QuASoQ 2018

6th International Workshop on
Quantitative Approaches to Software Quality

co-located with APSEC 2018
Nara, Japan, December 4th , 2018

Editors:

Horst Lichter, RWTH Aachen University, Germany
Thanwadee Sunetnanta, Mahidol University, Thailand
Toni Anwar, University Petronas, Malaysia
Taratip Suwannasart, Chulalongkorn Univiversity, Thailand

Table of Contents

Workshop Organization

- Horst Lichter (Chair), RWTH Aachen University, Germany
- Toni Anwar (Co-Chair), UTP Seri Iskander, Malaysia
- Thanwadee Sunetnanta (Co-Chair), Mahidol University, Thailand
- Taratip Suwannasart (Co-Chair), Chulalongkorn Univiversity, Thailand
- Matthias Vianden, Aspera GmbH, Aachen, Germany
- Wan M.N. Wan Kadir, UTM Johor Bahru, Malaysia
- Maria Spichkova, RMIT University, Melbourne, Australia
- Tachanun Kangwantrakool, ISEM, Thailand
- Jinhua Li, Qingdao University, China
- Apinporn Methawachananont, NECTEC, Thailand
- Nasir Mehmood Minhas, BTH Karlskrona, Sweden
- Chayakorn Piyabunditkul, NSTDA, Thailand
- Sansiri Tanachutiwat, Thai German Graduate School of Engineering, TGGS, Thailand
- Hironori Washizaki, Waseda University, Japan
- Hongyu Zhang, University of Newcastle, Australia
- Minxue Pan, Nanjing University, China

6th International Workshop on Quantitative Approaches to Software Quality

Key Note Speaker

Hongyu Zhang
University of Newcastle,
Australia

hongyujohn@gmail.com

Intelligent Fault Diagnosis and Prediction through Data Analytics

Abstract

Although a range of quality assurance measures have been taken, in reality released software and service systems could still contain faults and fail in operation. In the era of big data and artificial intelligence, we aim towards intelligent, data-driven fault diagnosis and prediction. During the development and maintenance of software and services, a vast amount of data is generated. These data include operation logs, historical failures, metrics, etc. Various machine learning and data analytics techniques can be utilized to mine these data to predict failures, prioritize testing resources, and automate fault diagnosis. As a result, software/service reliability and availability could be improved. In this talk, I will briefly introduce some of my recent work on data-driven fault diagnosis and prediction.

Cross-Sub-Project Just-in-Time Defect Prediction on Multi-Repo Projects

Yeongjun Cho, Jung-Hyun Kwon, In-Young Ko
School of Computing
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
{yj_cho, junghyun.kwon, iko}@kaist.ac.kr

***Abstract*—Just-in-time (JIT) defect prediction, which predicts defect-inducing code changes, can provide faster and more precise feedback to developers than traditional module-level defect prediction methods. We find that large-scale projects such as Google Android and Apache Maven divide their projects into multiple sub-projects, in which relevant source code is managed separately in different repositories. Although sub-projects tend to suffer from a lack of the historical data required to build a defect prediction model, the feasibility of applying cross-sub-project JIT defect prediction has not yet been studied. A cross-sub-project model to predict bug-inducing commits in the target sub-project could be built with data from all other sub-projects within the project of the target sub-project, or data from the sub-projects of other projects, as traditional project-level JIT defect prediction methods. Alternatively, we can rank sub-projects and select high-ranked sub-projects within the project to build a filtered-within-project model. In this work, we define a sub-project similarity measure based on the number of developers who have contributed to both sub-projects to rank sub-projects. We extract the commit data from 232 sub-projects across five different projects and evaluate the cost effectiveness of various cross-sub-project JIT defect prediction models. Based on the results of the experiments, we conclude that 1) cross-sub-project JIT defect prediction generally has better cost effectiveness than within-sub-project JIT defect prediction, especially when the sub-projects from the same project are used as training data; 2) in filtered-within-project JIT defect-prediction models, the developer similarity-based ranking can achieve higher cost effectiveness than the other ranking methods; and 3) although a developer similarity-based filtered-within-project model achieves lower cost effectiveness than a within-project model in general, we find that there is room for further improvement to the filtered-within-project model that may outperform the within-project model.**

***Index Terms*—Just-in-time prediction, defect prediction, sub-project**

I. Introduction

The quality assurance (QA) process on large-scale software usually requires a large amount of computing and human resources and time, which may not be affordable for organizations with limited testing resources. Because of the resource limits, an organization could not investigate all modules—files or packages—in a project but still needs to prioritize the inspection of the modules. If they fail to prioritize modules which have any defects inside and assign limited testing resources to others, it will result in a higher chance of defects in the released software. Defect prediction helps developers identify modules that are likely to have defects and provide a list of the modules that need to be treated first to efficiently assign limited resources [1]. To predict the likelihood of defects in each module, most of the defect prediction techniques provide a prediction model that is built based on various metrics, such as complexity [2] and change-history measures [3].

Although a defect prediction technique is helpful for developers to narrow down the modules to inspect, module inspection usually requires many resources because it is difficult for developers to locate a defect inside a suspicious module. To overcome the drawback of the file-level defect prediction technique, recent studies introduced just-in-time (JIT) defect prediction models that output a prediction of whether a *code change* induces defects, rather than a *file*. Because code change level is more fine-grained than the file level, predicting defect-inducing code changes is known to be effective to provide the precise feedback to developers. Moreover, JIT defect prediction technique can provide faster feedback as soon as a change is made into the source code repository [4]–[6].

In general, building a defect prediction model usually requires a large amount of historical data from a project; therefore, it is difficult to build a model for projects that have just started or for legacy systems in which past change history data are not available. A cross-project defect prediction model can be built by utilizing the history data of other projects to predict defects in a project that lacks data. Cross-project defect prediction techniques have been actively studied [7], [8] and existing studies state that when building a cross-project defect prediction model, choosing a set of training data that are closely related to the target project is essential to ensure the performance of defect prediction [8].

We found that many large-scale projects, such as Apache Maven, Google Android, and Samsung Tizen, divide their projects into multiple repositories, called sub-projects. Each sub-project contains files that are relevant to the sub-project's main concerns—like the project core, independent artifacts, and plug-ins—and those files are managed in a separate source code repository.

Each sub-project generally has a lesser number of commits within its repository compared to other monolithic repositories, where all files and commits are managed within a single repository. For instance, Google Android in 2012 had 275

sub-projects and 183 sub-projects that contain less than 100 commits [9]. In addition, there might be changes in the design and architecture of the project, which consequently deprecates some sub-projects and introduces new sub-projects to the project. In the case of Google Android, there were 275 sub-projects in 2012 [9] and the number of sub-projects grew by over one thousand at 2018[1].

Therefore, applying JIT defect prediction to sub-projects could be problematic because it is well known that building a prediction model based on a small amount of data may increase the risk of overfitting and make the prediction less robust to outliers [10]. This motivated us to investigate the feasibility of applying cross-sub-project JIT defect prediction to multi-repo projects.

As Kamei at el. [11] studied at the project level, it might be enough to use all available sub-projects to build a cross-sub-project JIT defect-prediction model without selecting only similar sub-projects to build a model. However, as the sub-project level change history is more fine-grained than that of the project level, we may achieve higher cost effectiveness by filtering out irrelevant data based on fine-grained information about the data. For instance, we find that developers of a target sub-project usually make contributions to multiple sub-projects rather than to a single sub-project of the target project. This inspired us to develop a new similarity metric that measures the similarity between two sub-projects based on the number of authors (developers) who made commits to both sub-projects. Because different developers usually have different defect patterns [12], we expect that using the JIT defect prediction model built from sub-project repositories whose contributors are similar to those of the target project could show better prediction performance than a prediction model that is built from all available repositories.

In this paper, we study the ways of transferring the JIT defect-prediction models to the models that are appropriate for multi-repo projects in terms of cost effectiveness. We establish two research questions (RQs) to check the effectiveness of using the cross-sub-project JIT defect prediction method:

RQ 1 Is the cross-sub-project model trained by sub-projects from the same project more cost effective than that trained by sub-projects from other projects?

To build a cross-sub-project JIT defect prediction model, training data from other sub-projects are required. As training data are among the most important factors to improve a model's cost effectiveness, choosing proper sources of training data is the first concern to build cross-sub-project models. Before selecting sub-projects based on the similarity against the target sub-project, we want to check the cost effectiveness of the models built with sub-projects from the same projects and those built with sub-projects from other projects. If the models built with the sub-projects from the same project perform better than the other models, an organization may not need to spend time collecting commit data from other projects.

[1]https://android.googlesource.com

RQ 2 Which sub-project ranking method performs best in filtered-within-project JIT defect prediction?

In this research question, we build a filtered-within-project JIT defect prediction model that filters low-ranked sub-projects within the project, which are ranked by a score calculated by a rank scoring method. We use four different sub-project ranking algorithms: developer similarity-based ranking by us, domain-agnostic similarity-based ranking by Kamei et al. [11], size-based ranking, and random ranking. We then compare the cost effectiveness of four similarity algorithms with different the numbers of sub-projects selected to find the best-performing ranking algorithm for cross-sub-project JIT defect prediction.

By answering the research questions above, we conclude that 1) a JIT defect prediction model built with the training data from the same project is preferred over a model built with the training data from other projects; 2) the amount of training data is not the only factor that affects the cost effectiveness of cross-sub-project models; 3) a developer similarity that counts the number of developers in both sub-projects is the most preferred way of filtering out irrelevant sub-projects in building filtered-within-project models; and 4) although filtered-within-project models have lower cost effectiveness than within-project models, we notice that there is room for further improvement of the cost effectiveness of filtered-within-project models.

The main contributions of this paper are 1) proposing a sub-project-level defect prediction that can achieve higher cost effectiveness than the traditional project-level within-project method with developer similarity-based filtered-within-project models; and 2) evaluating the cost effectiveness of various cross-sub-project JIT defect prediction models on 232 sub-projects to check its feasibility.

The rest of the paper is organized as follows: Section II describes the experiment settings and Section III presents the design and result of experiments. Section IV provides related works and Section V reports threats to validity. Finally, Section VI conclude this study.

II. Experiment Setting

A. Studied Projects and Sub-projects

We select five open-source projects that are divided into multiple sub-projects. We try to choose projects with different characteristics in terms of programming language, domain, and distribution of the number of commits.

Table I shows the statistics of five projects. For each project, we count the number of sub-projects used in the experiments and the distribution of commit counts in each sub-project. We excluded sub-projects with less than 50 commits because we regard such sub-projects have not enough data to evaluate. During the experiment, we split commits from each sub-project into ten slices for a cross-validation. If a slice only contains clean commits without bug-inducing commits, we cannot evaluate the performance measure for that slice. If there are too many discarded slices, then performance measures with extreme value may occur due to the lack of valid evaluation

results. If a slice has five commits, the probability that the slice has at least one buggy commit could be calculated by $((1 - \text{avg. defect ratio})^5)$. Because the average defect ratio across the five projects we collected is 0.14, this probability is about 0.52. Thus, we can expect that near half of slices will give valid evaluation results from sub-projects with more than 50 commits. For Tizen and Android projects, we use the subset of sub-project within those project because of their large size. For the Tizen project, we use the sub-projects with the prefix of `platform/core/api`. For the Android project, sub-projects with the prefix of `platform/package/apps` are used. We also calculate the ratio of commits with any defect by using the SZZ algorithm [13], which is explained in Section II-C, and calculate the average number of sub-projects contributed per developer, which indicates the feasibility of using information about the developer who contributed in multiple sub-projects within a project to calculate the developer similarity between sub-projects.

B. Change Metric

We used 14 change metrics that are widely adapted in the JIT defect prediction field [4], [14]. Table II shows descriptions of these change metrics. In metric description, the term *subsystem* represents directories that are directly accessible from the root directory.

We apply some modification to those metrics because our study is conducted on the sub-project level, not a project level where those metrics are defined. Originally, developer experience-related metrics are defined under the scope of the project, but we changed this to the scope of the sub-project.

To prevent the multi-collinearity problem in prediction models, we exclude metric values that are correlated with any other metrics. We calculate the Pearson correlation coefficient between each pair of 14 metrics across 344,005 commits. Then, we regard pairs with Pearson correlation coefficient values higher than 0.8 as being correlated and exclude one of the metrics in the correlated pair. As a result, we excluded `ND`, `NF`, `LT`, `NDEV`, `NUC`, and `SEXP` and eight types of metrics are used throughout our experiments. These metrics are listed in bold text in Table II.

C. Labeling

To build and evaluate a prediction model, buggy or clean labels must be specified for each change. Instead of manual labeling, which will cost a lot of time, we use the SZZ algorithm [13] to label bug inducing commits automatically. SZZ algorithm firstly finds bug-fixing changes by inspecting log messages from each change. Then, the algorithm backtracks bug-inducing changes from bug-fixing changes by looking at change history of files which are modified by bug-fixing changes.

D. Pre-processing

1) Sampling Training Data: The number of buggy changes from a source code repository is less than that from a clean one, as shown in the *Defect Ratio* column in TableI. This could be a serious problem, because imbalanced training data could lead to biased prediction results. To deal with this problem, we apply an under-sampling method to the training data. This sampling method randomly removes instances with the majority label until the numbers of instances with the buggy and clean labels are the same.

2) Log Transformation: When investigating the extracted metric values, which are all non-negative, we notice that most of them are highly skewed. To make the distribution of metric values similar to a normal distribution, we apply a logarithm transformation ($\log_2(x+1)$) to all metric values.

E. Prediction Model

There are more than 30 classification model learners used in cross-project defect prediction researches [7]. Because different model learners work better for different datasets, we choose three popular classification model learners that were used by other defect prediction papers. Random forest (RF) [11], logistic regression (LR) [4] and naive Bayes (NB) [15] are selected for our experiments.

Those model learners build a classification model with training instances. An instance consists of the change metric values of a commit and a label whether the commit has any defect. When the change metric values from a new commit are given to a classification model, the model returns a probability that the commit has any defect, so called defect-proneness, or a binary classification whether this commit is buggy or clean. New commits which needs to be inspected for quality assurance can be prioritized by their buggy probability to make suspicious commits checked first. Since we use cost effectiveness as our performance measure, as explained in the next sub-section, we use cost-aware defect-prediction models to consider the cost of investigating a commit. Whereas a general defect-prediction model returns defect-proneness, a cost-aware defect-prediction model returns defect-proneness divided by the number of lines of code [16].

F. Performance Measures

Many previous cross-project defect prediction studies [7] have adapted precision, recall, f1-score, and the area under the receiver operating characteristic curve ($AUCROC$) as performance measures, which are widely used in prediction problems. These performance measures indicate how many testing instances are predicted correctly by a classification model. However, they do not consider the effort for QA testers to inspect the instances predicted as defects, which makes the measures less practical for QA testers [17].

Instead, we use the area under the cost-effectiveness curve ($AUCCE$) [18] as a performance measure in this experiment. This measure considers the effort to investigate the source code, which enables more practical evaluation.

As developers examine the commits that are ordered by the defect-proneness one by one, the total number of LOCs investigated and the total number of defects found will increase. The cost-effectiveness curve, which is a monotonic function, plots changes in the total percentage of LOCs investigated

TABLE I
STATISTICS ABOUT SIZE OF FIVE PROJECTS

Project	# of Sub-projects	Defect Ratio	Avg. # of Contributed Sub-projects per Dev.	# of Commits							
				Sum	*Mean*	*Std.*	*Min.*	*25%*	*50%*	*75%*	*Max.*
Android	45	0.12	2.68	248860	5530.22	7751.91	53.0	568.00	3070.0	6732.00	39449.0
Appium	32	0.21	1.79	18034	563.56	1108.50	51.0	127.50	253.0	541.00	6326.0
Cordova	38	0.16	2.60	23551	619.76	808.88	52.0	197.25	330.5	618.75	3494.0
Maven	69	0.19	6.50	43703	633.38	1288.80	51.0	187.00	296.0	614.00	10344.0
Tizen	48	0.21	2.09	9857	205.35	518.24	52.0	77.25	116.0	153.00	3677.0

TABLE II
DESCRIPTIONS OF CHANGE METRICS

Name	Description
NS	total # of changed subsystems
ND	total # of changed directories
NF	total # of changed files
Entropy	distribution of modified code across files
LA	total # of added code lines
LD	total # of deleted code lines
LT	average # of code lines before the change
FIX	whether containing fix related keyword in a change log
NDEV	average # of developers who touched a file so far
AGE	average # of days passed since the last modification
NUC	total # of unique changes
EXP	# of commits an author made in a sub-project
SEXP	# of commits an author made in a subsystem
REXP	# of commits an author made in a sub-project, weighted by commit time

to the horizontal axis and changes in the total percentage of defects found to the vertical axis. Thus, a higher $AUCCE$ value can be achieved if defect-inducing commits that changed only small amount of source code are investigated earlier.

Although $AUCCE$ values are always between 0 and 1, the maximum and minimum achievable $AUCCE$ values can differ across model learners and sub-projects, so it is difficult to understand the overall performance of defect prediction models. Thus, we normalized the $AUCCE$ value of a prediction model by dividing the value by the $AUCCE$ value of a within-sub-project model. This percentage of $AUCCE$ from the within-sub-project model ($\%WSP_{AUCCE}$)" shows the cost effectiveness a JIT prediction model achieves compared to the within-sub-project model.

III. EXPERIMENTS

A. RQ1: Is the cross-sub-project model trained by sub-projects from the same project more cost effective than that trained by sub-projects from other projects?

1) Design: We evaluate the median $\%WSP_{AUCCE}$ of the two cross-sub-project models across all sub-projects. The first model, a within-project model, is built with commits from sub-projects that belong to the same project as a target sub-project. The other, a cross-project model, is constructed with commits from sub-projects from the other four projects. The goal of both cross-sub-project models—within-project model and cross-project model—are predicting defect-inducing commits within the target sub-project. Fig. 1 shows which training data are selected for two cross-sub-project models and one within-sub-project model. A rectangle represents the commit data of each sub-project and a rounded rectangle represents a project. Sub-projects within a dashed line are selected as training data for each JIT defect prediction model.

In the process of building a JIT defect prediction model, especially when applying the under-sampling method, there is randomness that yields non-deterministic experimental results. Thus, we repeated the experiments 30 times to minimize the effect of randomness. In addition, we statistically tested whether the cost effectiveness of the two models is significantly different. We used the Wilcoxon signed-rank test [19] for the test as the $\%WSP_{AUCCE}$ values are paired between the two models and distribution of the $\%WSP_{AUCCE}$ values are not from a normal distribution. In addition, we calculated effect sizes of the Wilcoxon signed-rank tests to see the $\%WSP_{AUCCE}$ difference in the two models.

2) Result: With 41,760 experimental results (3 model learners $\times$ 2 cross-sub-project models $\times$ 232 sub-projects $\times$ 30 repetitions), Table III shows the median $\%WSP_{AUCCE}$ value of two models on different projects and model learners. The *Diff.* row represents the differences in the median $\%WSP_{AUCCE}$ between the two models and the number in the parenthesis represents the effect size between models. Stars next to the effect size represent that there is a statistically significant difference between the performance measures of the two models. Three stars to one star represent a statistical significance at 99% ($\alpha = 0.01$), 95% ($\alpha = 0.05$) and 90% ($\alpha = 0.1$) confidence intervals, respectively.

As shown in Table III, the within-project model outperformed the cross-project model in terms of the median $\%WSP_{AUCCE}$ value, except in cases when logistic regression and naive Bayes learners are used in the Android project and logistic regression is used in the Cordova project. Although the cross-project models are built with more training data than the within-project models, their cost effectiveness is generally lower than that of the within-project models. It shows an evidence that more training data does not mean better performance. These results show that when applying cross-sub-project JIT defect prediction, an organization may not need to collect change data from other projects because prediction models built with those data may not perform better than those built with change data from sub-projects

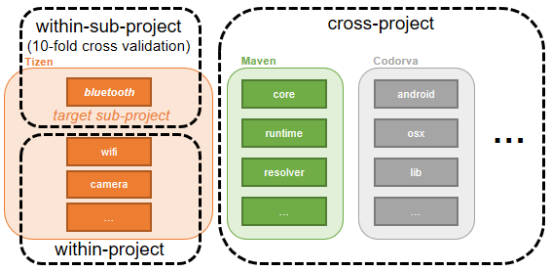

Fig. 1. Training data of within- and cross-project models

within the project. It is possible that filtering training data may increase the cost effectiveness of the cross-sub-project JIT prediction model. In this paper, however, we will focus on the feasibility of using within-project sub-project data and further data utilization would be handled in future work.

Another finding from the results is that the within-project models seem to have higher cost effectiveness than the within-sub-project models. Across all projects, except Android and all three model learners, the within-project models achieved a median $\%WSP_{AUCCE}$ value higher than 1.0, meaning that more than half of the within-project models achieved higher $AUCCE$ than the within-sub-project models. This result is different from the result of Kamei et al. [11] because, in their work, which was conducted at the project level, there was almost no cross-project JIT prediction models that achieved higher $AUCROC$ than the within-project model. We may explain a possible reason for the difference with the number of commits per sub-project. Table I shows that, except in the Android project, where the mean number of commits per sub-project is up to 27 times higher than in other projects, median commit counts (*50%* column) of the sub-project per project are near 500, which is much smaller than the number of commits per project (*Sum* column). We conducted additional experiments to confirm whether cross-sub-projects can be beneficial to sub-projects with a small number of commits, as explained in section I. The Spearman correlation coefficient between the number of commits in a target sub-project and the $\%WSP_{AUCCE}$ value of its within-project model is -0.379, indicating that there is a negative linear relationship. This means that the fewer commits a sub-project has, the greater the improvement in $AUCCE$ that its cross-sub-project models can achieve.

