt from being incurred or manage existing technical debt to maintain it under a desirable level [6]. TDM activities include TD identification, TD measurement, TD prioritization, TD prevention, TD monitoring, TD repayment, TD documentation, and TD communication [6]. Technical debt prioritization is considered as one of the TDM activities in which the identified technical debt items are ranked based on predefined rules to decide either immediate repayment or deferred repayment on the debt items [6]. Existing studies have discussed four decision approaches to deal with technical debt prioritization for complex decision-making: Cost-benefit analysis [3], Remediation cost analysis [16], Real Options [13] and Portfolio theory [4], [5]. Meanwhile, the systematic literature review on the financial aspect of managing technical debt conducted by Ampatzoglou et al. [25] concluded that the three most popular financial approaches are cost/benefit analysis, real-options analysis, and portfolio management.

Simple cost-benefit analysis: A simple cost-benefit approach was proposed to prioritize technical debt in terms of which classes should be re-factored first [3]. Each technical debt item consists of the estimations of three metrics: principal, interest probability, and interest amount. This approach prioritize code level debts based on the impact of the God classes on the software maintainability and correctness.

Remediation cost analysis: Moreover, the SQALE (Software Quality Assessment Based on Life-cycle Expectations) method with Sonar tool was proposed to analyze technical debt that associated with an application source code [16]. The authors proposed the synthesis of SQALE Quality and SQALE

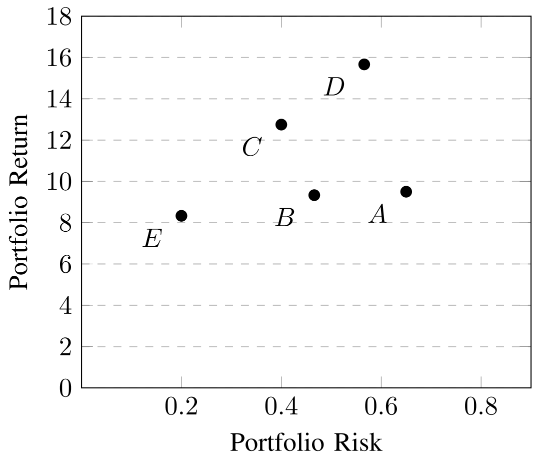


Fig. 2. Risk-return Trade-off of Five Debt Portfolios

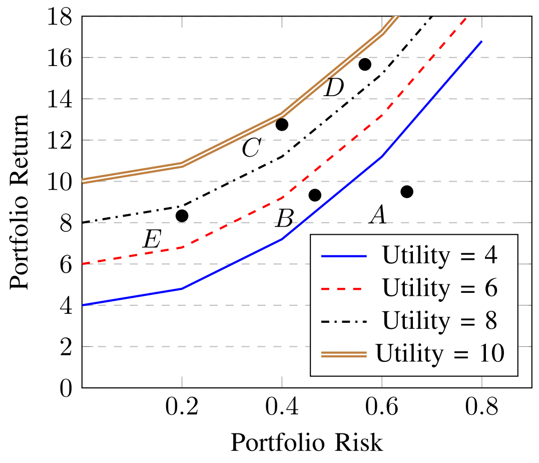


Fig. 4. The Optimal Debt Portfolio

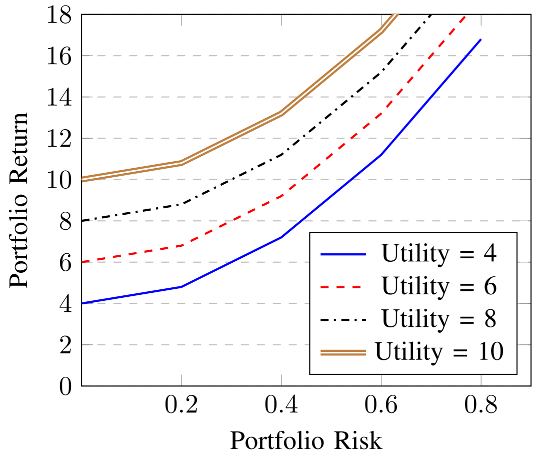


Fig. 3. Risk-indifference curves when risk aversion coefficient=2

Analysis model to measure technical debt in terms of the distance between the codes current quality state and its target state to indicate the quality of an application. On top of that, remediation index was used to represent the remediation cost of corrective actions required to resolve the non-compliance associated with each component of the applications software code.

Real-options approach: Another existing approach is incorporating real options thinking into the valuation of technical debt. Technically, the concept of a real option is about a right to make a future decision without any obligation depending on the way uncertainty is resolved. In other words, purchasing the real option is analogous to investing in paying off technical debt that facilitates future software changes. The real options theory was applied to effectively deal with unpredictable changes in system requirement engineering, time period, and development cost [13]. The proposed approach considers the risk associated with technical debt decisions to manage the value of an organization's strategic flexibility.

Portfolio theory: Furthermore, a portfolio approach was proposed to assist in decision-making in which technical debt items should be repaid and which one should be held for technical debt management [4], [5]. The measurement units embedded in portfolio model, such as expected return, return variance, and return standard deviation are mapped to the context of technical debt management. Each technical debt item was viewed as an asset and the application of portfolio mathematical formulation into a portfolio of debt items helps software developers to decide which technical debt items should be repaid first in order to minimize the future maintenance cost. Also, the portfolio theory was integrated into goal-obstacle method to specifically deal with requirements compliance debt [15].

Our approach differs from those works in two aspects. First, we broaden the scope of technical debt to the entire organization and propose a mapping of EA debt properties to portfolio theory properties. Second, the existing literature on applying portfolio theory to technical debt lacks an explicit description of its application, while we do.

Benchmarking and portfolio matrix: Instead of prioritizing technical debt in general, Plösch et al. [14] focus particularly on design debt, which is incurred due to the violations and non-conformance of design principles on source code level. The authors developed a tool called MUSE in which a portfolio matrix is used to prioritize the identified violations and to communicate design debt. In the proposed approach, a benchmarking-oriented measurement is applied to derive a quality index and categorize each design best practice into Q0- to Q5-area based on the number of identified design best practice violations.

Based on the analysis of existing literature, it is found that the limitation of the aforementioned approaches is that the relative importance of business impacts or operations are not taken into account. Therefore, the concept of linking EA debt to enterprise architecture and applying portfolio thinking is novel. While Stochel et al. [26] suggested to regard each distinct type of technical debt such as process debt as a debt portfolio, we suggest to map each EA project as a debt portfolio.

VI. Conclusion

Implementing EA is essential to enhance the business-IT alignment in a holistic manner. However, academia, software developers, and organizations have been focusing on technical debt, which deals with the quality issues on code, application, and system level. Considering the importance of EA in creating value to organizations, this work has explored a method to identify the optimal set of EA debt items, which should be repaid next. Therefore, we have elaborated on the necessary attributes of EA debts $(RQ1.1)$ and on the necessary process steps $(RQ1.2)$. To tackle $(RQ1.1)$, we have defined a mapping from the EA debt domain to the used terminology in portfolio optimization (see table I). This mapping is used as input for the process (see figure 1) to prioritize the EA debts that answers $(RQ1.2)$.

One of the limitations is that the portfolio-based EA debt model is developed based upon a high-level approach. This means any details, such as EA debt estimation tools and methods, are outside of the research scope. Therefore, estimation guidelines should be developed based on the professional experienced EA practitioners to provide the reference for estimating the debt principal and interest value of each assessed EA debt item.

In the meantime, coarse-grained estimations of EA debt measurement units have been made and we acknowledge that it is sufficient for prioritizing the EA debt items for preliminary decision-making. In real-world practice, EA practitioners are encouraged to substitute estimations based on historical measurements of extra costs required in EA debt repayment. More detailed planning can be made when more historical information is available upon which to facilitate the estimations.

Future research within this domain is two-fold. First, it is necessary to provide catalogs of EA debt items to enable practitioners to identify them in their EA. Those catalogs need to be validated and expanded. Further, effort should be invested to develop methods, which enable the practitioners to assess the measurement attributes that are needed to compute the optimal portfolio. Second, further means to prioritize technical debts need to be transferred to the domain of EA debts. Then, the different means need to be compared concerning their efficiency to determine the most efficient one.

References

[1] W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[2] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013.

[3] N. Zazworka, C. Seaman, and F. Shull, "Prioritizing Design Debt Investment Opportunities," *Proceeding of the 2nd working on Managing technical debt - MTD '11*, pp. 39–42, 2011.

[4] C. Seaman, Y. Guo, N. Zazworka, F. Shull, C. Izurieta, Y. Cai, and A. Vetrò, "Using technical debt data in decision making: Potential decision approaches," in *3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings*, 2012, pp. 45–48.