B. RQ2: Which sub-project ranking method performs best in filtered-within-project JIT defect prediction?

1) Design: We see from Section III-A2 that a small amount of training data could lead to better prediction performance. Thus, instead of using all the sub-project data within a project to build a cross-sub-project JIT defect-prediction model, we can filter out sub-projects that are less helpful in defect prediction to improve cost effectiveness of the JIT defect-prediction model. Fig. 2 shows how the training data of such models are selected. First, we calculate a score for each sub-project by using a ranking method. Then, we rank sub-projects by their scores and choose the top N sub-projects as training data for the filtered-within-project model. We use two different sub-project similarity-based ranking methods and two baseline ranking methods for this research question. The first is the developer similarity, which is our proposed method. The developer similarity is calculated with the number of developers who made any commit in both sub-projects. Sub-projects with more contributing developers in the target sub-project achieve higher similarity. As it can be seen in the average number of contributed sub-projects per developer in Table I, people tend to contribute to various sub-project within the project. The second is domain-agnostic similarity, used in the work by Kamei et al. [11]. Domain-agnostic similarity measure between the target sub-project and another sub-project is calculated in this order: 1) calculate the Spearman correlation between label values, which is 1 if a commit introduced any defect or 0 otherwise, and the values of each metric from the other sub-project; 2) select three metrics that achieved the highest Spearman correlation values; 3) calculate the Spearman correlation between each unordered pair of selected metrics for each sub-project. This generates a three-dimensional vector ($\binom{3}{2}$) for each sub-project; 4) Calculate the domain-agnostic similarity by the Euclidean distance between the two vectors. A smaller distance represents greater similarity.

Two baseline ranking methods we used are random and size rank. *Random rank* ranks the sub-projects randomly as a dummy baseline, and *size rank* ranks the sub-projects by the number of commits. The more commits a sub-project has, the higher the rank it achieves. Size ranking is comparable with our proposed ranking method because our method is correlated with the size of the sub-project.

We build cross-sub-project JIT defect prediction models for a target sub-project with commits from 1, 3, 5, 10, and 20 highest ranked sub-projects for each ranking method. Similar

TABLE III
MEDIAN $\%WSP_{AUCCE}$ VALUE OF WITHIN- AND CROSS-PROJECT MODELS ACROSS FIVE PROJECTS AND THREE MODEL LEARNERS

		Model Learner		
Project	**Model**	***LR***	***NB***	***RF***
Android	Cross	1.00	1.14	0.94
	Within	0.99	1.00	1.01
	Diff.	-0.01(-0.17**)	-0.14(-1.00**)	+0.06(+0.54**)
Appium	Cross	1.02	0.95	1.01
	Within	1.05	1.03	1.02
	Diff	+0.02(+0.77**)	+0.09(+0.89**)	+0.01(+0.08*)
Cordova	Cross	1.07	0.88	1.02
	Within	1.06	1.02	1.06
	Diff	-0.01(+0.02)	+0.15(+0.99**)	+0.04(+0.5***)
Maven	Cross	1.00	0.77	0.96
	Within	1.11	1.05	1.11
	Diff	+0.11(+0.98**)	+0.27(+1.00**)	+0.14(+0.96**)
Tizen	Cross	0.80	0.77	0.88
	Within	1.10	1.03	1.08
	Diff	+0.30(+0.99**)	+0.26(+0.98**)	+0.20(+0.96**)
All	Cross	0.99	0.90	0.96
	Within	1.06	1.02	1.05
	Diff	+0.07(+0.65**)	+0.13(+0.65**)	+0.09(+0.76**)

to RQ1, we evaluate the median $\%WSP_{AUCCE}$ values from each cross-sub-project JIT defect prediction model, repeat the experiments 30 times. P-values and effect sizes using the Wilcoxon signed-rank test are also calculated to compare performance of developer similarity based ranking method and other ranking methods.

2) Result: As Table IV shows, when only one sub-project is selected to build a filtered-within-project JIT defect prediction model, the developer similarity ranking method outperforms the domain-agnostic similarity and random ranking in all three learners and outperforms the size ranking in two learners. This result shows that a similarity measure designed for sub-project-level JIT defect prediction can be preferred over a similarity measure that is originally designed for cross-project JIT defect prediction when picking a sub-project to build a cross-sub-project JIT defect-prediction model.

As a prediction model is built with more sub-projects, median cost effectiveness generally increases and the differences of median $\%WSP_{AUCCE}$ value between four ranking methods become smaller. When we conduct experiments with more than 20 sub-projects selected for each model, the cost effectiveness of the various models becomes almost the same, so we do not include them in the table. This may be because as more number sub-projects are selected to build the filtered-within-project JIT defect prediction models, the models are trained with a more similar set of training data.

When we additionally refined our developer similarity ranking methods by not considering developers who barely contributed to the sub-project or by normalizing its value with the total number of developers who contributed in the other sub-project. Table V shows the comparison of performance measures normalization by the number of developers in the other sub-project is applied or not. Since filtering developer does not success in selecting similar sub-projects to get improved performance measures, we do not insert the table for that. However, in case of the normalization, it improved performance measures greatly when Naive Bayes learner is used and 3 to 5 sub-projects are selected. The improved performance measures even exceed the median $\%WSP_{AUCCE}$ value of within-project models which can be found in the "all" row at Table III.

In this research question, we see that the developer similarity-based cross-sub-project JIT defect prediction model is preferable to the other ranking methods. However, we notice that the median values of the performance measure achieved with filtered-within-project models (Table IV) are smaller than those achieved with within-project models (Table III). Still, we find the evidence that there is still room for improving the cost effectiveness of filtered-within-project models over that of within-project models with conducting additional experiments.

IV. Related Work

A. JIT Defect Prediction

Most studies on defect prediction have focused on predicting the defectiveness of software modules—files, packages, or functions—by utilizing project history data [20]. Recently, some research has been performed on just-in-time defect prediction, which predicts software changes that may introduce defects.

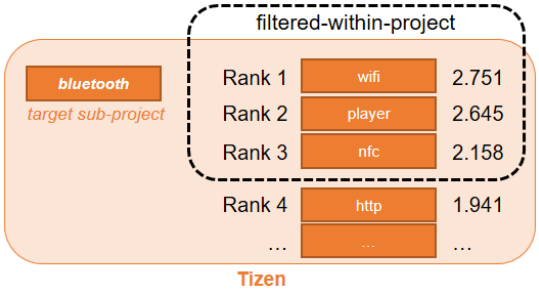

Fig. 2. Training data of filtered-within-project models

Kamei et al. conducted experiments applying just-in-time defect prediction to six large open-source projects and five large commercial projects [4]. To build the defect prediction models, they used 14 change metrics in five different dimensions—diffusion, size, purpose, history, and experience—and used the logistic regression learner. These change metrics are widely used in other just-in-time defect prediction studies [21], [22]. Kim et al. proposed a change level defect prediction method and tested it on 12 open-source software projects [5]. They used a set of text-based metrics that were extracted from the source code, log messages, and file names. In addition, they used metadata such as the change author and commit time. They also considered changes in the complexity of source code caused by commits. There are many other just-in-time defect prediction studies, such as applying deep learning [21] or unsupervised models [22] for JIT defect prediction. However, applying just-in-time defect prediction on projects that consist of sub-projects is not discussed yet.

B. Cross-Project Defect Prediction

Zimmermann et al. defined 30 project characteristics and showed the influence of the similarity of each characteristic between target and predictor projects on a module-level cross-project defect prediction [23]. They considered the project domain, programming language, company, quality assurance tools, and other aspects as project characteristics and concluded that the characteristics of a project that transfers a defect prediction model can influence precision, recall, and accuracy in cross-project defect prediction. However, we notice that this finding is barely applicable in cross-sub-project defect prediction, as sub-projects within the project share many common characteristics. For instance, sub-projects within the same project usually are developed by the same company using the same programming language and development tools. He et al. [8] investigated the feasibility of cross-project defect prediction at the module-level on 10 open-source projects. They concluded that a model trained with data from other projects can achieve higher accuracy than one trained with data from the target project in the best cases. In addition, they said that selecting the training dataset considering the distributional

TABLE IV
MEDIAN $\%WSP_{AUCCE}$ VALUE OF VARIOUS FILTERED-WITHIN-PROJECT MODELS ACROSS FIVE PROJECTS AND THREE MODEL LEARNERS

Model Learner	Ranking Method	# of Sub-projects Selected			
		1	*3*	*5*	*10*
LR	Developer	1.02	1.04	1.05	1.06
	Domain-agnostic	1.01(-0.02, -0.26**)	1.04(-0.01, -0.13**)	1.05(-0.00, -0.05**)	1.05(-0.01, -0.05**)
	Random	0.99(-0.04, -0.38**)	1.03(-0.02, -0.30**)	1.04(-0.01, -0.22**)	1.05(-0.01, -0.18**)
	Size	1.04(+0.02, +0.30**)	1.05(+0.01, -0.13**)	1.05(+0.00, -0.15**)	1.06(-0.00, -0.20**)
NB	Developer	1.03	1.03	1.03	1.03
	Domain-agnostic	1.01(-0.02, -0.25**)	1.04(+0.01, +0.19**)	1.04(+0.01, +0.24**)	1.04(+0.01, +0.24**)
	Random	1.01(-0.02, -0.26**)	1.03(+0.00, -0.01)	1.03(+0.00, +0.04**)	1.03(+0.00, +0.13**)
	Size	1.00(-0.03, -0.33**)	1.01(-0.02, -0.47**)	1.01(-0.02, -0.46**)	1.01(-0.02, -0.30**)
RF	Developer	0.98	1.01	1.02	1.03
	Domain-agnostic	0.96(-0.02, -0.14**)	1.01(-0.00, -0.09**)	1.02(-0.00, -0.09**)	1.03(+0.00, -0.01)
	Random	0.94(-0.04, -0.29**)	0.99(-0.02, -0.30**)	1.01(-0.02, -0.29**)	1.02(-0.01, -0.20**)
	Size	0.96(-0.02, -0.18**)	0.98(-0.03, -0.44**)	0.99(-0.03, -0.45**)	1.02(-0.02, -0.32**)

TABLE V
MEDIAN $\%WSP_{AUCCE}$ VALUE OF NORMALIZED DEVELOPER SIMILARITY-BASED FILTERED-WITHIN-PROJECT MODELS ACROSS FIVE PROJECTS AND THREE MODEL LEARNERS

Model Learner	Developer Similarity	# of Sub-projects Selected			
		1	*3*	*5*	*10*
LR	Not Normalized	1.02	1.04	1.05	1.06
	Normalized	0.95(-0.08, -0.62**)	1.02(-0.03, -0.46**)	1.03(-0.02, -0.38**)	1.05(-0.01, -0.32**)
NB	Not Normalized	1.03	1.03	1.03	1.03
	Normalized	0.99(-0.04, -0.30**)	1.06(+0.03, +0.22**)	1.06(+0.03, +0.24**)	1.04(+0.02, +0.25**)
RF	Not Normalized	0.98	1.01	1.02	1.03
	Normalized	0.92(-0.06, -0.43**)	0.98(-0.03, -0.35**)	1.00(-0.03, -0.34**)	1.02(-0.01, -0.22**)

characteristics of datasets could lead to better cross-project results. However, this characteristic could not be extracted in projects where no historical data exist, which hinders the application of distributional characteristics in cross-project defect prediction. In addition, neither Zimmermann et al. nor He et al. considered JIT defect prediction and the cost effectiveness of the defect-prediction model.

Kamei et al. [11] conducted cross-project defect prediction experiments in a change-level manner. They observed that predicting defects for changes in a target project with a model built with another project's data has lower accuracy than that of a within-project model. However, when the prediction is done with a model built with a larger pool of training data from multiple projects or combining the prediction results from multiple cross-project models, its performance is indistinguishable from that of a within-project model. They also applied domain-aware or domain-agnostic similarity measures between two projects to select a similar project. For the domain-agnostic type, they calculated the Spearman correlation between the metric values within a dataset and used the correlation values to find a similar project. For the domain-aware type, they used a subset of project characteristics proposed by Zimmermann et al. [23] and calculated the Euclidean distance to find a similar project. When these similarity measures are used to pick one project to transfer its JIT defect-prediction model to predict defects in the target project, they concluded that both measures successfully contributed to pick a better-than-average cross-project model. However, when they built a cross-project JIT defect prediction model with training data from multiple similar projects, it barely improved accuracy over a model trained with data from all other projects without filtering irrelevant projects. This work showed that cross-project defect prediction is feasible on JIT defect prediction, but there were no discussions on the sub-project-level JIT defect prediction. Moreover, the cost was not considered of investigating commits to find defects in the evaluation.

V. THREAD TO VALIDITY

A. Construct Validity

For our experiments, we implemented Python scripts to extract the change metric data from the source code repositories to build and test JIT defect prediction models. However, the scripts might have defects that affect the experiments and results. To reduce this threat, we used open-source frameworks and libraries that are commonly used in other studies, such as *scikit-learn* [24]. In addition, we also double-checked our source code and manually inspected extracted change measures for validation.

B. Dataset Quality

We used *CodeRepoAnalyzer* by Rosen et al. [25] to extract change metrics from git repositories. While using this tool, we noticed that it has some bugs. For instance, some extracted metric values were marked as negative, which is incorrect by definition. Although we handled the bugs found in this tool, there could be other bugs that were not found and that could have affected the extracted values.

Although the SZZ algorithm is widely used in JIT defect prediction research [4], [12], it is known that the accuracy of keyword-based labelling methods for bug-inducing commits are limited [26]. We may improve the accuracy of automatic labeling by utilizing bug-repository data [13].

VI. Conclusion

In this paper, we investigated the feasibility of transferring JIT defect prediction models built with data from other sub-projects to predict bug-inducing commits in a target sub-project. We conducted experiments with five projects, which comprise 232 sub-projects in total, and three different model learners. With two research questions, we conclude that 1) a cross-sub-project model has better cost effectiveness than within-sub-project models in general; 2) a cross-sub-project JIT defect prediction model built with data from sub-projects within the targets project has higher cost effectiveness than a JIT defect prediction model built with data from all available sub-projects; 3) the developer similarity-based ranking method is preferable for filtering sub-projects that are irrelevant to the target sub-project; and 4) although a developer similarity-based filtered-within-project model has lower cost effectiveness than a within-project model in general, we further improved the performance of the filtered-within-project model to outperform the within-project model in the best cases. Our contributions include 1) proposing defect prediction at the sub-project level that potentially has better cost effectiveness than traditional within-project models by using new developer-similarity-based filtered-within-project models and 2) initially evaluating the cost effectiveness of various sub-project-level JIT defect prediction models across 232 sub-projects. In future work, We plan to investigate a more polished way to apply filtered-within-project models, such as filtering developers by considering their contributions over various project resources [27] before calculating the developer similarity.

References

[1] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium On*. IEEE, 2004, pp. 417–428.

[2] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 423–433, May 1992.

[3] R. Moser, W. Pedrycz, and G. Succi, "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 181–190.

[4] 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.

[5] S. Kim, E. J. W. Jr, and Y. Zhang, "Classifying Software Changes: Clean or Buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, Mar. 2008.

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

[7] S. Herbold, "A systematic mapping study on cross-project defect prediction," *arXiv:1705.06429 [cs]*, May 2017.

[8] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Software Engineering*, vol. 19, no. 2, pp. 167–199, Jun. 2012.

[9] E. Shihab, Y. Kamei, and P. Bhattacharya, "Mining Challenge 2012: The Android Platform," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, ser. MSR '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 112–115.

[10] M. A. Babyak, "What you see may not be what you get: A brief, nontechnical introduction to overfitting in regression-type models," *Psychosomatic Medicine*, vol. 66, no. 3, pp. 411–421, 2004 May-Jun.

[11] 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, Oct. 2016.

[12] T. Jiang, L. Tan, and S. Kim, "Personalized Defect Prediction," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 279–289.

[13] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5.

[14] X. Yang, D. Lo, X. Xia, and J. Sun, "TLEL: A two-layer ensemble learning approach for just-in-time defect prediction," *Information and Software Technology*, vol. 87, pp. 206–220, Jul. 2017.

[15] B. Turhan, T. Menzies, A. B. Bener, and J. D. Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, Oct. 2009.

[16] T. Mende and R. Koschke, "Effort-Aware Defect Prediction Models," in *2010 14th European Conference on Software Maintenance and Reengineering*, Mar. 2010, pp. 107–116.

[17] Y. Kamei and E. Shihab, "Defect Prediction: Accomplishments and Future Challenges," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, Mar. 2016, pp. 33–45.

[18] T. Mende and R. Koschke, "Revisiting the Evaluation of Defect Prediction Models," in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, ser. PROMISE '09. New York, NY, USA: ACM, 2009, pp. 7:1–7:10.

[19] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[20] J. Nam, "Survey on software defect prediction," *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep*, 2014.

[21] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep Learning for Just-in-Time Defect Prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, Aug. 2015, pp. 17–26.

[22] W. Fu and T. Menzies, "Revisiting Unsupervised Learning for Defect Prediction," *arXiv:1703.00132 [cs]*, Feb. 2017.

[23] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 91–100.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, and D. Cournapeau, "Scikit-learn: Machine Learning in Python," *MACHINE LEARNING IN PYTHON*, p. 6.

[25] C. Rosen, B. Grawi, and E. Shihab, "Commit Guru: Analytics and Risk Prediction of Software Commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 966–969.

[26] T. Hall, D. Bowes, G. Liebchen, and P. Wernick, "Evaluating Three Approaches to Extracting Fault Data from Software Change Repositories," in *Product-Focused Software Process Improvement*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Jun. 2010, pp. 107–115.

[27] G. Gousios, E. Kalliamvakou, and D. Spinellis, "Measuring Developer Contribution from Software Repository Data," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 129–132.

Boost Symbolic Execution Using Dynamic State Merging and Forking

Chao Zhang
School of Software, Tsinghua University, China
zhang-chao13@mails.tsinghua.edu.cn

Weiliang Yin
HUAWEI, China
yinweiliang.ywliang@huawei.com

Zhiqiang Lin
HUAWEI, China
linzhiqiang@huawei.com

***Abstract*—Symbolic execution has achieved wide application in software testing and analysis. However, path explosion remains the bottleneck limiting scalability of most symbolic execution engines in practice. One of the promising solutions to address this issue is to merge explored states and decrease number of paths. Nevertheless, state merging leads to increase in complexity of path predicates at the same time, especially in the situation where variables with concrete values are turned symbolic and chances of concretely executing some statements are dissipated. As a result, calculating expressions and constraints becomes much more time-consuming and thus, the performance of symbolic execution is weakened in contrast. To resolve the problem, we propose a merge-fork framework enabling states under exploration to switch automatically between merging mode and forking mode. First, active state forking is introduced to enable forking a state into multiple ones as if a certain merging action taken before were eliminated. Second, we perform dynamic merge-fork analysis to cut source code into pieces and continuously evaluate efficiency of different merging strategies for each piece. Our approach dynamically combines paths under exploration to maximize opportunities for concrete execution and ease the burden on underlying solvers. We implement the framework on the foundation of the symbolic execution engine KLEE, and conduct experiments on GNU Coreutils code using our prototype to present the effect of our proposition. Experiments show up to 30% speedup and 80% decrease in queries compared to existing works.**

***Index Terms*—State Merging, Active State Forking, Local Merge-fork Analysis**

I. Introduction

The innovation of symbolic execution has been proposed over four decades ago [1], [2]. It shows great potential in software analysis and testing, and underlies a growing number of tools which are widely applied in practical cases [3]–[7]. For example, in [5], [6], they apply symbolic execution to greatly improve the performance of traditional fuzz testing [8], [9]. In [7], they incorporate symbolic execution as a bridge to reduce the false positive of static analysis [10] and the false negative of dynamic analysis and run-time verification [11]The key idea behind symbolic execution is to replace input values with symbols, update symbolic expressions for program variables, and calculate path predicates along explored execution paths to determine their feasibility [12], [13]. In this way, symbolic execution simulates how a program runs and explore feasible paths in an efficient way.

However on the other hand, path explosion is still the most significant bottleneck which limits scalability of symbolic execution. When traversing a program, each path encodes decisions on all its historical branches and maintains them in its path predicate [14]. Thus, the number of paths to be explored grows exponentially with the number of branches and exhaustively covering all paths could be extremely time-consuming even for a small program [15].

State-of-the-art solutions to resolve path explosion mainly aim at directly decreasing number of paths or optimizing the process within limited time budgets. A number of articles introduce various search heuristics to improve performance towards particular targets. For example, ideas from random testing and evolutionary algorithms are frequently used to guide the search order of path exploration so that coverage gets higher in the same time [16], [17]. The second way is to apply some imprecise techniques from static analysis, such as abstract interpretation and summary. These techniques calculate approximations to simplify analysis of code and decrease produced states [18], [19]. State merging is another way designed to cut down number of paths to be explored. When path exploration encounters a branch statement, there are two succeeding states constructed and in turn the original path is split into to two sub-paths, corresponding to true and false case respectively. If the two states join at the same location later, they can be merged into one again, with path predicate updated using disjunction and variable expressions using *ITE* operation [20]. Fig.1 shows an example where there are two paths in the given program: $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6$ corresponding to true case of if-statement and $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$ to false case. Succeeding states of 3 and 4 are merged at 5, with path predicate set to $n < 0 \vee \neg(n < 0)$ or simply $TRUE$, and expression of ret set to $ITE(n < 0, -n, n)$.

State merging is intended to decrease the number of paths to be explored while no imprecision is introduced. Meanwhile, it faces the challenge that path predicates and variable expressions may get more complex. Constraints involving disjunction and ITE (actually another form of disjunction) are extremely unfriendly to underlying solvers [21]. Such negative impact on performance is rather serious when expressions which can be evaluated concretely turn symbolic after merging. Modern symbolic execution engines commonly adopt concrete execution when code to be interpreted only involves concrete values. Improper merging actions may dismiss chances of concrete execution and lead to extra solving cost, which eliminates the benefit of decrease in path scale in reverse [20]. Actually, a

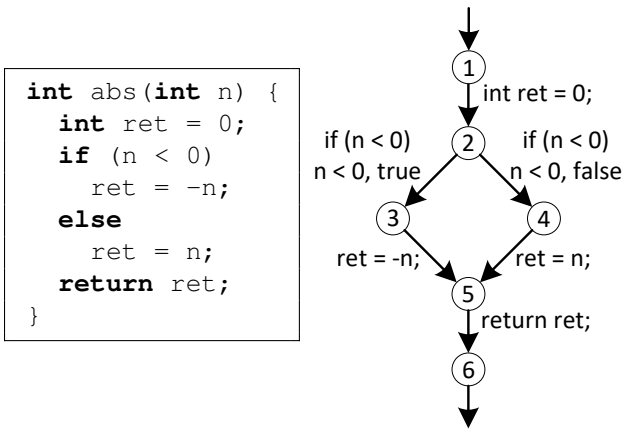

Fig. 1: An example of state merging.

merging action will speed up symbolic execution or not, or even speed down it, is determined by the future code which is still unexplored. Kuznetsov et al. have proposed an estimation algorithm in [14] to statically find variables frequently used in branch conditions and prevent any merging action erasing concrete values of these variables. In this way, state merging is selectively performed to exclude inefficient cases. We are inspired by this work and try to improve merging strategies to handle more complicated situations. We build a framework dynamically merging and forking states according to the code under execution. During execution, combinations of paths vary to keep chances of concrete execution while exploration of states and paths are joint as much as possible.

Contributions. In this paper, we present a framework boosting symbolic execution by automatically switching between merging states together and forking them separately throughout path exploration. We implement a prototype on the foundation of KLEE [22] and conduct experiments on GNU Coreutils code to evaluate the method. Our main contributions are:

- We introduce active state forking to eliminate a historical merging action and restore the state into original separate ones. By integrating symbolic execution with state merging and forking, we can change the way how states are combined and paths are explored at any moment during execution.
- We present dynamic merge-fork analysis to evaluate performance trade-offs for different path combinations. Decisions are made by estimating chances of concrete execution and efforts in reasoning future statements. The source code is cut into pieces so analysis is simplified and conclusions can be drawn in each piece locally.

The remainder of this paper is organized as follows. In Section II we have an overview of symbolic execution and state merging. Section III introduces active state forking, followed by description of dynamic merge-fork analysis in Section IV. Next we display experimental evaluation of a prototype implementing our merge-fork framework in Section V. Finally we give our conclusion in Section VI.

II. Symbolic Execution and State Merging

In this section, we first present a general process of symbolic execution and then have a discussion on state merging.

A. Classical Symbolic Execution

Symbolic execution starts from the program entry and keeps updating variable expressions and path predicates to systematically explore feasible paths. In this way the executor can reveal code reachability, find bugs or generate test inputs. Descriptions of a general work-flow of classical symbolic execution techniques are given in many works [14], [15]. We refer to these works and present a general algorithm in Algorithm 1. To be concise but without loss of generality, we consider a language made up of three kinds of statements, i.e. assignments, conditional branches and exit statements with exit code to indicate normal termination or failed assertion/error. By unwinding loops and inlining function calls we can transform a program to this form. In the paper, all programs used to describe our method are in such form if not explained particularly.

In symbolic execution, a state of the program is usually encoded as a triple (l, P, M) where l refers to the current program location, P to path predicate corresponding to the path from the entry to l, and M to symbolic store mapping from variables to their current expressions. The algorithm keeps taking out a state from $Worklist$, extending the state and pushing newly found ones back to $Worklist$. Call to $selectState$ at line 3 determines which state is chosen to be visited next, and thus implies the searching strategies of path exploration. The chosen state is then updated according to the next statement just after its location (line 5-21). Call to $eval$ and $checkCond$ in this procedure symbolically calculates expressions and checks satisfiability.

Code at line 22-30 of Algorithm 1 is designed for merging states. For a newly found state s, if there is another state s' in $Worklist$ sharing the same program location, s could be merged with s' rather than directly added to $Worklist$. Definition of $shouldMerge$ makes decisions on whether to merge and thus realizes strategies of state merging.

B. State Merging: SSE vs. DSE

By defining $selectState$ and $shouldMerge$ differently, the classical symbolic execution algorithm presented in Section II-A acts as two extremes in regard to design space of state merging. Dynamic symbolic execution (DSE, adopted in [3], [22]–[24] etc.) skips merging and every execution state encodes a unique path from the entry to its corresponding location. Static symbolic execution (SSE) ensures all paths joining at each junction node in the control flow structure are merged completely, which is common in symbolic model checking and verification condition generation [25], [26].

SSE systematically explores all feasible control paths with all possible input values taken into consideration [20]. It visits

Algorithm 1 Classical symbolic execution

```
1:  Worklist = {(l_0, true, M_0)}
2:  while Worklist ≠ ∅ do
3:    (l, P, M) = selectState(Worklist)
4:    Worklist = Worklist \ {(l, P, M)}
5:    switch nextInst(l) do
6:      case v := e:
7:        M' = M \ {(v, M[v])} ∪ {(v, eval(e, M))}
8:        S = {(succ(l), P, M')}
9:      case if cond goto l':
10:       sat = checkCond(cond, P, M)
11:       if sat == CONST_TRUE then
12:         S = {(l', P, M)}
13:       else if sat == CONST_FALSE then
14:         S = {(succ(l), P, M)}
15:       else if sat == NON_CONST then
16:         S = {(l', P ∧ eval(cond, M), M), (succ(l), P ∧
              ¬eval(cond, M), M)}
17:       end if
18:     case exit:
19:       S = ∅
20:       printPath()
21:     end switch
22:   for all s = (l, P, M) ∈ S do
23:     for all s' = (l', P', M') ∈ Worklist do
24:       if l = l' ∧ shouldMerge(s, s') then
25:         Worklist = Worklist \ {s'}
26:         M'' = {(v, ITE(P, M[v], M'[v]))|(v, M[v]) ∈
            M}
27:         S = S \ {s} ∪ {(l, P ∨ P', M'')}
28:       end if
29:     end for
30:   end for
31:   Worklist = Worklist ∪ S
32: end while
```

states in topological order to ensure that all corresponding states are merged at every junction node in control flow of the program. However, its major trade-offs are great difficulties in calculating path predicates and strict restrictions on searching strategies. DSE techniques are then proposed to interleave concrete and symbolic execution and focus on exploring possible execution paths one by one. A statement can be executed concretely rather than symbolically if all involving variables are concrete. For an assignment, the result is calculated directly if possible. For a conditional branch $if\ cond\ goto\ l'$, concrete execution is performed if $cond$ can be determined concretely. As a result, interpreting the statement does not need to split the current path. DSE techniques such as [3], [23], [24] introduce concrete inputs to make analysis benefit from concrete execution.

Taking both SSE and DSE into consideration, it requires a compromise of the two extremes to use state merging to deal with path explosion in real code. While dynamic techniques remain to keep symbolic execution benefiting from concrete execution and heuristic search order, states are selectively merged to eliminate duplicated efforts exploring paths joining at a certain location. The work of [14] tries to introduce state merging in DSE and switch between different searching strategies. The executor keeps discovering merging opportunities with positive effects on performance and prioritizes visit of involving states. Veritesting [27] proposes combination of SSE and DSE by changing execution mode in process. SSE makes code analyzed one-pass instead of visiting all feasible paths separately in DSE. In this way the executor is able to benefit from solvers and boost path exploration.

Our proposition tries to mix SSE and DSE on the foundation of [14]. In this article, Kuznetsov et al. have constructed a framework to estimate whether it is worth driving path exploration towards a potential merging action. They present a static bottom-up analysis to find variables which may be frequently used later and suggest merging actions with no impact on values of these variables. Path exploration is then under control of an SSE-like module temporally instead of a normal DSE process. We take a more complicated case into consideration where evaluation on a merging action varies across the code. A static decision - either merging or not - improves efficiency for some pieces of code but burdens performance for others. Actually, an optimal solution usually consists of multiple parts which are drawn in different pieces of code locally and combined together dynamically.

III. Active State Forking

In this section, we depict our active state forking technique. First, we use a series of examples to explain motivation for forking and then describe definitions and method of it.

A. Motivating Examples

As is discussed in last chapter, influence of a certain merging action varies throughout different pieces of code. We use some example code in Fig.2 to explain the issue in detail and present effect of active state forking. To be concise, some function calls are not inlined.

In function $f1$ state merging dismisses the opportunity of concrete execution. There are two if-blocks in $f1$, namely line 3 and line 4-5. State forks and path splits when reaching the first if-block (this is passive state forking caused by non-concrete if-condition), and then two sub-paths join after it. Expression of $flag$ gets different along two sub-paths, and is instantly used in if-condition at line 4. Expression of $flag$ is 1 along true case of the first if-block, 0 along false case, and $ITE(a == b, 1, 0)$ if states are merged before line 4. It can be seen that both the two separate states lead concrete execution on the second if-block since its condition can be decided concretely, but the merged one erases concrete value of $flag$ and results in passive state forking. Therefore, the merging action after the first if-block just makes expressions and path predicate more complex while number of paths to explore is not changed.

```
1  void f1(int a, int b) {
2    int flag = 0;
3    if (a == b) flag = 1;
4    if (flag) g1();
5    else g2();
6    exit(0);
7  }
8
9  void f2(int a, int b) {
10   int flag = 0;
11   if (a == b) flag = 1;
12   if (flag) g1();
13   else g2();
14   g3(a, b);
15 }
16
17 void f3(int a, int b) {
18   int flag = 0;
19   if (a == b) flag = 1;
20   g3(a, b);
21   if (flag) g1();
22   else g2();
23 }
```

Fig. 2: Motivating examples.

In function $f2$, $exit(0)$ at line 6 in $f1$ is replaced with another function call $g3(a, b)$. While things go the same as for $f1$, it is required to reason about call to $g3$ on each path. If paths are merged before line 14, call to $g3$ will be analyzed only once rather than once along each path. Hence, it is expected that states forked in the first if-block should be merged at the exit of the second if-block.

For function $f3$, call to $g3$ goes before the second if-block. As is discussed for $f1$ and $f2$, we can make a similar conclusion that states forked in the first if-block should be merged immediately to avoid repeated efforts reasoning about call to $g3$, but later "restored" to two separate paths with concrete value for $flag$ along each. Common methods try to compare advantages and disadvantages to reveal a static decision on state merging, and our approach, in another way, makes it possible to fork a merged state as if a historical merging action were not taken. In $f3$, paths are made to join after line 19 and fork actively before line 21 with $flag$ restored to concrete along each forked path. In this way, the benefit of state merging is maximized while its negative impact on concrete execution is avoid as far as possible.

B. Merge and Fork States with Extended Execution Graph

The motivating examples figure out that an optimal merging strategy may promote contrast decisions for different pieces of code. Active state forking together with state merging makes it possible to change how paths are combined during exploration. To maintain information of historical merging actions and support forking states, we make extensions to the formalism in classical symbolic execution.

A DSE method as shown in Algorithm 1 conducts path exploration by visiting execution states and creating new ones. This process can be efficiently represented as a graph in which nodes are execution states and edges indicate transitions between them. Since DSE does not merge states, such a graph would be a tree, which is so called symbolic execution tree. Definitions of execution state and symbolic execution tree are given in Definition 1 and 2.

***Definition** 1:* An execution state is defined as a tuple $es = (l, P, M)$ where

- $l \in L$ is the current program location.
- $P \in \mathcal{F}(\Sigma, Const)$ is the path predicate, which is a formula over symbolic inputs Σ and constants $Const$ encoding all decisions of branches along the path from entry to l.
- $M : V \rightarrow \mathcal{F}(\Sigma, Const)$ is the symbolic store, which maps a program variable to the formula corresponding to its expression.

***Definition** 2:* A symbolic execution tree is a tree $T = (ES, T)$ where

- ES is the set of execution states.
- $T \subseteq ES \times ES$ is the set of transitions between execution states.

Symbolic execution tree records every explored state and path, and is widely used to characterize the process of symbolic execution [25]. Our extended formalism is built on top of it. While state merging is performed, an execution state could encode multiple nodes in the corresponding symbolic execution tree. Such relations naturally extract information of merging actions, and thus, we introduce virtual states to hold the similar structure. Definition 3 presents how virtual states are defined.

***Definition** 3:* A virtual state is defined as a tuple $vs = (es, CV)$ where

- $es \in ES$ is the corresponding execution state.
- $CV \subseteq V \times Const$ is the set of variables with concrete values together with their values.

When two execution states are merged to be one, we create virtual state for each sub-state respectively and map them to the produced state. Every virtual state tracks its exclusive concrete variables. Our formalism then constructs transitions between execution states and virtual states, resulting in a directed acyclic graph. An execution state contains data of path predicate and symbolic store, and is used to actually execute statement. It appears as a node in the graph only if it never experiences merging or we are not concerned with its historical merging actions anymore. Otherwise, its corresponding virtual states are used instead to make up the graph. Definition 4 describes our extended execution graph formally.

***Definition** 4:* An extended execution graph is a directed acyclic graph $G = (S, T, \psi, \Delta)$ where

- $S \subseteq ES \cup VS$ is the set of states.
- $T \subseteq S \times S$ denotes transitions between states.
- $\psi : ES \rightarrow 2^{VS}$ maps a execution state to the set of its corresponding virtual states.

- $\Delta : VS \to \mathcal{F}(\Sigma, Const)$ maps a virtual state to its newly added conditions compared to its predecessor's path predicate.

While execution states hold essential data and drive execution to explore paths, virtual states simulate the structure of symbolic execution tree and preserve merging history. If there is no further operation on virtual state, our extended execution graph would degenerate into a tree. However, virtual states are merged and swapped for execution states when updates on variables efface complexity introduced by merging. In the remaining part of this section, we explain in detail how to perform state merging and forking with our extended execution graph.

Merging. When trying to merge $es_1 = (l, P_1, M_1)$ and $es_2 = (l, P_2, M_2)$ into $es_3 = (l, P_1 \vee P_2, M)$ where $M(v) = ITE(P_1, M_1(v), M_2(v))$ for each $v \in V$, we first create virtual states and then update the graph $G = (S, T, \psi, \Delta)$.

If es_1 appears as a node in G (in other word $es_1 \in S$ and $\psi(es_1) = \emptyset$), we create a virtual state directly to represent its corresponding sub-state encoded by es_3: $vs = (es_3, CV)$. CV denotes exclusive concrete variables of this sub-state, so it is made up of variables concrete in es_1 but not in es_2. That is, $CV = \chi(M_1) \setminus \chi(M_2)$ where $\chi(M) = \{(v, M(v) | M(v) \in \mathcal{F}(Const)\}$.

If es_1 encodes multiple sub-states so that $es_1 \notin S$ or $\psi(es_1) \neq \emptyset$, we create a virtual state for every sub-state in $\psi(es_1)$ and add up CV of sub-state and exclusive concrete variables of es_1 to assign to newly created one.

Creating virtual states for es_2 is in the same way, except that exclusive concrete variables of es_2 is calculated as $\chi(M_2) \setminus \chi(M_1)$.

All these generated virtual states are then added to G, together with corresponding transitions. $\psi(es_3)$ is set to be all these new states. Δ of them can be simply set to empty since there is no change between their and their predecessors' path predicates.

Update of Assignment. Updating an execution state es in G can be done immediately. However, if $es \notin S$, set of concrete variables of each virtual state $vs = (es, CV) \in \psi(es)$ requires update. Those not concrete or not exclusively concrete anymore should be removed from concrete variables, while those becoming exclusively concrete should be added to concrete variables.

E.g. for assignment $v := e$, $(v, c_v) \in CV$ is removed if e is not concrete, or it is but calculation does not involve any concrete variables in CV. Otherwise v and its value is added to CV if there is no $(v, c_v) \in CV$, and e is concrete and depends on certain variables and values in CV.

When updating virtual states, ones might point to the same execution state and possess equal concrete variables and values. These states can be merged in the graph since distinguishing such sub-states makes no sense to merging and forking. If all virtual states of an execution state are merged, we can swap the production for the execution state itself, which means that all its historical merging actions do not influence any concrete variable and hence, will never invoke active forking.

Forking. For an execution state es in G, passive forking by conditional branch is the same as in traditional DSE. Otherwise, forking action is applied to every sub-state $vs \in \psi(es)$ and Δ is updated with branch decision accordingly for forked productions.

In the case of active forking aimed at eliminating historical merging actions, the work flow gets more complicated: it is required to partitioning sub-states, constructing path predicates and updating expressions.

The execution state es to be forked encodes multiple sub-states which are represented by virtual states in $\psi(es)$. Active forking is invoked because some variables are turned from concrete to symbolic by merging actions. Thus, the first step is to identify concerned variables. Every virtual state $vs \in \psi(es)$ where these variables are concrete is separated to be a forked state. For sake of efficiency, states with same concrete values for these variables can be joined. In this way, all sub-states of es are partitioned into several groups and each group corresponds to a potential forked state.

For each sub-state vs, its path predicate can be calculated by backtracking to its ancestor execution state and combining with newly added conditions denoted by Δ along the path. The results are used to construct path predicate for each potential forked state and decide ITE selections of variables expressions. The original state es is then forked into several execution states as if some historical merging actions were eliminated. Besides, it is remarkable in real case that optimization of expressions and constraints based on data structure, constant propagation, query cache and other practical techniques is adopted to cut down efforts and boost speed of calculation in state forking.

IV. Dynamic Merge-fork Analysis

Active state forking makes it possible to raise dynamic merging strategies, and next task is to identify "good" merging actions. In this section, we will explain how to identify possibilities of concrete execution and how to make decisions on merging by analyzing code piece by piece.

A. Identify Concrete Values and Concrete Execution

State merging avoids repeated efforts in analysis but may burden underlying solvers as trade-off. Considering examples in Fig.2, the ideal solution would be that we can selectively merge states so that 1) variables turned symbolic are used to calculate future if-conditions as little as possible; 2) merging actions with no influence on future if-conditions are as many as possible. The former prevents state merging from being obstacle to concrete execution while the latter indicates greater benefit from merging.

A bottom-up estimation identifying variables frequently used is presented in [14]. The basic idea is tracking variables used in if-conditions along use-def chain. Our method also pays attention to use in if-condition, but works in top-down order instead because analysis will be break down in different pieces of code. A list of variables with concrete values is maintained and updated when executing. Thus, we can identify

possible concrete values and in turn, find out chances of concrete execution.

B. Local Merge-fork Analysis

Active state forking makes it possible to redraw decisions on merging actions, so we can cut a program into pieces and analyze code piece by piece. The division is randomly made under these rules: difference in scale between pieces is limited within a proper bound; cut points are located at exit of if-blocks, end of inlined functions, or terminators of sequential blocks. And a piece is divided further if it contains greatly different parts, e.g. variables used in one part are totally disjoint with ones in another. Nevertheless, too detailed division achieves little increase in performance while goes time-consuming, so we limit the times of further division.

After code is divided into pieces, we can make decision for states whether they should be merged or not in a piece. There are still situations where different parts of code have opposite votes for a merging action in the same piece. To resolve the issue, we propose another two types of decision: merge until a location and merge after a location. The former performs merging temporally and invokes analysis again later, while the latter make a delay to merging. Because of these choices of merging decision, analysis of how to combine paths is invoked at junction node in control flow structure, at the beginning of each piece, and at locations specified by ”merge until” and ”merge after” decisions.

Hence, our merge-fork analysis determines locally how to handle each merging opportunities within a piece of code. For the simplified language used in our discussion, this process can be implemented by counting occurrences of if-condition and in turn estimating number of queries needed. Besides, unresolved function call and code block difficult to reason can also be counted to influence strategy of merging and forking. When decided to merge two states, path exploration is temporally taken over from searching heuristics and involving states are prioritized to visit.

V. Experimental Evaluation

In this section we illustrate some experimental evaluation on our proposition. We build a prototype to implement our merge-fork techniques (Section V-A), and run it with Coreutils programs to analyze performance in two scenarios: exhaustively exploring all feasible paths (Section V-B) and incomplete exploration within certain time budget (Section V-C). Besides, we also make a comparison with previous work of Kuznetsov et al. [14] (Section V-D).

A. Prototype

To evaluate the effect of our merge-fork framework, we implement active state forking and dynamic merge-fork analysis in a prototype on the foundation of the famous symbolic execution engine KLEE. It takes LLVM bitcode compiled from source code as input. The original executor of KLEE performs path exploration with no approximation, and path feasibility is checked at every conditional branch. State forking is invoked

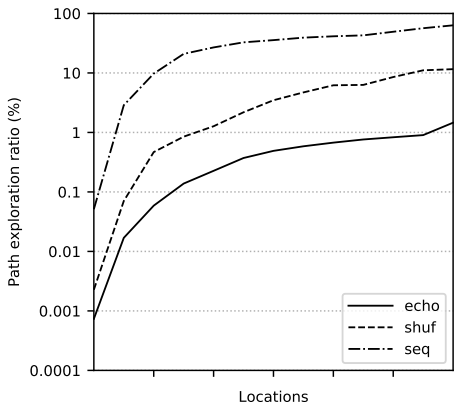

Fig. 3: Ratio of explored paths for 3 representative programs.

for each non-determined branch, and states at the same control location are never merged. KLEE guides exploration with a number of optional search strategies including random search, traditional breadth/depth first search, strategy prioritizing visiting previously unexplored code, and combination of various strategies.

KLEE has built in models of execution state and symbolic execution tree. We extend them to implement our active state forking method. Thanks to implementation of expressions and optimization of constraints in KLEE, it is easy to track updates of path predicates and we could perform state forking by directly imposing conditions of dismissed merging action on path predicates and symbolic store. To support dynamic merge-fork analysis, we make a call in state selection ($selectState$) to an LLVM Pass constructing concrete dependence graph and estimating merging effects locally. If merging is suggested, search strategy is temporally fixed to continuously extending paths to target merging location. Otherwise, states are explored according to original search strategies of KLEE.

The original work of KLEE presents an experiment on GNU Coreutils programs which are widely used UNIX command-line utilities. There are 101 files in total adding up to over 70,000 lines of code. We test these programs using our prototype and collect information from output files.

B. Speedup Path Exploration

Our merge-fork framework is aimed at identifying valuable merging opportunities and conducting state merging, so we are first concerned with the influence of our method on number of explored paths.

A direct comparison is measuring number of explored paths with different time spent. However, the number is hard to figure out since there are paths explored partially if exhaustive exploration is not completed. We therefore choose to calculate the number of paths at a specific location instead of with some time spent during the execution. The size of symbolic inputs is properly set to make it possible for exhaustive exploration, so the accurate number of feasible paths of the program is

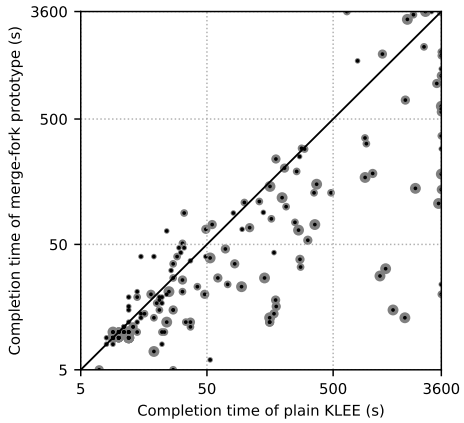

Fig. 4: Compare completion time with plain KLEE for exhaustive exploration tasks.