[5] Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *Proceeding of the 2nd working on Managing technical debt - MTD '11*, 2011, p. 31. [Online]. Available: https://resources.sei.cmu.edu/asset_files/Presentation/2011_017_001_516999.pdfhttp://portal.acm.org/citation.cfm?doid=1985362.1985370

[6] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015.

[7] N. Brown, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, N. Zazworka, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, and R. Nord, "Managing technical debt in software-reliant systems," *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10*, no. January, p. 47, 2010.

[8] M. Albarak and R. Bahsoon, "Prioritizing technical debt in database normalization using portfolio theory and data quality metrics," in *Proceedings of the 2018 International Conference on Technical Debt - TechDebt '18*, 2018, pp. 31–40.

[9] D. H. Olsen, "Enterprise Architecture management challenges in the Norwegian health sector," *Procedia Computer Science*, vol. 121, pp. 637–645, 2017.

[10] J. A. Zachman, "A Framework for Information Systems Architecture," *IBM Systems Journal*, vol. 26, pp. 276–292, 1987.

[11] F. Gampfer, A. Jürgens, M. Müller, and R. Buchkremer, "Past, current and future trends in enterprise architectureA view beyond the horizon," *Computers in Industry*, vol. 100, pp. 70–84, 9 2018.

[12] S. Hacks, H. Höfert, J. Salentin, Y.-C. Yeong, and H. Lichter, "Towards the Definition of Enterprise Architecture Debts," *arXiv e-prints*, 2019. [Online]. Available: https://arxiv.org/pdf/1907.00677.pdf

[13] Z. S. H. Abad and G. Ruhe, "Using real options to manage Technical Debt in Requirements Engineering," *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, no. January, pp. 230–235, 2015.

[14] R. Plösch, J. Bräuer, M. Saft, and C. Körner, "Design debt prioritization," in *Proceedings of the 2018 International Conference on Technical Debt - TechDebt '18*, vol. 10, no. 18. ACM, 2018, pp. 95–104.

[15] B. Ojameruaye and R. Bahsoon, "Systematic Elaboration of Compliance Requirements Using Compliance Debt and Portfolio Theory," in *20th International Working Conference, REFSQ, Essen, Germany*, 2014, pp. 152–167.

[16] J.-L. Letouzey, "The SQALE Method for Managing Technical Debt," in *Proceedings of the 3rd International Workshop on Managing Technical Debt (MTD12), IEEE, Zurich, Switzerland, 2012*, 2012, pp. 31–36.

[17] H. Markowitz, "Portfolio Selection," *The Journal of Finance*, vol. 7, no. 1, pp. 77–91, 1952.

[18] M. Engels, "Portfolio Optimization: Beyond Markowitz," Ph.D. dissertation, Universiteit Leiden, 2004.

[19] I. Omisore, M. Yusuf, and N. I. Christopher, "The modern portfolio theory as an investment decision tool," *Journal of Accounting and Taxation*, vol. 4, no. 2, pp. 19–28, 2012.

[20] J. Norstad, "An Introduction to Utility Theory," 2011. [Online]. Available: http://sriharisheshadri.com/docs/how-much-would-you-bet/norstad_utility.pdf

[21] C. Carlsson, R. Fullér, and P. Majlender, "A possibilistic approach to selecting portfolios with highest utility score," Turku Centre for Computer Science, Tech. Rep., 2002.

[22] S. Lim, T. J. V. Saldanha, S. Malladi, and N. P. Melville, "Theories Used in Information System Research : Insights from Complex Network Analysis," *Journal of Information Technology Theory and Application*, vol. 14, no. 2, pp. 5–46, 2013.

[23] I. Attarzadeh and S. H. Ow, "Project Management Practices: The Criteria for Success or Failure," *Communications of the IBIMA*, vol. 1, no. 28, pp. 234–241, 2008.

[24] The Open Group, "TOGAF." [Online]. Available: http://www.opengroup.org/togaf

[25] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Information and Software Technology*, vol. 64, pp. 52–73, 2015.

[26] M. G. Stochel, M. R. Wawrowski, and M. Rabiej, "Value-Based Technical Debt Model and Its Application," *ICSEA 2012, The Seventh International Conference on Software Engineering Advances*, no. c, pp. 205–212, 2012.