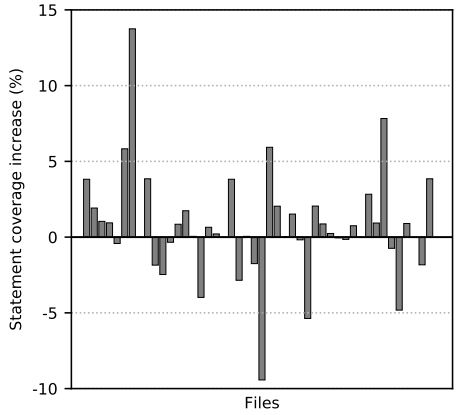

Fig. 5: Compare statement coverage with plain KLEE for uncompleted exploration tasks.

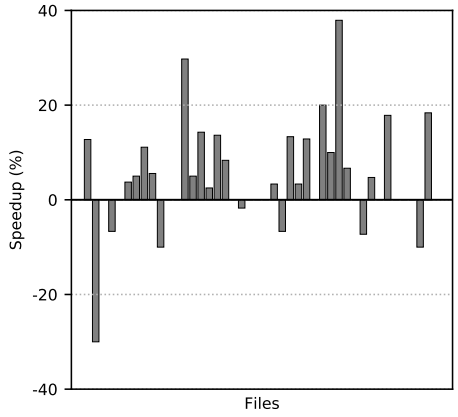

Fig. 6: Speedup over efficient state merging of [14].

known. Then we run our prototype and count paths at given control locations. In this way we can reveal how state merging efficiently joins branches and decrease paths. It is remarkable that paths getting through a certain location might enclose a large gap in time.

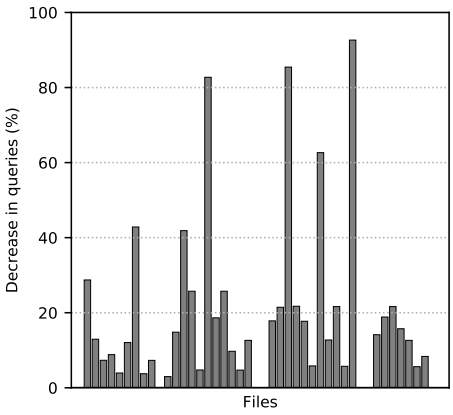

Fig. 7: Decrease in queries comparing with efficient state merging of [14].

Fig.3 shows the growth of path exploration ratio corresponding to program locations for 3 representative files. The final ratio is obtained at the exit of code. It can be seen that the effect in decreasing paths varies among the programs. While a large part of paths are merged in *echo*, state merging makes little influence on *seq*. The trend of growth indicates structure of control flow and involvement of active state forking. Since merging and forking work simultaneously, change in path numbers appear smooth.

The target of decreasing paths is to free underlying solvers' burden and achieve faster execution by omitting duplicated analysis efforts. Hence, we run Coreutils under symbolic inputs with growing size and compare completion time of plain KLEE and our prototype. In order to compare with [14] easily, we draw a similar graph Fig.4 showing the result within 1 hour time limit. The black dots correspond to experiment instances and the gray disks indicate the relative size of the symbolic inputs used in each instance. The result presents that in the majority of cases, our prototype achieves positive speedup over KLEE. The rightmost dots represent those exceed time limit in KLEE but can be solved completely in our framework.

C. Exploration Guided by Statement Coverage

We now look into the situation where tasks can not be finished completely. We choose a proper size of symbolic inputs to keep the engine busy and measure statement coverage in limited time. Plain KLEE uses coverage-oriented search strategy to guide path exploration towards uncovered locations. Our merge-fork framework would interrupt the process and temporally drive exploration towards merging states. Fig.5 shows its influence on statement coverage. While state merging saves analysis efforts, there are more paths can be visited in some cases, which leads to increase in coverage. However, other cases suffer from change in search order and in turn coverage is decreased. Data in Fig.5 confirms that estimation of merging opportunities and fast-forwarding some states do

not cause any obvious troubles to the original exploration goal of searching heuristics.

D. Compare to Existed Work

While our proposition is on the foundation of the previous work of Kuznetsov et al. [14], we make some comparison with their experiment results. All the data is obtained from their official website *http://cloud9.epfl.ch*. Since experimental environment and KLEE version may vary, all the data used in our comparison is increase/decrease over our respective KLEE baseline.

First we compare the completion time under the same size of symbolic inputs. Fig.6 illustrates the result. We use completion time of plain KLEE to normalize data of both tools and then calculate the effect of speedup. Over 80% of the tasks get faster with our merge-fork method and average speedup is around 10%. Most instances with negative speedup are finished in a rather short time (10^1 magnitude in seconds).

We also collect queries generated during the execution. Fig.7 shows the comparison. The number of queries is also normalized against data from plain KLEE. It confirms the effect of active state forking since some queries are omitted on separate paths by concrete execution. The improvement in performance is achieved through decreasing paths via state merging and invoking concrete execution by restoring merged paths when needed.

VI. Conclusions

In symbolic execution, state merging decreases the number of paths but eliminates the benefit of concrete execution in contrast. To resolve the conflict and improve the performance, we proposed a merge-fork framework to enable a bi-directional switch for states to be merged and forked. State merging and active state forking can make state and path combinations fit different code dynamically. This in turn makes it possible to divide code into pieces and dynamically draw decision on state merging piece by piece. Experiment on a prototype trivially displays the potential of our approach. While explosive development in dynamic symbolic execution and search-based techniques has a huge impact on classical state merging methodology, it shows promise combining our approach with these techniques and making state merging an efficient method to solve practical issues in symbolic execution.

References

[1] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on software engineering*, no. 3, pp. 215–222, 1976.

[2] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[3] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[4] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[5] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 61–64.

[6] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang, "Fuzz testing in practice: Obstacles and solutions," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 562–566.

[7] C. Wang, Y. Jiang, X. Zhao, X. Song, M. Gu, and J. Sun, "Weak-assert: a weakness-oriented assertion recommendation toolkit for program analysis," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 69–72.

[8] J. GLiang, M. Wang, Y. Chen, and Y. Jiang, "Pafl: Extend fuzzing optimizations of single mode to industrial parallel mode," 2018.

[9] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," 2018.

[10] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 896–899.

[11] Y. Jiang, H. Song, Y. Yang, H. Liu, M. Gu, Y. Guan, J. Sun, and L. Sha, "Dependable model-driven development of cps: From stateflow simulation to verified implementation," *ACM Transactions on Cyber-Physical Systems*, vol. 3, no. 1, p. 12, 2018.

[12] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[13] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 1066–1071.

[14] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *Acm Sigplan Notices*, vol. 47, no. 6, pp. 193–204, 2012.

[15] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao, "Eliminating path redundancy via postconditioned symbolic execution," *IEEE Transactions on Software Engineering*, 2017.

[16] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. IEEE, 2009, pp. 359–368.

[17] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. IEEE, 2008, pp. 443–446.

[18] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 252–271, 2017.

[19] T. Chen, X.-s. Zhang, S.-z. Guo, H.-y. Li, and Y. Wu, "State of the art: Dynamic symbolic execution for automated test generation," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1758–1773, 2013.

[20] T. Hansen, P. Schachte, and H. Søndergaard, "State joining and splitting for the symbolic execution of binaries," in *International Workshop on Runtime Verification*. Springer, 2009, pp. 76–92.

[21] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. CAV'07, 2007, pp. 519–531.

[22] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[23] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, p. 10, 2008.

[24] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[25] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003, pp. 553–568.

[26] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," *tools and algorithms for construction and analysis of systems*, vol. 2988, pp. 168–176, 2004.

[27] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1083–1094.

A Case Study on Robustness Fault Characteristics for Combinatorial Testing - Results and Challenges

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*—Combinatorial testing is a well-known black-box testing approach. Empirical studies suggest the effectiveness of combinatorial coverage criteria. So far, the research focuses on positive test scenarios. But, robustness is an important characteristic of software systems and testing negative scenarios is crucial. Combinatorial strategies are extended to generate invalid test inputs but the effectiveness of negative test scenarios is yet unclear. Therefore, we conduct a case study and analyze 434 failures reported as bugs of an financial enterprise application. As a result, 51 robustness failures are identified including failures triggered by invalid value combinations and failures triggered by interactions of valid and invalid values. Based on the findings, four challenges for combinatorial robustness testing are derived.**

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

I. Introduction

Combinatorial testing (CT) is a black-box approach to reveal conformance faults between the system under test (SUT) and its specification. An input parameter model (IPM) with input parameters and interesting values is derived from the specification. Test inputs are generated where each input parameter has a value assigned. The generation is usually automated and a *combination strategy* defines how values are selected [1].

CT can help detecting interaction failures, e.g. failures triggered by the interaction of two or more specific values. For instance, a bug report analyzed by Wallace and Kuhn [2] describes that "the ventilator could fail when the altitude adjustment feature was set on 0 meters and the total flow volume was set at a delivery rate of less than 2.2 liter per minute". The failure is triggered by the interaction of `altitude=0` and `delivery-rate<2.2`. This is called a *failure-triggering fault interaction* (FTFI) and its dimension is $d = 2$ because the interaction of two input parameter values is required.

Testing each value only once is not sufficient to detect interaction faults and exhaustively testing all interactions among all input parameters is almost never feasible in practice. Therefore, other combinatorial coverage criteria like t-wise where $1 \leq t < n$ denotes the testing strength are proposed [1].

The effectiveness of combinatorial coverage criteria is also researched in empirical studies [2]–[7]. Collected bug reports are analyzed and FTFI dimensions are determined for different types of software [4]. If all failures of a SUT are triggered by an interaction of d or fewer parameter values, then testing all d-wise parameter value combinations should be as effective as exhaustive testing [4]. No analyzed failure required an interaction of more than six parameter values to be triggered [7], [8]. The results indicate that 2-wise (pairwise) testing should trigger most failures and 4 to 6-wise testing should trigger all failures of a SUT.

However, so far research focuses on *positive* test scenarios, i.e. test inputs with valid values to test the implemented operations based on their specification. Since robustness is an important characteristic of software systems [9], testing of negative test scenarios is crucial. Invalid test inputs contain invalid values, e.g. a string value when a numerical value is expected, or invalid combinations of otherwise valid values, e.g. a begin date which is *after* the end date.

They are used to check proper error-handling to avoid abnormal behavior and system crashes. Error-handling is usually separated from normal program execution. It is triggered by an invalid value or an invalid value combination and all other values of the test input remain untested. Therefore, a strict separation of valid and invalid test inputs is suggested and combination strategies are extended to support generation of invalid test inputs [1], [10]–[13].

But, the effectiveness of negative test scenarios is unclear as this is not yet empirically researched. To the best of our knowledge, it is only Pan et al. [14], [15] who characterize data of faults from robustness testing. Their results obtained from testing robustness of operating system APIs indicate that most robustness failures are caused by single invalid values. Though, there is no more information on failures caused by invalid value combinations. Because only one type of software is analyzed, more empirical studies are required to confirm (or reject) the distribution and upper limit of FTFIs for other software types (Kuhn and Wallace [4]).

To gather more information on failures triggered by invalid value combinations, we conducted a case study to analyze bug reports of a newly developed distributed enterprise application for financial services. In total, 683 bug reports are examined and 434 of them describe failures which are further analyzed.

The paper is structured as follows. Section II and III summarize foundations and related work. In Section IV, the design of the case study is explained. The results are discussed

$$p_1 : PaymentType \quad V_1 = \{CreditCard, Bill\}$$
$$p_2 : DeliveryType \quad V_2 = \{Standard, Express\}$$
$$p_3 : TotalAmount \quad V_3 = \{1, 500\}$$

Listing 1: Exemplary IPM for a Checkout Service

Table I: Pairwise Test Suite

PaymentType	DeliveryType	TotalAmount
Bill	Express	500
Bill	Standard	1
CreditCard	Standard	500
CreditCard	Express	1

in Section V and challenges for combinatorial testing are discussed in Section VII. Afterwards, potential threads to validity are discussed and we conclude with a summary of our work.

II. Background

A. Robustness Testing

Testing is the activity of stimulating a system under test (SUT) and observing its response [16]. System testing (also called functional testing) is concerned with the behavior of the entire system and usually corresponds to business processes, use cases or user stories [17]. Both, the stimulus and response consist of values. They are called test input and test output, respectively. In this context, input comprises anything explicable that is used to change the observable behaviour of the SUT. Output comprises anything explicable that can be observed after test execution.

A test case covers a certain scenario to check whether the SUT satisfies a particular requirement [18]. It consists of a test input and a test oracle [19]. The test input is necessary to induce the desired behavior. The test oracle provides the expected results which can be observed after test execution if and only if the SUT behaves as intended by its specification. Finally, the expected result to the actual result are compared to determine whether the test passes or fails.

Since robustness is an important software quality [9], testing should not only cover positive but also negative scenarios to evaluate a SUT. Robustness is defined as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [18]. Positive scenarios focus on valid intended operations of the SUT using valid test inputs that are within the specified boundaries. Negative scenarios focus on the error-handling using invalid test inputs that are *outside* of the specified boundaries. For instance, input that is malformed, e.g. a string input when numerical input is expected, or input that violates business rules, e.g. a begin date which is *after* the end date.

B. Combinatorial Testing

Combinatorial testing (CT) is a black-box approach to reveal interaction failures, i.e. failures triggered by the interaction of two or more specific values, because the SUT is tested with varying test inputs. A generic test script describes a sequence of steps to exercise the SUT with placeholders (variables) that represent variation points [17]. The variation points can be used to vary different inputs to the system, configuration variables or internal system states [8]. With CT, varying test inputs are created to instantiate the generic test script.

An input parameter model (IPM) is created for which input parameters and interesting values are derived from the specification. The IPM is represented as a set of n input parameters $IPM = \{p_1, ..., p_n\}$ and each input parameter p_i is represented as a non-empty set of values $V_i = \{v_1, ..., v_{m_i}\}$.

Test inputs are composed from the IPM such that every test input contains a value for each input parameter. Formally, a test input is a set of parameter-value pairs for all n distinct parameters and a parameter-value pair (p_i, v_j) denotes a selection of value $v_j \in V_i$ for parameter p_i.

Listing 1 depicts an exemplary IPM to test the checkout service of an e-commerce system with three input parameters and two values for each input parameter. One possible test input for this IPM is `[PaymentType:Bill, DeliveryType:Standard, TotalAmount:1]`. Formally, a test input $\tau = \{(p_{i_1}, v_{j_1}), ..., (p_{i_n}, v_{j_n})\}$ is denoted as a set of pairs. In this paper, we use the aforementioned notation with brackets which is equal to $\tau = \{(p_1, v_2), (p_2, v_1), (p_3, v_1)\}$.

The composition of test inputs is usually automated and a *combination strategy* defines how values are selected [1]. Since testing each value only once is not sufficient to detect interaction faults and exhaustively testing all interactions among all input parameters is almost never feasible in practice, other coverage criteria like t-wise are proposed.

For illustration, Table I depicts a test suite for the e-commerce example that satisfies the pairwise coverage criterion. For more information on the different coverage criteria, please refer to Grindal et al. [1]. To satisfy the coverage criterion, all pairwise value combinations of $PaymentType \times DeliveryType$, $PaymentType \times TotalAmount$ and $DeliveryType \times TotalAmount$ must be included in at least one test input. If the first test input was not executed, pairwise coverage would not be satisfied because the combinations `[PaymentType:Bill, DeliveryType:Express]`,`[PaymentType:Bill, Total Amount:1]`, `[DeliveryType: Express, Total Amount:1]` would be untested.

In comparison to exhaustive testing, fewer test inputs are required to satisfy the other coverage criteria. But as the example illustrates, problems with only one test input might lead to combinations being not covered and failures that are triggered by these combinations remain undetected.

If we suppose that the checkout service requires a total amount of at least 25 dollar, then two test inputs of the example (Table I) with `[TotalAmount:1]` are expected to abort with a message to buy more products. In those cases, the SUT deviates from the normal control-flow and an error-handling procedure is triggered. The value `[TotalAmount:1]` that is responsible for triggering the error-handling is called *invalid value*. If we also suppose that the checkout service rejects payment by bill for total amounts greater than 300 dollar, then `[PaymentType:Bill, TotalAmount:500]` would trigger

error-handling as well. Even though both values are valid, the combination of them denotes an *invalid value combination*.

Valid test inputs do not contain any invalid values and invalid value combinations. In contrast, an invalid test input contains at least one invalid value or invalid value combination. If an invalid test input contains exactly one invalid value or one invalid value combination, it is called a *strong* invalid test input.

Once the SUT evaluates an invalid value or invalid value combination, error-handling is triggered. The normal control-flow is left and all other values and value combinations of the test input remain untested. They are masked by the invalid value or invalid value combination [13]. This phenomenon is called *input masking effect* which we adapt from Yilmaz et al. [20]: "The input masking effect is an effect that prevents a test case from testing all combinations of input values, which the test case is normally expected to test".

To prevent input masking, a strict separation of valid and invalid test inputs is suggested [1], [10]–[13]. Combination strategies are extended to support t-wise generation of invalid test inputs. Values can be marked as invalid to exclude them from valid test inputs and to include them in invalid test inputs. The invalid value is then combined with all $(t-1)$-wise combinations of valid values. An extension that we proposed also allows to explicitly mark and generate t-wise invalid test inputs based on invalid value combinations [13].

C. Fault Characteristics

According to IEEE [18], an error is "the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition." It is the result of a mistake made by a human and is manifested as a fault. In turn, a *fault* is statically present in the source code and is the identified or hypothesized cause of a failure. A *failure* is an external behavior of the SUT, i.e. a behavior observable or perceivable by the user, which is incorrect with regards to the specified or expected behavior.

In CT, we assume that the execution path through the SUT is determined by the values and value combination of the test input. If an executed statement contains a fault that causes an observable failure and if a certain value or a certain value combination is required for executing the statement, then the value or value combination is called a *failure-triggering fault interaction* (FTFI). The number of parameters involved in a FTFI is its *dimension* denoted as d with $0 \leq d \leq n$. For instance, if the checkout service contains a fault and accepts a total amount of one but only if express is chosen as the delivery type, then `[TotalAmount:1, DeliveryType:Express]` is a FTFI with a dimension of two.

In general, different types of triggers exist to expose failures [21]. A *trigger* is a set of conditions to expose a failure if the conditions are satisfied . We focus on FTFI, i.e. failures triggered by test input variations, rather than on failures triggered by ordering or timing of stimulation.

In addition, we introduce the following terms for robustness fault characteristics. A failure is a *robustness failure* if the FTFI contains an invalid value or an invalid value combination. Then, the number of parameters that constitute the invalid value or invalid value combination are denoted as the *robustness size*. In case of an invalid value, the robustness size is one.

The extension to support t-wise generation of invalid test inputs is based on the assumption that failures are triggered by an interaction of an invalid value (or invalid value combination) and a $(t-1)$-wise combination of valid values of the other parameters. This is a robustness interaction and its *robustness interaction dimension* can be computed by subtracting the robustness size from the FTFI dimension. There is no robustness interaction if the robustness size and FTFI dimension are equal, i.e. the robustness interaction dimension is zero. For instance, there is no robustness interaction if `[TotalAmount:1]` or `[PaymentType:Bill, Total Amount:500]` trigger failure. In contrast, the robustness interaction dimension of the aforementioned example `[TotalAmount:1, DeliveryType:Express]` is one because the invalid value interacts with one valid value.

III. Related Work

If the highest dimension of parameters involved in FTFIs is known before testing, then testing all d-wise parameter value combinations should be as effective as exhaustive testing [4]. However, d cannot be determined for a SUT a-priori because the faults are not known before testing. Hence, the motivation of fault characterization in black-box testing is to empirically derive fault characteristics to guide future test activities.

Existing research on the effectiveness of black-box testing derives the distribution and maximum of d among different types of software based on bug reports. Wallace and Kuhn [2] review 15 years of recall data from medical devices, i.e. software written for embedded systems. Kuhn and Reilly [3] analyze bug reports from two large open-source software projects; namely the Apache web server and the Mozilla web browser. Kuhn and Wallace [4] report findings from analyzing 329 bug reports of a large distributed data management system developed at NASA Goddard Space Flight Center. Bell and Vouk [5] analyze the effectiveness of pairwise testing network-centric software. They derive their fault characteristics from a public database of security flaws and create simulations based on that data. Kuhn and Okum [6] apply combinatorial testing with different strengths to a module of a traffic collision avoidance system which is written in the C programming language. Though, the experiments use manually seeded "realistic" faults rather than a specific bug database. Cotroneo et al. [21] again analyze bug reports from Apache and the MySQL database system and Ratliff et al. [22] report the FTFI dimensions of 242 bug reports from the MySQL database system.

As concluded by Kuhn et al. [7], the studies show that most failures in the investigated domains are triggered by single parameter values and parameter value pairs. Progressively fewer failures are triggered by an interaction of three or more parameter values. In addition to the distribution of FTFIs, a maximum interaction of four to six parameter values is

identified. No reported failure required an interaction of more than six parameter values to be triggered. Thus, pairwise testing should trigger most failures and 4- to 6-wise testing should trigger all failures of a SUT [8].

Several tools include the concept of invalid values to support the generation of invalid test inputs [10]–[12]. An algorithm that we proposed [13] extends the concept to invalid value combinations. In their case study, Wojciak and Tzoref-Brill [23] report on system level combinatorial testing that includes testing of negative scenarios. There, t-wise coverage of negative test inputs is required because error-handling depends on a robustness interaction between invalid and valid values. Another case study by Offut and Alluri [24] reports on the first application of CT for financial calculation engines but robustness is not further discussed.

From an empirical point of view, the effectiveness of negative test scenarios is yet unclear. To the best of our knowledge, it is only Pan et al. [14], [15] who characterize data on faults from robustness testing. The results of testing robustness of operating system APIs indicate that most robustness failures are caused by single invalid values. Though, there is no more information on failures triggered by invalid value combinations. Also, as Kuhn et al. [4] state, more empirical studies are required to confirm (or reject) the distribution and upper limit of FTFIs for other software types. Therefore, we conducted another case study which is described in the subsequent sections.

IV. Case Study Design

A. Research Method

We follow the guidelines for conducting and reporting case study research in software engineering as suggested by Runeson and Höst [25]. As they state, a case study "investigates a contemporary phenomenon within its real life context, especially when the boundaries between phenomenon and context are not clearly evident". Case study research is typically used for exploratory purposes, e.g. seeking new insights and generating hypotheses for new research.

The guidelines suggest to conduct a case study in five steps. First, the objectives are defined and the case study is planned. As a second step, the data collection is prepared before the data is collected in a third step. Afterwards, the collected data is analyzed and finally, the results of the analysis are reported.

B. Research Objective

The overall objective of this case study is to gather information on the effectiveness of combinatorial testing with invalid test inputs, and to compare the obtained results with the ones of other published case studies. For example, the work by Pan et al. [14], [15] indicates that most robustness failures in operating system APIs are triggered by single values rather than value combinations, i.e. a FTFI dimension and robustness size of one. Hence, our aim is to either confirm or reject this indication for enterprise applications. This leads to the following two concrete research objectives:

RO1: Identify the different FTFI dimensions of robustness failures that can be observed in our case study.

The generation of t-wise invalid test inputs is based on the assumption that failures are triggered by a robustness interaction between the invalid value (or invalid value combination) and $(t-1)$-wise combination of the valid values of the other parameters. If the highest robustness interaction dimension is known before testing, then testing all t-wise invalid test inputs of that dimension should be as effective as exhaustive robustness testing.

RO2: Identify different robustness sizes and derive the robustness interaction dimensions that can be observed in our case study.

C. Case and Unit of Analysis

The case is a software development project from an IT service provider for an insurance company. A new system is developed to manage the lifecycle of life insurances. It is based on an off-the-shelf framework which is customized and extended to meet the company's requirements. In total, the new system consists of 2.5 MLOC and an estimated workload of 5000 person days. The core is an inventory sub-system with a central database to store information on customer's life insurance contracts. In addition, complex financial calculation engines and business processes like capturing and creating new customer insurances are implemented. The business processes also integrate with a variety of different already existing systems which are, for instance, responsible to manage information about the contract partners, about claims and damages and to the support insurance agents.

Since life insurance contracts have decade-long lifespans and rely on complex financial models, the correctness of the system is business critical. Mistakes can have severe effects which can even amplify over the long-lasting lifespans and cause enormous damage to the company. Therefore, thorough testing is important.

Even though, the business processes are managed by the new system, they rely on other systems of which each again relies on other systems. This makes it hard to test the system or its parts in isolation. It is also difficult to control the state of the systems and to observe the complete behavior which makes testing even more complicated.

Therefore, most testing is conducted on a system level within an integrated test environment in which all required systems are deployed. The test design is often based on experience and error-guessing. Tests are executed mostly manually because of the low controllability and observability.

D. Data Collection Procedure

To yield the research objectives, the case study relies on archival data from the aforementioned software development project. A project-wide issue management system contains all bug reports from the project start in 2015 to the productive deployment at the beginning of 2018. In general, a bug report is a specifically categorized issue which coexists with other project management- and development-related issues.

For our case study, we analyzed the issue's title, its category, its initial description, additional information in the comment section and its status. Further on, some bug reports are also connected to a central source code management system. If a bug report does not contain sufficient information, the corresponding source code modifications can be analyzed as well.

The issues are filtered to restrict the analysis to only reasonable bug reports. Therefore, issues created automatically by static analysis tools are excluded. Further on, only issues categorized as bug reports whose status is set to *complete* are considered because we expect only them to contain a correct description on how to systematically reproduce the failure.

E. Data Analysis Procedure

Once the bug reports are exported from the issue management system, each bug report is analyzed one at a time. First, it is checked if the bug report describes a failure in the sense that an incorrect behaviour is observable by the user. Otherwise, the bug report is rejected.

Afterwards, the trigger type of the reported failure is determined. The bug report is not further analyzed if no systematically reproducible trigger is found. It is also rejected if the failure is not triggered by a test input variation but rather triggered by unlikely ordering or timing. If a specific value or value combination is identified to trigger the failure, the dimension of the FTFI is determined in the next step.

Then, the bug report is classified as either *positive* or *negative* depending on whether any invalid values or invalid value combinations are contained. If it is classified as negative, i.e. if it is a robustness failure, the *robustness size* of the invalid value or invalid value combination is determined as well.

A robustness size which is lower than the FTFI dimension indicates a robustness interaction between the invalid value (combination) and valid combinations of the other parameters. If possible, the robustness interaction dimension is also extracted from the bug report.

V. Results and Discussion

A. Analyzed Data

In total, 683 bug reports are analyzed. All reported bugs are revealed and fixed during the development phase of the system. Even though filters are applied to export the bug reports, 249 bug reports are classified as *unrelated*, because the issue management system is also used as a communication and task management tool. For instance, problems with configurations of test environments, refactorings or build problems are categorized as bug reports as well.

The remaining 434 bug reports describe failures, they are classified as follows. Eight bug reports do not provide enough information for further analysis and classification. 38 reported bugs require specific timing and ordering of sequences to be triggered. For instance, one sequence to trigger a failure is to search for a customer, open its details, edit the birthday, press cancel and edit the birthday again. Three reported bugs are related to robustness testing. They are triggered by other systems that timeout and do not response to requests. All these bug reports are excluded form further analysis because CT is about varying test inputs rather than varying sequences and timing.

The remaining 388 bug reports describe failures triggered by some test input. A subset of 176 bug reports describes integration failures with other systems where values are not correctly mapped from one data structure to another. They can be triggered by *any* test input. There are so many reported integration issues (45% out of 434 bug reports) because the system consists of several independently developed components which are early and often integrated with the other components and other systems using one of the test environments.

Finally, 212 bug reports are considered to be suitable for CT, which is 49% of all 434 bug reports that describe failures which is 55% of all 388 bug reports that are triggered by some test input To reproduce one of the bugs reported, the test input requires at least one specific value.

B. Observed FTFI Dimensions

The observed FTFI dimensions for the 212 bug reports are depicted in Table II. Most failures are triggered by single parameter values and parameter value pairs and progressively fewer failures are triggered by 3- and 4-wise interactions. In our case, no reported bug requires an interaction of more than 4 parameters in order to trigger the failure.

Table III presents the cumulative percentage of the FTFI dimensions. The last three columns refer to our case study and shows that 76% of all reported failures require 1-wise (each choice) coverage to be reliably triggered. It adds up to 96% when testing with pairwise coverage and 100% are covered when all 4-wise parameter value combinations are used for testing.

To compare our results, the first columns of the table show the results of previous case studies, briefly introduced in the related work section. The numbers and also the average percentage values are taken from Kuhn et al. [7]. The distribution of FTFIs obtained in our case study is not in contradiction to the other cases. However, the distribution is mostly similar to cases [2] and [4]. While there are no obvious similarities with embedded systems for medical devices [2], the large data management system [4] is probably quite similar to our case in terms of requirements and used technologies. Similar to our case, the bug reports are also from a development project whereas the other studies analyze fielded products [7]. For all three cases, most failures are triggered by single parameter values and almost all failures are triggered by the combination of single parameter values and pairwise parameter value combinations. All failures should be triggered by 4-wise parameter value combinations.

So far, only the dimension of failure-triggering fault interactions is considered but differences between positive and negative scenarios are not discussed.

All in all, 51 robustness failures are identified which are classified as follows. 22 failures are caused by *incorrect error-detection* of abnormal situations because conditions to detect

Table II: Observed FTFI Dimensions

d	All	Positive	Negative
1	162	121	41
2	40	31	9
3	6	5	1
4	4	4	-
5	-	-	-
6	-	-	-

Table III: Cumulative Percentage of FTFI Dimensions

	Previous Studies								Our Study		
d	[2]	[3]a	[3]b	[4]	[5]	[21]	[22]	Avg.	All	Pos.	Neg.
1	66	28	41	67	18	9	49	39.7	76	75	80
2	97	76	70	93	62	47	86	75.9	96	94	98
3	99	95	89	98	87	75	97	91.4	98	98	100
4	100	97	96	100	97	97	99	98.0	100	100	
5		99	96		100	100	100	99.0			
6		100	100					100.0			

abnormal situations are either wrong or missing. Consequently, these abnormal situations are not discovered. For instance, bank transfer is accepted as a payment option even though incorrect or no bank account information is provided.

In 19 cases, reported failures are caused by *incorrect error-signaling*. Errors are signaled if an abnormal situation is detected but the error should be handled somewhere else. For instance, a misspelled first name is detected by a user registration service but the error message complaints a misspelled last name.

For three reported failures, the abnormal situation is correctly detected and the error is correctly signaled. However, the system performs *incorrect error-recovery* because the instructions to recover from the abnormal situation contain faults. For instance, the user is asked to correct wrong input, e.g. a misspelled first name. After the input is corrected, the system does not recover and the corrected input cannot be processed.

In seven cases, failures are triggered by the system's runtime environment. For instance, a `NullPointerException` is signaled when the runtime environment detects unexpected and illegal access of `NULL` values. Since developers did not expect `NULL` values, no respective error-handlers are implemented and the processes terminate. These failures denote *incorrect flows* from error-signaling to error-recovery.

Table II depicts the observed FTFI dimensions and their distribution divided into positive and negative test scenarios. As can be seen, the maximum dimension of robustness interaction is three. Compared to positive test scenarios, the negative scenarios discover fewer failures and the FTFI dimensions are also lower. For single parameter values and parameter value pairs, the ratio is 3:1 of valid vs invalid test inputs and no invalid test inputs are identified for higher dimensions.

While these numbers indicate that most failures are triggered by valid test inputs, we emphasize that the test design is based on experience and error-guessing, robustness testing was not in the focus. Hence, the ratio can also result from a general bias towards testing of positive scenarios which is identified in research [26]–[28].

Nevertheless, these findings underpin the results of Pan et al. [14], [15] who observe that most robustness failures in operating systems APIs are triggered by single invalid values. In their study, 82% of robustness failures are triggered by single invalid values. We observe the same ratio in our case.

The bug reports also demonstrate the importance of *strong* invalid test inputs, i.e. test inputs with exactly one invalid value or exactly one invalid value combination. For instance, the component that manages contracting parties ensure data quality by checking that, e.g. the title of a person matches the gender of the first name and that the first name and family name are correct and not confused with each other. However, when using an unknown invalid title, the system responds with a wrong error message saying that the family name was wrong. If an invalid family name was combined with the unknown title, the failure would not have been discovered.

To yield the second research objective, the 10 invalid test inputs with a FTFI dimension greater than one are further analyzed. As a result, two reported bugs that describe failures with robustness interactions are discovered. Even though a combination of two and three specific parameter values is required to trigger the robustness failures, the robustness size is only one and two, respectively.

Furthermore, two reported bugs require an interaction of invalid value (combinations) and a valid value of another parameter. One reported bug is related to the communication between two systems. The response of the second system contains one parameter value with error information to indicate whether the requested operation succeeded or failed. Another parameter provides details about the internal processing of the request and a certain value indicates an internal resolution of the error. In that case, the calling system is expected to handle the error in a different way.

Another reported bug belongs to the storage of details on contracting parties where one contracting party must be responsible for paying the insurance premiums. This responsibility is stored as a role called *contributor*. If direct debit is chosen as the payment method but an invalid bank account, i.e. an invalid IBAN number, is provided, the resulting error message remains even after the invalid IBAN is replaced by a valid IBAN. While the combination `[payment-method:direct-debit, account-number:invalid]` is required as an invalid combination, the bug report states that this phenomenon could only be observed for `[role:contributor]`.

VI. Threads to Validity

The biggest thread to validity is that case studies are difficult to generalize from [25]. Especially, because only one particular type of software of one company is analyzed. The archival data of the case study is only a snapshot and the ground truth, i.e. the set of all failures that can be triggered, is unknown. Hence, the data set can be biased, for instance, towards positive scenarios which has been observed in research [26]–[28]. Since the bug reports result from tests based on experience

an error-guessing, it may apply here as well.

The data can also be biased towards certain fault characteristics. Relevant and reasonable bug reports may be excluded by our filtering because the bug reports are incorrectly categorized. Maybe not all triggered failures are reported. For instance, a developer who finds a fault might just fix it without creating a bug report.

VII. Challenges for Combinatorial Testing

One challenge in combinatorial testing is to find an effective coverage criteria. Based on the aforementioned empirical studies, a recommendation for positive scenarios is to use pairwise coverage to trigger most failures and 4- to 6-wise coverage that should trigger all failures. For the application in practice, one major challenge is to generate test suites of minimal or small size with 4- or 6-wise coverage.

To test negative scenarios, different challenges in combinatorial testing can be observed.

In our case study, four classes of incorrect error-handling are identified. First, incorrect error-detection is caused by conditions which are either too strict or too loose. Second, incorrect error-signaling results in a wrong type of error to be signaled. Third, incorrect recovery of a signaled error is caused by a fault in the appropriate recovery instructions. Fourth, incorrect flow from error-signaling to error-recovery is caused by a signaled error for which no appropriate recovery instructions are implemented.

Challenge 1 - Avoid the Input Masking Effect: Incorrect error-detection that is caused by a too strict condition can be revealed by positive test input that mistakenly triggers error-recovery. But, revealing a condition that is too loose requires invalid test input that mistakenly does not trigger error-recovery. To ensure that too strict and too loose conditions can be detected, the generation of valid and invalid test inputs must be separated and both sets of test input must satisfy separate coverage criteria.

Challenge 2 - Generate Strong Invalid Test Inputs: Another challenge is the generation of strong invalid test inputs such that one invalid value or invalid value combination cannot mask another. Incorrect error-detection and incorrect error-recovery may remain undetected if the signal that would result from an incorrect condition is masked by the computation of another invalid value or invalid value combination.

Challenge 3 - Consider Invalid Value Combinations: Since the error-detection conditions may depend on an arbitrary number of input values, it is not sufficient to only consider invalid values as most combinatorial testing tools do. As our case study and Pan et al. [14], [15] show, 80% of the robustness failures are triggered by invalid values, i.e. a robustness size of one, but also 20% of the robustness failures require invalid value combinations to be triggered. Error-detection with more complex conditions must be tested as well. Invalid value combinations should be excluded when generating positive test inputs but included when generating invalid test inputs [13]. Therefore, appropriate modeling facilities and algorithms that consider invalid value combinations are another challenge.

Challenge 4 - Support Alternative Coverage Criteria: To reveal incorrect handling and incorrect recovery, the SUT must be stimulated by the failure-triggering invalid test input. The majority of analyzed robustness failures does not indicate any robustness interaction between valid values and invalid values or invalid value combinations. Then, the failure is triggered by the invalid value or invalid value combination. To satisfy the coverage criterion, it is sufficient to have a separate test input for each invalid value or invalid value combination.

However, robustness failures where invalid values or invalid value combinations interact with valid values and valid value combinations could also be observed. The failure is triggered by a t-wise interation of one or more valid values and the invalid value or invalid value combination. For instance, suppose the valid `role=contributor` is responsible for selecting the strategy which is used to process the bank data of a customer. If the invalid combination of `[payment-method:direct-debit]` and `[account-number:invalid]` is handled incorrectly by the selected strategy, then the interaction of all three values is required to trigger the failure.

The observed failures of our case study are in line with a case study by Wojciak and Tzorref-Brill [23] who faced error-handling that would be different depending on firmware in control and system configurations. Different configuration options can also be modelled as input parameters, a robustness interaction of configuration options with invalid values and valid value combinations is also reasonable.

Since only low dimensions of robustness interaction are observed, we believe it is unlikely that the generation of 4- to 6-wise test suites is a challenge here as well. Instead, alternative coverage criteria that, for instance, allow a variable strength interaction with some other input parameters can become a relevant to reduce the number of test inputs.

VIII. Conclusion

The effectiveness of negative test scenarios is unclear from an empirical point of view. We conducted a case study to get information on failures triggered by invalid test inputs. The motivation for our and others studies was that if all failures are triggered by an interaction of d or fewer parameter values, then testing all d-wise parameter value combinations should be as effective as exhaustive testing [4].

In our case study we analyzed bug reports which originate from a development project that manages life insurances. In total, 683 bug reports are analyzed. 434 bug reports describe actual failures and 212 of them are failures triggered by a 2-wise or higher interaction of parameter values.

In general, the distribution of FTFI dimensions conforms to the pattern of previous empirical studies. But in contrast to positive test scenarios, fewer robustness failures with lower FTFI dimensions are identified. Overall, the robustness failures are grouped in four classes: incorrect error-detection, incorrect error-signaling, incorrect recovery from a signaled error and incorrect flow from error-signaling to error-recovery. Most robustness failures (80%) are triggered by single invalid values.

The remaining robustness failures require an interaction of two and three input parameter values. Two reported bugs require an interaction of valid values with an invalid value or invalid value combinations to trigger the robustness failure.

Based on the findings of this case study, we derive challenges for combinatorial robustness testing. To ensure that failures do not remain hidden, possible masking should be reduced. Valid and invalid test inputs should be separated and invalid test inputs should be strong, i.e. should only contain one invalid value or invalid value combination.

Further on, it is not sufficient to only consider invalid values as most combinatorial testing tools do. Invalid value combinations should be excluded when generating valid test inputs but considered for invalid test inputs. Therefore, appropriate modeling facilities and algorithms are required.

Since only low robustness interactions are observed, the generation of test inputs with 4- to 6-wise coverage is not that important for negative scenarios. But, the support of variable strength generation for invalid inputs is another challenge.

Most robustness failures do not involve any robustness interaction. But, there are situations where robustness interactions can be observed since different input values, configuration options or internal states are modelled as input parameter values. Depending on expected costs of failure, t-wise testing of invalid test inputs is an option.

In the future, we will work on facilities that support the modelling of invalid value combinations and we will integrate variable strength in a combinatorial algorithm for invalid input generation. To reduce the number of invalid test inputs, we will conduct experiments to investigate the efficiency of different coverage criteria.

References

[1] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: A survey," *Software Testing, Verification and Reliability*, vol. 15, no. 3, 2005.

[2] 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.

[3] 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.

[4] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, June 2004.

[5] K. Z. Bell and M. A. Vouk, "On effectiveness of pairwise methodology for testing network-centric software," in *Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on*. IEEE, 2005, pp. 221–235.

[6] D. R. Kuhn and V. Okum, "Pseudo-exhaustive testing for software," in *2006 30th Annual IEEE/NASA Software Engineering Workshop*, April 2006, pp. 153–158.

[7] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Estimating t-way fault profile evolution during testing," in *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, vol. 2. IEEE, 2016, pp. 596–597.

[8] R. Tzoref-Brill, "Advances in combinatorial testing," ser. Advances in Computers. Elsevier, 2018.

[9] M. M. Hassan, W. Afzal, M. Blom, B. Lindstrom, S. F. Andler, and S. Eldh, "Testability and software robustness: A systematic literature review," in *2015 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2015.

[10] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, 1997.

[11] J. Czerwonka, "Pairwise testing in real world," in *24th Pacific Northwest Software Quality Conference*, 2006.

[12] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Acts: A combinatorial test generation tool," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 370–375.

[13] K. Fögen and H. Lichter, "Combinatorial testing with constraints for negative test cases," in *2018 IEEE Eleventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 7th International Workshop on Combinatorial Testing (IWCT)*, 2018.

[14] J. Pan, "The dimensionality of failures - a fault model for characterizing software robustness," *Proc. FTCS '99, June*, 1999.

[15] J. Pan, P. Koopman, and D. Siewiorek, "A dimensionality model approach to testing and improving software robustness," in *AUTOTESTCON'99. IEEE Systems Readiness Technology Conference, 1999. IEEE*. IEEE, 1999, pp. 493–501.

[16] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, 2015.

[17] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2007.

[18] IEEE, "Ieee standard glossary of software engineering terminology," *IEEE Std*, vol. 610.12-1990, 1990.

[19] N. Li and J. Offutt, "Test oracle strategies for model-based testing," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 372–395, April 2017.

[20] C. Yilmaz, E. Dumlu, M. B. Cohen, and A. Porter, "Reducing masking effects in combinatorial interaction testing: A feedback driven adaptive approach," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, 2014.

[21] D. Cotroneo, R. Pietrantuono, S. Russo, and K. Trivedi, "How do bugs surface? a comprehensive study on the characteristics of software bugs manifestation," *Journal of Systems and Software*, vol. 113, pp. 27 – 43, 2016.

[22] Z. B. Ratliff, D. R. Kuhn, R. N. Kacker, Y. Lei, and K. S. Trivedi, "The relationship between software bug type and number of factors involved in failures," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct 2016, pp. 119–124.

[23] P. Wojciak and R. Tzoref-Brill, "System level combinatorial testing in practice - The concurrent maintenance case study," *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014*, 2014.

[24] J. Offutt and C. Alluri, "An industrial study of applying input space partitioning to test financial calculation engines," *Empirical Software Engineering*, vol. 19, no. 3, pp. 558–581, Jun 2014.

[25] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, p. 131, Dec 2008.

[26] L. M. Leventhal, B. M. Teasley, D. S. Rohlman, and K. Instone, "Positive test bias in software testing among professionals: A review," in *Human-Computer Interaction*, L. J. Bass, J. Gornostaev, and C. Unger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 210–218.

[27] B. E. Teasley, L. M. Leventhal, C. R. Mynatt, and D. S. Rohlman, "Why software testing is sometimes ineffective: Two applied studies of positive test strategy." *Journal of Applied Psychology*, vol. 79, no. 1, p. 142, 1994.

[28] A. Causevic, R. Shukla, S. Punnekkat, and D. Sundmark, "Effects of negative testing on tdd: An industrial experiment," in *Agile Processes in Software Engineering and Extreme Programming*, H. Baumeister and B. Weber, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 91–105.

What do we know about software security evaluation? A preliminary study

Séverine Sentilles
School of Innovation, Design and Engineering, Mälardalen University
Västerås, Sweden
severine.sentilles@mdh.se

Efi Papatheocharous
ICT SICS
RISE Research Institutes of Sweden
Stockholm, Sweden
efi.papatheocharous@ri.se

Federico Ciccozzi
School of Innovation, Design and Engineering, Mälardalen University
Västerås, Sweden
federico.ciccozzi@mdh.se

***Abstract*—In software development, software quality is nowadays acknowledged to be as important as software functionality and there exists an extensive body-of-knowledge on the topic. Yet, software quality is still marginalized in practice: there is no consensus on what software quality exactly is, how it is achieved and evaluated. This work investigates the state-of-the-art of software quality by focusing on the description of evaluation methods for a subset of software qualities, namely those related to software security. The main finding of this paper is the lack of information regarding fundamental aspects that ought to be specified in an evaluation method description. This work follows up the authors' previous work on the Property Model Ontology by carrying out a systematic investigation of the state-of-the-art on evaluation methods for software security. Results show that only 25% of the papers studied provide enough information on the security evaluation methods they use in their validation processes, whereas the rest of the papers lack important information about various aspects of the methods (e.g., benchmarking and comparison to other properties, parameters, applicability criteria, assumptions and available implementations). This is a major hinder to their further use.**

***Index Terms*—Software security, software quality evaluation, systematic review, property model ontology.**

I. Introduction

Software quality measurement quantifies to what extent a software complies with or conforms to a specific set of desired requirements or specifications. Typically, these are classified as: (i) functional requirements, pertaining to what the software delivers, and (ii) non-functional requirements, reflecting how well it performs according to the specifications. While there exists a vast plethora of options for functional requirements, non-functional requirements have been studied extensively and classified through standards and models (i.e., ISO/IEC 9126 [1], ISO/IEC 25010 [2] and ISO/IEC 25000 [3]). They are often referred to as extra-functional properties, non-functional properties, quality properties, quality of service, product quality or simply metrics.

Despite the classification and terminology accompanying these properties, their evaluation and the extent of how well a software system performs under specific circumstances are both hard to quantify without substantial knowledge on the particular context, concurring measurements and measurement methods. In practice, quality assessment is still marginalized – there is no consensus on what software quality exactly is, how it is achieved and evaluated.

In our research we investigate to what extent properties and their evaluation methods are explicitly defined in the existing literature. We are interested in properties such as reliability, efficiency, security, and maintainability. In our previous work [4], we investigated which properties are most often mentioned in literature and how they are defined in a safety-critical context. We found that the most prominent properties of cost, performance and reliability/safety were used, albeit not always well-defined by the authors: divergent or non existent definitions of the properties were commonly observed.

The target of this work is to investigate evaluation methods related to software security, which is a critical extra-functional property directly and significantly impacting different development stages (i.e., requirements, design, implementation and testing). Security is defined by McGraw [5] as "engineering software so that it continues to function correctly under malicious attack". The high diversity of components and complexity of current systems makes it almost impossible to identify, assess and address all possible attacks and aspects of security vulnerabilities.

More specifically, in this paper we want to answer the following research question: *"What percentage of evaluation method descriptions contains pertinent information to facilitate its use?"*. For this, we identified a set of papers representing the state-of-the-art from the literature in the field and we assessed the degree of explicit information about evaluation methods. The set of papers was analyzed to identify definition and knowledge representation issues with respect to security, and answer the following three sub-questions:

- RQ1: What proportion of properties is explicitly defined?
- RQ2: What proportion of evaluation methods provide explicit key information to enable their use (e.g., formula, description)?
- RQ3: What proportion of other supporting elements of the evaluation methods is explicit (e.g., assumptions, available implementations)?

We used a systematic mapping methodology [6], started from 63 papers and selected 24 papers, which we then analyzed further. After thorough analysis, we excluded eight papers, resulting in a final set of 16 papers, which we used for

synthesizing and reporting our results. The final set of papers is listed in Table V. The main contribution of this paper is represented by the identification of the following aspects:

- relation between evaluation methods and a generic property definition (i.e., if it exists explicitly and how it is described),
- which aspects of other supporting elements to improve understandability and applicability of the method (i.e, assumptions, applicability, output etc.) are explicitly mentioned, and
- the missing information regarding fundamental aspects which ought to be specified in a property and method description.

The remainder of the paper is structured as follows. In Section II we introduce the related work in the topic, while in Section III we describe our methodology. Section IV summarizes quantitative and qualitative interpretations of the extracted data, while Section V reports the main results. Section VI discusses the threats to validity and Section VII concludes the paper and delineates possible future directions.

II. Related work

An extensive body-of-knowledge already exists on software security and many studies have already investigated security assessment as well. For example, the U.S. National Institute of Information Standards and Technology (NIST) in [7] developed a performance measurement guide for information security and in particular how an organization through the use of metrics can identify the adequacy of controls, policies and procedures. The approach is mostly focused on the level of technical security controls in the organization rather than the technical security level of specific products.

In [8], a taxonomy for information security-oriented metrics is proposed in alignment to common business goals (e.g., cost-benefit analysis, business collaboration, risk analysis, information security management and security dependability and trust for ICT products, systems and services) covering both organizational and product levels.

Verendel [9] analyzed quantitatively information security from a set of articles published from 1981-2008 and concluded that theoretical methods are difficult to apply in practice and there is limited experimental repeatability. A recent literature survey [10] carried out on ontologies and taxonomies of security assessment identified among others a gap in works addressing research issues like knowledge reuse, automatic processes, increasing the assessment coverage, defining security standards and measuring security.

Morrison et al. [11] carried out a systematic mapping study on software security metrics to create a catalogue of metrics, their subject of measurement, validation methods and mappings based on the software development life cycle (SDLC). Based on the vast catalogue of metrics and definitions collected, a major problem is that they are extremely hard to be compared, as they typically measure different aspects of the property (something obvious from the emergent categories proposed in the paper). Thus, there is no agreement on their degree of assessment coverage or how to achieve their evaluation. Moreover, in their work, the generic property of security, including its definition, relation to a reference and detailed explanation on how to achieve assessment, is not addressed.

Therefore, we decided to investigate thoroughly the literature and quantify the quality of the description of security evaluation methods. Our approach differs from the work of Morrison et al. [11] in that we are not interested to collect metrics and their mapping to SDLC. Instead, we examine evidence on properties' explicit definitions, their evaluation methods and parameters (including their explanations on how they are used) for assessing security. We develop a mapping from an applicability perspective of the evaluations, which may be used by researchers and practitioners at different phases of security evaluations, i.e., at the low-level internal or at the external (often called operational) phase of software engineering.

III. Methodology

We base our work on the mapping study previously conducted by Morrison et al. in [11]. The reasoning is that the basis of the work is well aligned and applicable to answering our RQs. The details of the methodology are explained in the following.

A. Search and selection strategy

The data sources used in [11] are the online databases, conference proceedings and academic journals of ACM Digital Library, IEEE Xplore and Elsevier. The search terms include the keywords: "software", "security", "measure", "metric" and "validation". Considering another 31 synonyms, the following search string was used:
"`(security OR vulnerability) AND (metric OR measure OR measurement OR indicator OR attribute OR property)`",
in which the terms are successively replaced with the identified synonyms.

The selection criteria of Morrison et al. include the sets of inclusion and exclusion items listed below.

Inclusion criteria:

- Paper is primarily related to measuring software security in the software development process and/or its artifacts. For example software artifacts (e.g., source code files, binaries), software process (e.g., requirements phase, design, implementation, testing), and/or software process artifacts (e.g., design and functional specifications).
- Measurements and/or metrics are the main paper subject
- Refereed paper
- Paper published since 2000

Exclusion criteria:

- Related to sensors
- Related to identity, anonymity, privacy
- Related to forgery and/or biometrics
- Related to network security (or vehicles)

- Related to encryption
- Related to database security
- Related to imagery, audio, or video
- Specific to single programming languages

As explained above, search string, search strategy, inclusion and exclusion criteria defined by Morrison et al. are applicable to our work. Thus, we reuse the list of selected papers from their mapping study as the initial set for our study. However, as we wanted to investigate the applicability and quality of the security evaluation methods found, we extended the selection and data extraction methodology to isolate the subset of selected papers that is considered most relevant. In order to identify the subset, we performed the following actions:

- Define new exclusion criteria
- Design a new selection process
- Define a new data collection scheme
- Extract the data
- Intepret data and synthesize results

B. New exclusion criteria

We decided to exclude sources according to the following criteria:

- Full-text is not available
- Sources are Ph.D. dissertations or books
- Sources are not industrially validated or do not use well-known software, tool or platforms for their validations
- Sources are model predictions that do not assess any property

C. New selection process

The selection is performed on the list of 63 selected papers from [11] by three independent researchers by looking at the title, abstract and by going through a quick reading of the content. Papers independently selected by all researchers are included. Similarly, papers which are discarded by all researchers are excluded. For papers for which there is a disagreement, a discussion is held between the involved researchers to unanimously decide whether to include or exclude the papers. In one case (p18), an extended version of the paper was found and used instead (i.e. we used [12] instead of [13]). This selection process resulted in 24 included papers and 39 rejected. Furthermore, 8 papers were later excluded during the data extraction as they did not contain useful information or were not of sufficient quality. Hence, our selection set is based on the 16 papers listed in Table V.

D. Data collection scheme

As support for answering the RQs, we created a form based on excel spreadsheets. The form consists of the questions specified in Table I, with the list of possible values for each question. The questions are derived from our previous work on the Property Model Ontology (PMO), which formally specifies which concepts should be described for properties and their evaluation methods and how these concepts relate to one another. For details on the PMO, the reader is directed to [4].

E. Data extraction and interpretation

The data extraction is carried out by the researchers using independent spreadsheet columns (one for each paper). The data extraction for each paper is reviewed by another researcher (i.e. reviewer) and discussions are carried out until an agreement is reached. The data is then interpreted and analyzed by all researchers together, as explained in the next section.

IV. Data interpretation

A. Quantitative analysis

One interesting, albeit not very surprising fact, is that most of the descriptions contained in the papers explicitly state what is supposed to be calculated (i.e., the property and the output) and how (i.e., through the overall approach description, the parameters involved in the approach, and to some extent the advantages and disadvantages). As visible in Table II, which shows the results of the quantitative analysis, the property is explicitly referred to in 87,5% of the papers, the output in 75%, the approach description in 87,5%, the parameters in 81%, the advantages in 62% and the disadvantages in 69%.

No method explicitly specifies the unit to be used for the output. This might be explained by the fact that software security is a rather recent research field and there is currently no widely acknowledged metric and corresponding units to be used. As a result, most methods typically fall back on standard data types for which units are less relevant (e.g., percentage, probability, item counts).

Applicability criteria are rarely explicitly mentioned, only in 19% of the papers. When they are mentioned, it is not obvious whether the set of provided criteria is complete or at least sufficient to ensure the applicability of the evaluation method.

Only half of the papers clearly specify the assumptions and hypotheses that are assumed to hold when using the method. This is a setback as these aspects are important for correctly applying a method.

Drivers are rarely explicitly mentioned; only in 12.5% of the papers. Drivers can be important as they can implicitly affect the output of a method.

Despite being often mentioned in the papers as part of the solution being implemented, in practice, only a few implementations or programs are available to directly support the method evaluation. We could find mentions of available tool support in only 2 out of the 16 papers (i.e., WEKA toolkit and Fortify Source Code Analyzer (SCA) version 5.10.0.0102 in p43 and p47). A few additional papers (4) refer to available tool support but no explicit link or reference is provided.

Out of the 16 analyzed papers, only 1 (i.e., p60) performed some kind of benchmarking or comparison to similar properties or evaluation methods. This corresponds to only 6% of all the papers.

To a large extent, the information provided in the papers is insufficient to directly apply the method, especially by non-experts (in 12 papers, 75%).

TABLE I: Data collection form

	Data to collect	**Possible Values**
q1	PaperId	Px, with x a number
q2	Method name	Free text
q3	Does the evaluation method provide an explicit reference to a property or a metric	Yes | no
q4	If yes on q3, how?	Name + def. + ref. | name + def. | name + ref. | other
q5	If other on q4, how?	Free text
q6	If yes on q3, which one? Give the example	Free text
q7	If yes on q3, does the reference match what is needed for the evaluation method?	Yes | no | I don't know
q8	Does the evaluation method explicitly state the output of the method	Yes | no
q9	If yes on q8, how?	With data format and unit | With data only | other
q10	If other on q9, how?	Free text
q11	If yes on q8, which one? Give the example	Free text
q14	Does the evaluation method explicitly state applicability criteria?	Yes | no
q15	If yes, which ones?	Free text
q16	Does the evaluation method explicitly state how the property is evaluated?	Yes | no | I don't know
q17	If yes on q16, how?	Free text
q18	Does the evaluation method explicitly explain the parameters involved in the evaluation?	Yes | no
q19	If yes on q19, which parameters (with their explanations)?	Free text
q25	Does the evaluation method explicitly describe additional drivers that might affect the evaluation?	Yes | no
q26	If yes on q25, which ones?	Free text
q27	Does the evaluation method explicitly state the assumptions, hypotheses which the method is based on?	Yes | no
q28	If yes on q27, which ones?	Free text
q29	Does the evaluation method explicitly mentioned advantages for the method?	Yes | no
q30	If yes on q30, which ones?	Free text
q31	Does the evaluation method explicitly mentioned disadvantages for the method	Yes | no
q32	If yes on q31, which ones?	Free text
q33	Does the evaluation method explicitly reference an implementation or a program that can be used?	Yes | no
q34	If yes on q33, which ones?	Free text
q35	If yes on q33, is the implementation or program accessible	Yes | no
q36	Is there any comparison to other properties (e.g., benchmark, validation)	Yes | no
q37	If yes on q36, how?	Free text
q38	Additional comments	Free text
q39	Is the information provided in the paper sufficient to understand the method and directly use it?	Yes | no
q40	Extractor's name	Séverine | Efi | Federico
q41	Reviewer's name	Séverine | Efi | Federico

B. Qualitative analysis

From analyzing the answers to the questions in Table II, papers can be categorized into three groups based on their main purpose. The first group focuses on defining a new property or metric to assess some specific software security aspects (p1, p18, p27, p47, p63). The papers belonging to the second group (p15, p25, p37, p51, p60) base their work on already defined properties. Their main objective is to either find ways to combine existing metrics to evaluate a given security aspect, define a new evaluation method for existing properties or validate previously specified methods in applying them on a specific system. The last group aims at finding correlation between already defined properties or performing predictions to evaluate the accuracy of previously defined models (p38, p43, p53, p55, p59). There are papers that do not belong to these groups as they are not explicitly referring to any property, method or metric (p22, p50). This is shown in the answers to questions on whether the paper explicitly refers to a property and, if yes, how (i.e., q3 and q4). Papers belonging to the first group use "definition without reference", papers from the second group use "definition plus reference" and papers from the last group use "other".

From the data extracted, 34 definitions and references on security are collected from the papers, as listed in Table IV. We categorized the collected properties in two different levels, depending on their level of specificity (less coarse-grained). A few papers shared some definitions (e.g., p25, p59, p63) whereas most papers defined their own security aspect to evaluate. The security aspect expressed (even if in some cases is explained with many details) is only able to capture some facets of the property, thus it is hard to know if the property definitions together with their evaluations are enough to asses security in a rational way.

Related to the descriptions of the properties and evaluations extracted, we captured the level of applicability between them, as they were described in the papers: most commonly the deployment and operation phase (post-release), the implementation (development or code level, including testing) and the design phase as shown in Table III. The level at which security evaluation is carried out varies among the papers. 7 of them (44%) apply the evaluation method at the implementation (code-level) phase, 1 paper at the testing phase, 6 papers (37,5%) at the operational phase, 1 paper during maintenance and 4 papers (25%) at the design level.

TABLE II: Quantitative results

Question	**Count of 'Yes' (%)**	**Count of 'No' (%)**
q3: Explicit property	14 (87,5%)	2 (12,5%)
q8: Explicit method output	12 (75%)	4 (25%)
q14: Explicit applicability criteria	3 (19%)	13 (81%)
q16: Explicit description	14 (87,5%)	2 (12,5%)
q18: Explicit method parameters	13 (81%)	3 (19%)
q25: Explicit drivers	2 (12,5%)	14 (87,5%)
q27: Explicit assumptions	8 (50%)	8 (50%)
q29: Explicit advantages	10 (62,5%)	6 (37,5%)
q31: Explicit disadvantages	11 (69%)	5 (31%)
q33: Explicit tool reference	6 (37,5%)	10 (62,5%)
q35: Tool accessibility	2 (12,5%)	4 (25%)
q36: Benchmark	1 (6%)	15 (94%)
q39: Information is sufficient	4 (25%)	12 (75%)

TABLE III: Security level applicability

pID	**Applicability level**	**Analysis level***
p1	system architecture - design - implementation	internal
p15	implementation	internal
p18	operational	external
p22	operational - system architecture - design	int. & ext.
p25	operational	int. & ext.
p27	operational	external
p37	implementation	internal
p38	design	internal
p43	design - implementation	internal
p47	operational	external
p50	operational	external
p51	implementation	internal
p53	maintenance	external
p59	testing	external
p60	detailed design - implementation	internal
p63	implementation	internal

*Based on the definitions from [1] on internal and external metrics for product quality.

As mentioned above, no units are explicitly specified. However, the output of the evaluation methods falls back onto implicit scales of measurements: nominal (p60), ordinal (p1, p18, p37, p47, p50, p60, p63), interval (p53) and ratio (p18, p25, p43, p60).

Regarding advantages of the methods, the most commonly reported are: reliability of the method (p1, p15, p25), simplicity of the method (p1, p18, p25), accuracy of the method (p18, p43) and objectivity of the results (p47, p60). The disadvantages mostly refer to the accuracy of the results being dependent on the quality of the available data (p18, p25, p37, p43) and subjectivity involved in the method (p22, p37, p63).

V. Results and discussions

As an answer to our initial research question "What percentage of evaluation methods description contains pertinent information to facilitate its use?", only 25% of the papers that we investigated were judged to provide enough information to enable a direct application of the method. Overall, the papers were good at explicitly stating the property under study, describing how the property is evaluated and which parameters are involved. On the other hand, in several cases the papers lacked information regarding the unit representing the value of the property, the applicability criteria for the method as well as its assumptions, possible advantages and disadvantages, eventual openly available tool support, and comparison to other properties.

Security being a relatively new area in software engineering can be a major contributing factor to these results. For example, advantages, disadvantages, and applicability criteria are difficult to identify initially. They require time and perspective on the topic as well as the methods having been used over time in a wide range of applications and scenarios. Given that many of the papers which were included in our set focused on defining new properties for security, it is not so surprising that so few papers mentioned those aspects. Furthermore, following the same reasoning, even if those criteria were explicitly stated in the papers, it is not certain that the proposed lists of advantages, disadvantages and applicability is exhaustive, or at least sufficient to guarantee the proper usage of the evaluation method that they introduce.

When comparing to the results in [11], we found fewer metrics and definitions. This is due to the fact that our purpose was to identify the main property (or properties) of the paper and their corresponding evaluation methods. Some of the metrics identified in [11] are classified in our results as parameters. Others have been ignored as they were used for comparison purposes and therefore not as relevant for our work. However, the conclusions by Morrison et al. still hold and are further supported by our results. Security properties are not mature, most of them have been proposed and evaluated solely by their authors, and few comparisons between already defined properties exist. "Despite the abundance of metrics found in the literature, those available give us an incomplete, disjointed, hazy view of software security."

These results are to be put in perspective, since none of the authors of this paper is an expert in software security and due to the small number of papers that were studied in this work. However, we are confident that the observations resulting from this study are good indications of the issues occurring with property and evaluation methods descriptions in software security.

A bi-product of our analysis is the following interesting aspect. Especially for the works assessing software vulnerability, evaluation methods exploit well-established prediction models that leverage a discernible set of software metrics (related to the code itself, developers activity, versioning data, etc.). A (what seems to be) common step to the definition of security-related evaluation methods, especially when dealing with vulnerability, is the comparison of existing prediction models, or the comparison of a newly defined prediction model with a set of existing ones, in order to identify the "best" model

TABLE IV: Collection of properties or coarse-grained metrics explicitly defined in the papers[1]

pID	Generic/coarse-grained definition	Less generic definition
p1	Security to mean control over data confidentiality [...] and data integrity [...][1]	Total Security Index (TSI) as the sum of the security design principle metrics
p15	Vulnerability as a weakness in a software system that allows an attacker to use the system for a malicious purpose	
p18	End-user software vulnerability exposure as a combination of lifespans and vulnerability announcement rates	Median Active Vulnerabilities (MAV): the median number of software vulnerabilities which are known to the vendor of a particular piece of software but for which no patch has been publicly released by the vendor Vulnerability Free Days (VFD): captures the probability that a given day has exactly zero active vulnerabilities
p25	Vulnerability (defined in [14])	
p27	Operational security as representation of as accurately as possible the security of the system in operation, i.e., its ability to resist possible attacks or, equivalently, the difficulty for an attacker to exploit the vulnerabilities present in the system and defeat the security objectives	Mean Effort To security Failure (METF): Mean effort for a potential attacker to reach the specified target
p37	Security through an attack surface metric	Attack surface metric a measure of a systems attack surface along three dimensions by estimating the total contribution of the methods, the total contribution of the channels, and the total contribution of the data items to the systems attack surface [...][1]
p38	Security as through the number of violations of the least privilege principle and surface metrics Maintainability as coupling between components and components instability	LP principle metric (defined in [15]) Attack surface metric (defined in [16]) Coupling between components (CBM) (defined in [17]) Components instability (CI) (defined in [18])
p43	Vulnerability through dependency graphs (*)	
p47	Vulnerability through static analysis (*)	Static-analysis vulnerability density (SAVD): number of vulnerabilities a static-analysis tool finds per thousand LOC Density of post-release reported vulnerability (NVD) (*) Application vulnerability rank (SAVI) (*)
p50	Vulnerability as flaws or weakness in a systems design, implementation, or operation and management that could be exploited to violate the systems security policy. Any flaw or weakness in an information system could be exploited to gain unauthorized access to, damage or compromise the information system.	
p51	Application security (defined in [5]) Software security (defined in [5]) Security at program level (defined in [19])	Stall Ratio (SR): a measure of how much a programs progress is impeded by frivolous activities. Coupling Corruption Propagation (CCP): Number of child methods invoked with the parameter(s) based on the parameter(s) of the original invocation Critical Element Ratio (CER) (*)
p53	Security through vulnerability density (*)	Static analysis vulnerability density (SAVD) (see p47) Security Resources Indicator (SRI): the sum of four indicator items, ranging from 0 to 4. The items are: the documentation of the security implications of configuring and installing the application, a dedicated e-mail alias to report security problems, a list or database of security vulnerabilities specific to the application, and the documentation of secure development practices, such as coding standards or techniques to avoid common secure programming errors.
p59	Vulnerability (defined in [14])	
p60	Software vulnerability (defined in [19])	Structural severity: uses software attributes to evaluate the risk of an attacker reaching a vulnerability location from attack surface entry points [...][1] Attack Surface Entry Points (defined in [20]) Reachability Analysis (*)
p63	Software vulnerability (defined in [14])	Vulnerability-Contributing Commits (VCCs): original coding mistakes or commits in the version control repository that contributed to the introduction of a post-release vulnerability

[1]The definitions have been shorten to comply with the page limitation. Readers are referred to the original paper for the complete definition.
(*) No explicit definition is found.

to use as basis for evaluation purposes. Another interesting aspect is represented by the fact that, in several papers, authors exercise their reasoning and methods on well-known codebases (e.g., Windows Vista, Mozilla); this shows an interesting strong inclination towards assessing the applicability of research (theoretical) results to practical cases, which is too seldom seen in other research branches within software engineering.

VI. Threats to validity

Construct validity relates to what extent the phenomenon under study represents what the researchers wanted to investigate and what is specified by the research questions. We explicitly defined the context of the work and discussed related terms and concepts. Also, the research questions were formulated based on these clarified notions and the research followed a methodological procedure known as systematic mapping.

The selection of papers was based on the work of Morrison et al. [11], thus we inherit the work's limitations. Therefore, papers not listed in the sources used, i.e., ACM, IEEE and Elsevier, have been missed. In addition, we excluded Ph.D. dissertations and books since we limited our selection to peer-reviewed conference and scholar journal publications only.

The way the involved researchers individually interpreted the methods described in the papers reflects their own biases and views. However, we worked hard to reduce the bias by discussing the content of the papers in pairs to dissolve uncertainties. In several cases, where the decision to include or not a paper, a third research was involved to reach a consensus.

Morrison at al. [11] excluded sources dealing with networks, sensors, database and vehicle security and this limited the set of papers analyzed in our study. This opens however up for opportunities of further research targeting these particular types of systems.

External validity is about to what extent the findings are generalizable. Due to the focus on the security aspect and the small selection of papers, one should avoid generalizing the results over other properties such as reliability and safety. However, despite the limitations of our study (i.e., low number of papers, one method per paper, no snowballing to find complementary information), our conclusions are representative of the current issues in software security. Furthermore, other works in the state-of-the-art on software quality share similar conclusions as Morrison et. al. in [11], which points towards the applicability of our results to other properties.

VII. Conclusions

Despite the low number of papers this work is based on, we believe it is, to some extent, representative of the current issues in the state-of-the-art on software quality in general and security in particular: a number of useful properties and methods to evaluate them do exist today. However, it is difficult to know about them and understand whether they are applicable in a given context and if they are, how to use them. This can be attributed to the lack of information in the descriptions of their evaluation methods. This hampers the activities towards better quality assurance in software engineering and it limits at the same time the application of these activities or newly specified methods in industrial settings. For example, knowing when to apply a method (e.g., applicability at design time, at run-time, etc.) restricts which methods can be used in a given context. Knowing the advantages and disadvantages allows to trade-off available methods and limits selection bias. Older, more established or traditional, software quality fields provide more reference properties and methods to systematically compare to; however the level of detail and quality of given information is still relatively low.

As future work, we plan to expand the selection of papers to include those from the references in the analyzed papers so to investigate if our conclusions still hold. Similarly, we will explore the quality of assessments of other quality properties in literature such as reliability, safety and maintainability. The results of these studies will be included in PROMOpedia [21], an online encyclopedia of software properties and their evaluation methods. Lastly, we plan to propose an improved and validated ontology to express several critical and time sensitive properties towards a more systematic (in terms of consistent) and formal (in terms of codified) way and approach a better trade-off support between the properties.

Part of the work is also supported by the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737422. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Austria, Spain, Finland, Ireland, Sweden, Germany, Poland, Portugal, Netherlands, Belgium, Norway.

Acknowledgments

The work is supported by a research grant for the ORION project (reference number 20140218) from The Knowledge Foundation in Sweden.

References

[1] ISO/IEC, *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[2] ——, *Systems and software engineering-Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*, 2011.

[3] ——, "ISO/IEC 25000 software and system engineering–software product quality requirements and evaluation (SQuaRE)–guide to SQuaRE," *International Organization for Standardization*, 2005.

[4] S. Sentilles, E. Papatheocharous, F. Ciccozzi, and K. Petersen, "A property model ontology," in *Software Engineering and Advanced Applications (SEAA), 2016 42th Euromicro Conference on*. IEEE, 2016, pp. 165–172.

[5] G. McGraw, *Software security: building security in*. Addison-Wesley Professional, 2006, vol. 1.

[6] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering." in *EASE*, vol. 8, 2008, pp. 68–77.

[7] E. Chew, M. M. Swanson, K. M. Stine, N. Bartol, A. Brown, and W. Robinson, "Performance measurement guide for information security," Tech. Rep., 2008.

[8] R. Savola, "Towards a security metrics taxonomy for the information and communication technology industry," in *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*. IEEE, 2007, pp. 60–60.

[9] V. Verendel, "Quantified security is a weak hypothesis: a critical survey of results and assumptions," in *Proceedings of the 2009 workshop on New security paradigms workshop*. ACM, 2009, pp. 37–50.

[10] F. d. F. Rosa, R. Bonacin, and M. Jino, "The security assessment domain: A survey of taxonomies and ontologies," *arXiv preprint arXiv:1706.09772*, 2017.

[11] P. Morrison, D. Moye, and L. A. Williams, "Mapping the field of software security metrics," North Carolina State University. Dept. of Computer Science, Tech. Rep., 2014.

[12] J. L. Wright, M. McQueen, and L. Wellman, "Analyses of two end-user software vulnerability exposure metrics (extended version)," *Information Security Technical Report*, vol. 17, no. 4, pp. 173–184, 2013.

[13] ——, "Analyses of two end-user software vulnerability exposure metrics," in *2012 Seventh International Conference on Availability, Reliability and Security*. IEEE, 2012, pp. 1–10.

[14] I. V. Krsul, *Software vulnerability analysis*. Purdue University West Lafayette, IN, 1998.

[15] K. Buyens, B. De Win, and W. Joosen, "Identifying and resolving least privilege violations in software architectures," in *Availability, Reliability and Security, 2009. ARES'09. International Conference on*. IEEE, 2009, pp. 232–239.

[16] P. K. Manadhata, D. K. Kaynar, and J. M. Wing, "A formal model for a system's attack surface," CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, Tech. Rep., 2007.

[17] M. Lindvall, R. T. Tvedt, and P. Costa, "An empirically-based process for software architecture evaluation," *Empirical Software Engineering*, vol. 8, no. 1, pp. 83–108, 2003.

[18] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[19] C. P. Pfleeger and S. L. Pfleeger, *Security in computing*. Prentice Hall Professional Technical Reference, 2002.

[20] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Transactions on Software Engineering*, no. 3, pp. 371–386, 2010.

[21] S. Sentilles, F. Ciccozzi, and E. Papatheocharous, "PROMOpedia: a web-content management-based encyclopedia of software property models," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 45–48.

TABLE V: List of selected papers with their ID

pID	Paper reference
p1	B. Alshammari, C. Fidge, and D. Corney, "A hierarchical security assessment model for object-oriented programs," in Quality Software(QSIC), 2011 11th International Conference on. IEEE, 2011, pp. 218227.
p15	Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," inProceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement. ACM, 2008, pp. 315317.
p18	J. L. Wright, M. McQueen, and L. Wellman, "Analyses of two end-user software vulnerability exposure metrics (extended version)," Information Security Technical Report, vol. 17, no. 4, pp. 173184, 2013
p22	M. Almorsy, J. Grundy, and A. S. Ibrahim, "Automated software architecture security risk analysis using formalized signatures," in Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013, pp. 662671.
p25	Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," IEEE Transactions on Software Engineering,vol. 37, no. 6, pp. 772787, 2011.
p27	R. Ortalo, Y. Deswarte, and M. Kaaniche, "Experimenting with quantitative evaluation tools for monitoring operational security," IEEE Transactions on Software Engineering, no. 5, pp. 633650, 1999.
p37	P. Manadhata, J. Wing, M. Flynn, and M. McQueen, "Measuring the attack surfaces of two FTP daemons," in Proceedings of the 2nd ACMworkshop on Quality of protection. ACM, 2006, pp. 310.
p38	K. Buyens, R. Scandariato, and W. Joosen, "Measuring the interplay of security principles in software architectures," in Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE Computer Society, 2009, pp. 554563.
p43	V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in Proceedings of the 6th International Workshop on Security Measurements and Metrics. ACM, 2010, p. 3.
p47	J. Walden and M. Doyle, "Savi: Static-analysis vulnerability indicator," IEEE Security & Privacy, no. 1, 2012.
p50	J. A. Wang, H. Wang, M. Guo, and M. Xia, "Security metrics for software systems," in Proceedings of the 47th Annual Southeast Regional Conference. ACM, 2009, p. 47.
p51	I. Chowdhury, B. Chan, and M. Zulkernine, "Security metrics for source code structures," in Proceedings of the fourth international workshop on Software engineering for secure systems. ACM, 2008, pp. 5764.
p53	J. Walden, M. Doyle, G. A. Welch, and M. Whelan, "Security of open source web applications," in Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on.IEEE, 2009, pp. 545553
p59	M. Gegick, P. Rotella, and L. Williams, "Toward non-security failures as a predictor of security faults and failures," in International Symposium on Engineering Secure Software and Systems. Springer, 2009, pp. 135149.
p60	A. A. Younis, Y. K. Malaiya, and I. Ray, "Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability," in High-Assurance Systems Engineering (HASE),2014 IEEE 15th International Symposium on. IEEE, 2014, pp. 18
p63	A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on. IEEE,2013, pp. 6574.

Generation of Mimic Software Project Data Sets for Software Engineering Research

Maohua Gan
Graduate School of Natural Science and Technology
Okayama University
Okayama, Japan
pa2i5772@s.okayama-u.ac.jp

Kentaro Sasaki
Previously at Faculty of Engineering
Okayama University
Okayama, Japan
ken.default.0828@gmail.com

Akito Monden
Graduate School of Natural Science and Technology
Okayama University
Okayama, Japan
monden@okayama-u.ac.jp

Zeynep Yucel
Graduate School of Natural Science and Technology
Okayama University
Okayama, Japan
zeynep@okayama-u.ac.jp

***Abstract*—To conduct empirical research on industry software development, it is necessary to obtain data of real software projects from industry. However, only few such industry data sets are publicly available; and unfortunately, most of them are very old. In addition, most of today's software companies cannot make their data open, because software development involves many stakeholders, and thus, its data confidentiality must be strongly preserved. This paper proposes a method to artificially generate a "mimic" software project data set whose characteristics (such as average, standard deviation and correlation coefficients) are very similar to a given confidential data set. The proposed method uses the Box–Muller method for generating normally distributed random numbers, then, exponential transformation and number reordering are used for data mimicry. Instead of using the original (confidential) data set, researchers are expected to use the mimic data set to produce similar results as the original data set. To evaluate the usefulness of the proposed method, effort estimation models were built from an industry data set and its mimic data set. We confirmed that two models are very similar to each other, which suggests the usefulness of our proposal.**

Keywords— empirical software engineering, data confidentiality, software effort estimation, data mining

I. INTRODUCTION

In the research field of empirical software engineering, researchers demand for data of real software development projects from industry. However, only few industry data sets are publicly available. Also, these data sets are quite old, which becomes a great problem in ensuring the validity and reliability of the research. For example, tera-Promise repository [13] provides several industry data sets such as Desharnais [7], COCOMO '81 [4], Kemerer [9], Albrecht [1], but these data were recorded in the 1980's; thus, the development environments and processes may greatly differ from modern software development. In addition, the sample size is often very small, e.g. Kemerer has only 15 projects and Albrecht has only 24 projects. Surprisingly, these old and small data sets are still actively used in recent research papers in top journals (e.g. [2][11][16]) due to lack of new industry data sets.

Meanwhile, although many companies measure and accumulate data of recent software development projects, it becomes more and more difficult for university researchers to use them for the research because the legal compliance to various data protection regulations has become extremely important for todays' companies. Moreover, since software development involves many stakeholders, their data confidentiality must be strongly preserved; thus, it became more difficult to take the data out of the company. In addition, although there are some studies performed using the latest software development data, only their analysis results are disclosed and the data itself is not disclosed. For example, the white paper on software development data in 2016-2017 [17] provides various analysis results of 4046 software development projects held in 31 Japanese software development companies; however, the data set itself is not disclosed.

In this paper, to make it possible for academic researchers to use the confidential software project data set of a company, we propose a method to artificially create a mimic data set that has very similar characteristics to a given confidential data set. Instead of using the original (confidential) data set, researchers are expected to use the mimic data set to produce similar results as the original data set. For example, researchers can use the mimic data set for the purpose of evaluation of software effort estimation methods, because many industry data sets are required for the evaluation of the stability assessment of the methods [16]. Moreover, such a mimic data set is also useful to practitioners because many companies want to compare their software development performance (such as productivity and defect density) with other companies.

As a basic idea of our proposal, we measure statistics of each variable as well as correlation coefficients between all pairs of variables in a confidential data set. Next, to produce a mimic variable, we use the Box–Muller method [5] for generating normally distributed random numbers; then, exponential transformation is applied to the generated values to mimic the value distribution of the original variable. After generating all mimic variables, number reordering is applied to the generated values to mimic the correlation coefficients between all pairs of original variables.

Interestingly, our method can freely determine the number of data points to generate. For example, we could produce a data set of sample size n = 1000, which means 1000 projects, from an original data set with much smaller samples, e.g. n = 30. This

TABLE I. AN EXAMPLE OF SOFTWARE PROJECT DATA SET (EXCERPT FROM DESHARNAIS DATA SET [7])

PM experience	Team experience	Language A	FP	Duration	Effort (person-hours)
1	1	0	140	5	2520
7	4	0	113	13	1603
3	1	1	291	8	3626
3	1	1	67	10	1267
4	0	0	99	6	546
5	4	1	645	26	9100

also means that there is no one-to-one mapping of projects between the original data set and the mimic data set. Therefore, data privacy and confidentiality are effectively protected even if the mimic data set is made open.

In contrast, conventional data anonymizing methods for software engineering data employ *data mutation* techniques to gain data privacy [14][15]. Since data mutation keeps the one-to-one mapping of data points between the anonymized data set and the original data set, threats of breaking the anonymity cannot be perfectly prevented. Moreover, since strong data mutation yields change of data characteristics, balancing privacy and utility is a big challenge in this approach [15]. On the other hand, our mimic data set is composed of randomly generated data points without keeping the one-to-one mapping to the original data set, data anonymization is much more effectively achieved. We believe that companies are more confident in using our method than using the data mutation to comply with various data protection regulations.

To evaluate the utility of the proposed method, this paper presents a case study of producing a mimic data set from Deshanais data set [7], which is one of the most frequently used data sets in software effort estimation study [10]. In the case study, we built effort estimation models from both the original data set and the mimic data set to see whether we could obtain similar results from both data sets.

II. RELATED WORK

Peters et. al [14] proposed a data anonymization method called MORPH to solve privacy issues in software development organizations. They target defect prediction research and try to anonymize the defect data set that consists of various software metrics measured for each source file of a software product. They use data mutation techniques, which add small amount of changes to each value to make it difficult to identify a specific source file in a data set. They further propose a method called CLIFF, which allows to eliminate some data points that are not necessary for the defect prediction. Combining CLIFF with MORPH, they try to balance privacy and utility of defect data sets [15].

Since their approach is specially proposed for two group classification problem (i.e., distinguishing defect-prone files and not defect-prone files in a defect data set), it cannot be applied to general purpose data sets such as software project data sets that we target in this paper. In addition, since data mutation keeps the one-to-one mapping of data points between the anonymized data set and the original data set except for eliminated ones, threats of breaking the anonymity cannot be perfectly prevented. In contrast, we try to produce a completely artificial data set from given characteristics of a confidential data set.

III. THE PROPOSED METHOD

A. Basic Idea and Procedure

In this paper, a confidential data set that needs to be kept secret is called a "source data set" or a simply "source data." And, the artificially generated data to mimic the source data is called a "mimic data set" or "mimic data."

As source data, we target software project data sets. Table I shows a part of Desharnais data set [7], which is one of the commonly used software project data sets for effort estimation studies. In Table I, "PM" stands for "project manager" and "FP" stands for "function point." Many software companies record similar data sets that consist of various project features. In this paper we assume that there is no missing value in a data set.

Typically, software project data sets contain software size metrics such as Function Point (FP) and Source Lines of Code (SLOC), as well as the project length (often denoted as "duration"), and the development effort. It has been known that the probability distribution of these variables roughly follows log-normal distribution [10]. Therefore, this paper approximates the value distribution of quantitative variables by the log-normal distribution.

After setting the number of cases n to be generated in the mimic data, the procedure to generate mimic data from source data is described as follows:

- Step 1: For each ratio scale or interval scale variable in the source data, generate a set of artificial values whose distribution is similar to the source data.
- Step 2: For each ordinal scale or nominal scale variable in the source data, generate a set of artificial values whose distribution is similar to the source data.
- Step 3: For all variables in the mimic data, repeat swapping of values so that the correlation coefficient matrix of the mimic data becomes similar to that of the source data.

In the next section, details of these steps are described.

B. Step 1. Generation of Ratio/Interval Scale Variables

This paper employs the Box–Muller method [5] to generate quantitative variables. The Box–Muller method, also called the Box-Muller transform, is an algorithm for generating a pairs of normally distributed random numbers $N(\mu, \sigma^2)$ from given uniformly distributed random numbers. Its mathematical expression is as follows:

$$N_1 = \sigma\sqrt{-2logR_1}cos2\pi R_2 + \mu$$

$$N_2 = \sigma\sqrt{-2logR_1}sin2\pi R_2 + \mu$$

where R_1 and R_2 are independent samples from the uniformly distributed random numbers on the interval $(0,1)$. These R_1 and R_2 are easily generated in many programming languages (e.g. by using rand() function in C language). N_1 and N_2 are independent random variables with a normal distribution. In this paper we use N_1 only.

As mentioned above, we assume quantitative variables follow log-normal distribution. To generate log-normally distributed random numbers, we apply exponential transformation, which is inverse transformation of the logarithmic transformation, to values obtained by the Box-Muller method.

As an example, Fig. 1 shows the value distribution of "effort" in Desharnais data set, which we consider as source data. Fig. 2 shows its log-transformed value distribution. We see in Fig. 2 that log-transformed effort values roughly follow the normal distribution. We can use the standard deviation σ and the mean value μ of Fig. 2 to generate the mimic data by the Box-Muller method. Fig. 3 shows the result of the Box-Muller method, which is the mimic data of Fig. 2. Finally, Fig. 4 shows the result of its exponential transformation, which is a mimic data of Fig. 1. Although values in Fig. 4 are all artificially generated ones, we see that Fig 4 well resembles Fig. 1.

In addition, by the following equation, we can directly obtain the standard deviation σ and the mean value μ of log-transformed source data from the standard deviation σ' and the mean value μ' of original source data.

$$\sigma^2 = \ln\{1 + (\sigma'/\mu')^2\}$$

$$\mu = \ln(\mu') - \sigma^2/2$$

This means that, a company who own a (secret) source data only needs to provide σ' and μ' directly computed from the source data.

C. Step 2. Generation of Ordinal/Nominal Scale Variables

For each ordinal scale or nominal scale variable in the source data, we generate a set of artificial values so that the percentage of cases in each bin is same as the source data. For example, assuming that we have an ordinal scale variable "requirement clarity," which has four ranks or bins ("1. very clear", "2. clear", "3. unclear", "4. very unclear"). As also assume that the percentage of values belonging to these bins are 20% for "1. very clear", 25% for "2. clear", 35% for "3. unclear" and 10% for "5. very unclear" respectively. Then, to generate a mimic data, we simply generate an artificial mimic sample whose percentage of cases in each bin is same as that of the source data.

D. Step 3. Mimicking the Relationship among Variables

For all pairs of variables in the source data, there may exists some sort of relationship. This paper captures such relationships via the correlation coefficient matrix of the source data; and, the proposed method tries to make the correlation coefficient matrix of the mimic data close to that of the source data. This can be done by swapping values within a variable, which does not break the value distribution of that variable. In this study, we assume there is some outliers in the source data; therefore, we decided to use Spearman's rank correlation coefficient instead of the Pearson correlation coefficient to capture the relationships among variables.

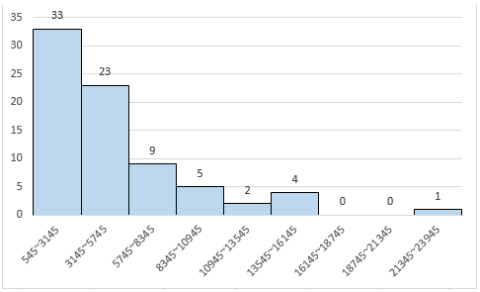

Fig. 1 Histogram of software development effort of Desharnais data set.

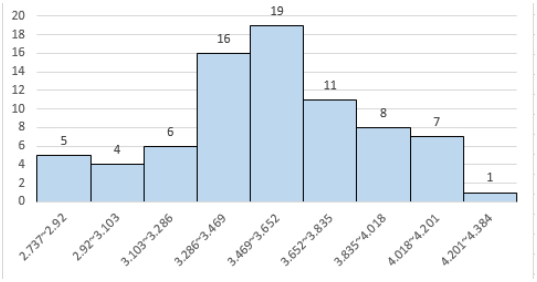

Fig. 2 Histogram of log-transformed software development effort of Desharnais data set.

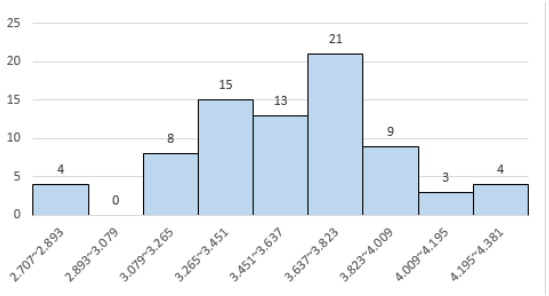

Fig. 3 Histogram of mimic data of log-transformed software development effort of Desharnais data set.

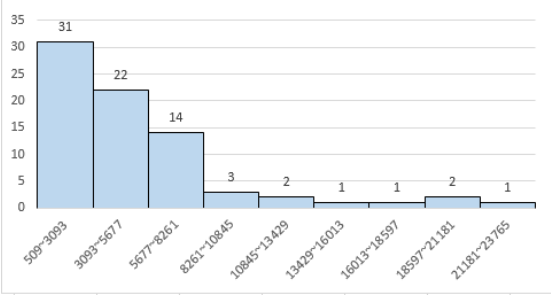

Fig. 4 Histogram of mimic data of software development effort of Desharnais data set.

We propose the following procedure to mimic the relationship among variables in source data.

1. Compute the correlation coefficient matrix of the source data.
2. Randomly select one variable in the mimic data. Then, randomly select two values from this variable, and swap them.
3. If the correlation coefficient matrix of the mimic data becomes more similar to that of the source data, we consider that the value swapping is successful, and go back to Step 2. Otherwise, we consider that the swapping is unsuccessful, cancel swapping and go back to Step 2. To evaluate the similarity of the correlation coefficient matrices, we use the sum of squared differences ($\sum(a_i - b_i)^2$) between the set of rank correlation coefficients of the mimic data and those of the source data. If the sum of squared differences becomes smaller after the swapping, then we consider that the correlation coefficient matrices get more similar.
4. When the sum of squared differences converges, swapping is completed (i.e. stop repeating Step 2.)

E. Rounding Off Generated Values

This is an additional step to make the mimic data visually more similar to the source data. Since the values of quantitative variables are generated from random numbers, their significant figures are different from that of source data. For this reason, each value should be rounded off to an appropriate precision according to the significant figure of source data. For example, Function Point is an integer in source data, so it should be rounded off to integer.

IV. CASE STUDY

To evaluate the effectiveness of the proposed method, this section presents a case study to generate a mimic data from Desharnais data set [7]. In the case study, we built effort estimation models from both the source data and the mimic data to investigate their similarity.

A. Source data set

The Desharnais data set is one of the most frequently used data sets in software effort estimation research [10]. It contains 77 projects without missing values. This case study generated mimic data of same sample size (n=77.) Quantitative variables used in this paper are Duration, Transactions, Entities, PointsAdjust, and Effort. And, qualitative variables used are TeamExp, ManagerExp, and Lang. TeamExp and ManagerExp are ordinal scale variables, the TeamExp ranges from 0 to 4, and the ManagerExp ranges from 0 to 7. The variable Lang is divided into two binary variables Lang2 and Lang3.

B. Characteristics of Generated Ratio/Interval Scale Variables

The mean value, standard deviation, maximum value and minimum value of quantitative variables of source data and mimic data are shown in Table II and Table III respectively. Their relative differences are shown in Table IV. From these results, we see that the difference of mean value, standard deviation and minimum value between two data sets are very small, which indicates effectiveness of the proposed method. On the other hand, the maximum values are turned out to be not very similar. This is because source data contain outliers. Mimicking the outliers are our important future work.

TABLE II. STATISTICS OF SOURCE DATA

	Mean value	Standard deviation	Maximum value	Minimum value
Duration	11.299	6.742	36	1
Transactions	177.468	145.129	886	9
Entities	120.545	85.547	387	7
PointsAdjust	298.013	181.076	1127	73
Effort	4833.909	4160.9	23940	546

TABLE III. STATISTICS OF MIMIC DATA

	Mean value	Standard deviation	Maximum value	Minimum value
Duration	11.571	7.172	42	3
Transactions	180.078	139.485	822	39
Entities	123.208	91.018	534	29
PointsAdjust	300.299	168.783	986	99
Effort	4913.26	4246.176	25365	893

TABLE IV. RELATIVE DIFFERENCE OF STATISTICS BETWEEN SOURCE DATA AND MIMIC DATA

	Mean value	Standard deviation	Maximum value	Minimum value
Duration	0.024	0.06	0.143	0.667
Transactions	0.014	0.04	0.078	0.769
Entities	0.022	0.06	0.275	0.759
PointsAdjust	0.008	0.073	0.143	0.263
Effort	0.016	0.02	0.056	0.389

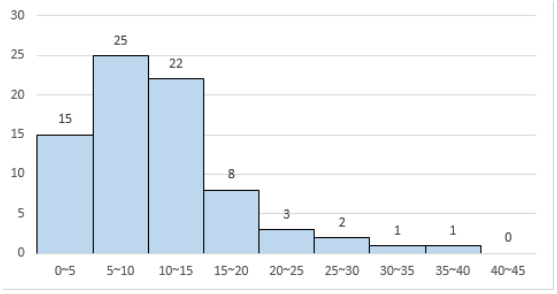

Fig. 5.1 Histogram of Duration of source data.

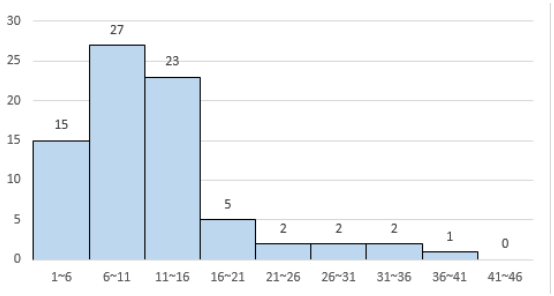

Fig. 5.2 Histogram of Duration of mimic data.

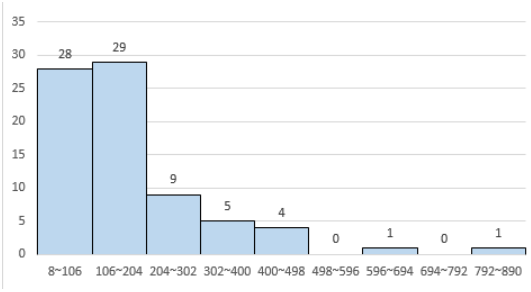

Fig. 5.3 Histogram of Transactions of source data.

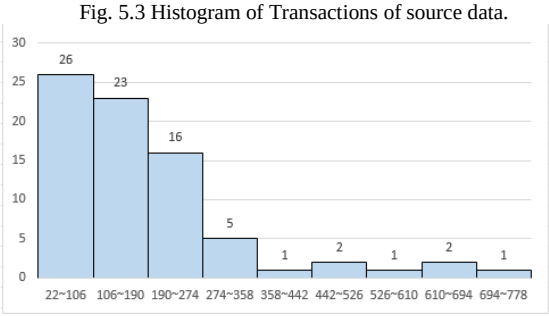

Fig. 5.4 Histogram of Transactions of mimic data.

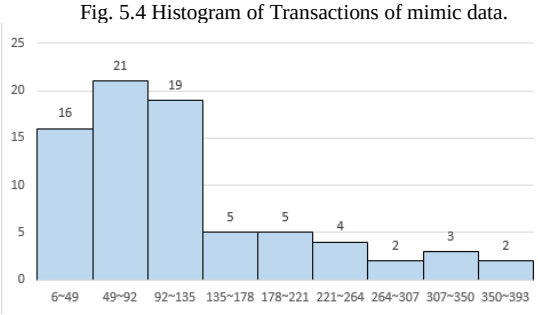

Fig. 5.5 Histogram of Entities of source data.

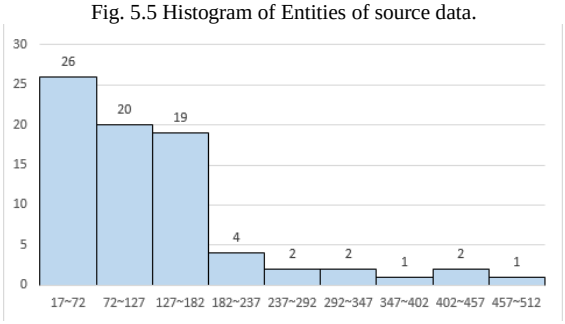

Fig. 5.6 Histogram of Entities of mimic data.

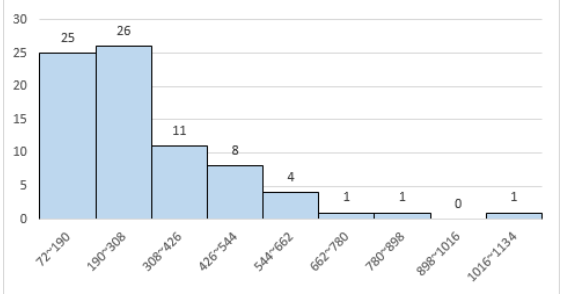

Fig. 5.7 Histogram of PointsAdjust of source data.

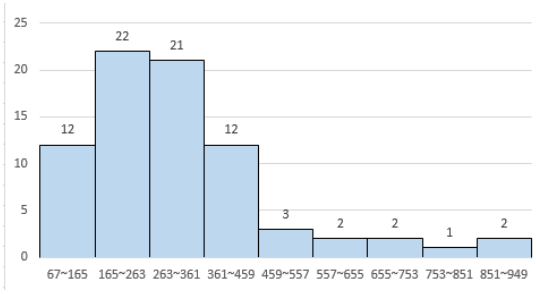

Fig. 5.8 Histogram of PointsAdjust of mimic data.

For more details of the generated variables, the distribution of source data and mimic data of the four quantitative variables are shown in Figure 5.1 to Figure 5.8. From these figures we can also visually see the similarity between two data sets. (For the variable "Effort", we have already shown the histograms in Fig. 1 and Fig. 4.)

C. Rank Correlation Coefficient Matrix

Fig. 6 shows the convergence of the sum of squared differences of rank correlation coefficients when increasing the number of updates (i.e. successful swapping) of variables. As shown in the figure, the sum squared differences becomes very close to zero (0.000069) as the number of updates increases.

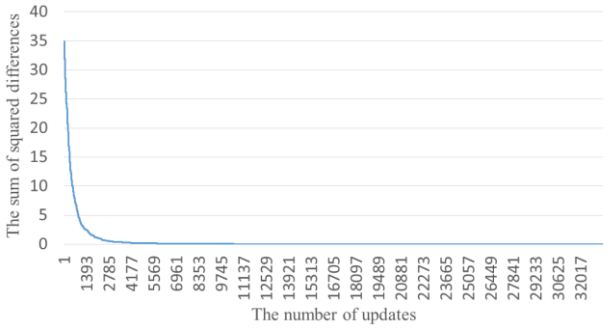

Fig. 6 Convergence of the sum of squared differences of rank correlation coefficients.

TABLE V. RANK CORRELATION COEFFICIENT MATRIX OF SOURCE DATA

	TeamExp	Manager Exp	Duration	Transactions	Entities	PointsAdjust
TeamExp	1.000					
ManagerExp	0.388	1.000				
Duration	0.365	0.233	1.000			
Transactions	0.087	0.109	0.382	1.000		
Entities	0.319	0.170	0.533	0.265	1.000	
PointsAdjust	0.266	0.190	0.592	0.744	0.778	1.000
Lang2	-0.073	0.157	0.147	-0.129	0.045	-0.039
Lang3	-0.075	0.180	-0.106	0.248	-0.120	0.077
Effort	0.252	0.086	0.572	0.467	0.647	0.688

TABLE VI. RANK CORRELATION COEFFICIENT MATRIX OF MIMIC DATA

	TeamExp	Manager Exp	Duration	Transactions	Entities	PointsAdjust
TeamExp	1.000					
ManagerExp	0.389	1.000				
Duration	0.365	0.235	1.000			
Transactions	0.088	0.109	0.381	1.000		
Entities	0.319	0.170	0.532	0.265	1.000	
PointsAdjust	0.266	0.190	0.591	0.742	0.776	1.000
Lang2	-0.071	0.165	0.146	-0.128	0.045	-0.039
Lang3	-0.067	0.187	-0.106	0.248	-0.120	0.077
Effort	0.252	0.086	0.572	0.466	0.647	0.690

TABLE VII. EFFORT ESTIMATION MODEL FOR SOURCE DATA

	Coefficient	p-value
Intercept	1.373	0.000
TeamExp	-0.006	0.727
ManagerExp	0.010	0.550
LOG(Length)	0.254	0.016
LOG(Transactions)	0.204	0.127
LOG(Entities)	0.184	0.173
LOG(PointsAdjust)	0.504	0.046
Lang2	-0.065	0.195
Lang3	-0.604	0.000

TABLE VIII. EFFORT ESTIMATION MODEL FOR MIMIC DATA

	Coefficient	p-value
Intercept	1.502	0.000
TeamExp	-0.020	0.329
ManagerExp	0.011	0.554
LOG(Length)	0.162	0.180
LOG(Transactions)	0.259	0.082
LOG(Entities)	0.217	0.159
LOG(PointsAdjust)	0.424	0.127
Lang2	-0.073	0.205
Lang3	-0.602	0.000

A part of rank correlation coefficient matrix for each data set is shown in Table 5 and Table 6. From Table 5 and Table 6, we can see that the maximum of the difference is 0.008, which is sufficiently small. So it is considered that the relationship between any two variables is sufficiently reproduced.

D. The Comparison of Predictive Model about Man-hour

Assuming the effort estimation research using mimic data. we conduct log-log regression modeling on both source data and mimic data respectively, and investigate their similarity. The objective variable is "Effort" and other variables are predictor variables. The log-log regression model is a linear regression model with logarithmic transformation applied to both predictor variables and the objective variable before model construction. Kitchenham and Mendes [10] pointed out the necessity of logarithmic transformation to improve the prediction performance of effort estimation models.

The result of log-log regression for source data and mimic data are shown in Table VII and Table VIII. From these tables, we see constant (intercept) and coefficients of predictor variables are similar. The R^2 values of these models are 0.882 for source data and 0.820 for mimic data, which are also similar. Looking at p-values, for some variable, p-value is not very similar. One of the possible reason is that outliers might affected the p-value. We need further investigation in our future study. Also, in future we will evaluate the prediction performance of the models.

V. SUMMARY

In this paper we proposed a method for artificially generating a mimic data set from a given (confidential) source data set. From a case study with a software project data set, our main findings are as follows.

- The standard deviation and the mean value of quantitative variables of mimic data are very similar to that of source data.
- The rank correlation coefficient matrix of mimic data is very similar to that of source data.
- Effort estimation models using log-log regression modeling built from source data and mimic data are similar in their coefficients.

In future, we will evaluate the prediction performance of the built models. Also, we will apply various data analysis techniques such as clustering and association rule mining for mimic data to evaluate the utility of the proposed method. In addition, we will try to improve our method by mimicking more aspects in source data, such as outliers, skewness and kurtosis of variables.

REFERENCES

[1] A. J. Albrecht, J. Gaffney, "Software function, source lines of code, and development effort prediction," IEEE Transactions on Software Engineering, vol. 9, pp.639-648, 1983.

[2] M. Azzeh, M., "A replicated assessment and comparison of adaptation techniques for analogy-based effort estimation," Empirical Software Engineering, vol.17, no.1-2, pp.90-127, 2012.

[3] B. Baskeles, B. Turhan, and A. Bener, "Software effort estimation using machine learning methods," Proc. 22nd International Symposium on Computer and Information Sciences (ISCIS2007), pp.126-131, Dec. 2007.

[4] B. Boehm, "Software engineering economics," Prentice-Hall, NY, 1981.

[5] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," The Annals of Mathematical Statistics, vol. 29, no. 2 pp. 610–611, 1958.

[6] L. Briand, T. Langley, and I. Wieczorrek, "A replicated assessment and comparison of common software cost modeling techniques," Proc. 22nd International Conference on Software Engineering (ICSE2000), pp.377-386, 2000.

[7] J.-M. Desharnais, "Analyse statistique de la productivitie des projects informatique a partie de la technique des point des function," Master's Thesis, University of Montreal, 1989.

[8] M. C. Jones, and A. Pewsey, "Sinh-arcsinh distributions," Biometrika, vol.96, no.4, pp.761-780, Dec. 2009.

[9] C. F. Kemerer, "An empirical validation of software cost estimation models," Communications of the ACM, vol. 30, no. 5, pp. 416-429, 1987.

[10] B. Kitchenham, and E. Mendes, "Why comparative effort prediction studies may be invalid," Proc. 5th International Conference on Predictor Models in Software Engineering, Article no.4, May 2009.

[11] E. Kocaguneli, T. Menzies, J. Keung, "On the value of ensemble effort estimation", IEEE Transactions on Software Engineering, vol. 38, no. 6, pp. 1403-1416, 2012.

[12] K. Maxwell, "Applied statistics for software managers," Englewood Cliffs, NJ, Prentice-Hall, 2002.

[13] T. Menzies, R. Krishna, and D. Pryor, "The promise repository of empirical software engineering data," http://openscience.us/repo, North Carolina State University, Department of Computer Science, 2015.

[14] F. Peters and T. Menzies, "Privacy and utility for defect prediction: experiments with MORPH," Proc. International Conference on Software Engineering, pp.189-199, 2012.

[15] F. Peters, T. Menzies, L. Gong, and H. Zhang, "Balancing privacy and utility in cross-company defect prediction," IEEE Transactions on Software Engineering, vol. 39, no. 8, pp. 1054-1068, 2013.

[16] P. Phannachitta, J. Keung, A. Monden, and K. Matsumoto, "A stability assessment of solution adaptation techniques for analogy-based software effort estimation," Empirical Software Engineering, vol.22, no.1, pp.474-504, 2017.

[17] Software Reliability Enhancement Center, Information-technology Promotion Agency, "White paper on software development data in 2016-2017," SEC Books, 2016

Real-time DevOps Analytics in Practice

Mahesh Kuruba
Digitate, Tata Consultancy Services
Pune, India
mahesh_kuruba@ieee.org

Prashant Shenava
Digitate, Tata Consultancy Services
Pune, India
prashant.shenava@digitate.com

Jenny James
Digitate, Tata Consultancy Services
Pune, India
jenny.james@digitate.com

***Abstract*—Agile is being adopted widely in the industry, but it is posing new challenges to operations team. The quality of software delivered at such a rapid pace may not necessarily meet the operational requirements and may break the existing system or the dependent systems. The operations team may not have enough knowledge or know-how on the changes made to the system which are moved to production. Also, they do not have enough visibility on the stability of application build. This most frequently results in to a logjam between development and operations team to release the functionality. This phenomenon is discouraging for agile adoption. Hence, the need for collaboration between Dev and Ops teams increasingly becoming significant. To ensure DevOps activities are meeting the expectations and providing transparency across development and operations, the need for using abundant data generated during operations and development activities becomes significantly important. Leveraging analytics for the data generated will provide enough insights to development and operations teams to improve code quality and faster incident resolution for increased productivity; and application owners to make informed decisions.**

In this paper, we present our experience of designing, developing and usage of DevOps analytics system using open source tools for a product team in industrial setting. We define DevOps analytics, identify the focus areas for the product team and present the visual correlation of the data to bring in transparency and help development team in improving the quality of their code and for operations team to resolve the incidents faster. Also, we present our next steps for the DevOps analytics journey.

Keywords—DevOps Analytics, Error analytics, Performance analytics, DevOps Glass wall, Infrastructure analytics

I. INTRODUCTION

Agility in other words is agile methodology, has brought in new challenges to operations team. The rate at which the software is being delivered and the issues that operations team has to address in production (post release) are on rise and this is becoming extremely challenging for operations team to maintain applications, as they need to quickly detect the problem and fix it. Today locating the problem across the layers (application, middleware, infrastructure) itself consumes lot of time. The operations team is not confident enough to take the changes at such a rapid pace, hence they would need enough visibility on the 'build' stability. In short, operations team need a quality software which doesn't break the existing system. The magnitude of the problem is amplified, if it's a multi-vendor scenario in a medium/large IT estate. This is becoming a deterrent to move software changes to production at the same rate as agile teams delivering software. Due to this, the business doesn't necessarily see the value of agile methodology.

To address this problem, organizations are adopting DevOps practices to increase the collaboration across Dev and Ops teams. An integrated analytics based approach that provides runtime-data visibility to development team, and design-time data visibility to operations team, helps bring trust and transparency among the teams.

Software applications and underlying infrastructure generate huge data during design time and runtime. Analyzing this data and deriving insights involves necessary tools and expertise. In ever changing IT landscape it is humanly impossible for the development or operations teams to go across multiple layers of application and technology stack to detect problem at any point of time and to make decisions. Hence, there is a need for a real-time DevOps analytics tool, which will provide insights to support and development team on a real-time basis.

Gartner [1] predicts that by 2020, 50% of IT organizations will apply advanced analytics in application development to improve application quality and speed of delivery. IDC [2] predicts by 2019, over 70% of routine development-lifecycle tasks will be automated, supported by Artificial Intelligence (AI) fed from existing DevOps pipeline data.

As a first step towards DevOps intelligence, in this paper, we present our experience of designing, developing and usage of DevOps descriptive analytics using the open source tools for a product team in industrial setting. We defined DevOps analytics, identified the focus areas for the product team and presented the correlation of the data to bring in transparency and help development team in improving the quality of their code and for operations team to resolve the incidents faster. Also, we present our next steps in the DevOps analytics journey.

II. LITERATURE REVIEW

Snyder and Curtis [4] adopted analytics to monitor Agile-DevOps transformation program using CAST application intelligence platform [4]. The focus was on quality and productivity in the development phase. The quality was based on static code analysis tools. Augustine et.al [5] at ABB has deployed analytics based on the ALM tools data, like work item count and its status, cycle time of a work item, defect backlog count, and code ownership. The focus was on the progress and status. Gousios et al [6] presented an approach for real-time software analytics, but do not mention any specific areas of application. Cito [7] proposed research focusing on data-driven decision making for software engineers during development based on the run-time information of method

calls and execution times. Baysal et.al [8] proposed using qualitative information for developers, in addition to quantitative dashboards to improve code quality. Johnson [9] highlighted the need for automated and low barrier analytics tools.

Based on the literature review, we observe that there is enough focus on analytics based on design data and there are tools in the market. Operational analytics has been implemented by several organization at the infrastructure and middleware layer and has various tools in the market. The existing tools does not offer a blend of development and operations analytics. Hence, there is a need for DevOps analytics which provides insights to both Dev and Ops teams thus making the wall between Dev and Ops as a '*DevOps Glass wall*' which is transparent between the teams and enable in building trust between teams.

III. DEVOPS ANALYTICS

A. *DevOps framework*

DevOps has multiple interpretations in the industry based on the practices adopted and their maturity level. The practices include Continuous integration-Continuous delivery-Continuous deployment, automated testing, automated environment provisioning in the cloud or virtual machines and collaboration between Dev and Ops teams. Author's view of DevOps is adopted from Sharma and Coyne [3] as shown in fig 1. The various phases of software development are performed in smaller and faster increments and range from continuous planning, build and integration, testing, deployment, monitoring in operations to providing feedback through analytics and insights. The key pillars to achieve them are collaboration, automation, infrastructure and environment provisioning and metrics. Analytics will leverage data from various phases and sources in the DevOps cycle and provide necessary insights thus strengthening the feedback loop.

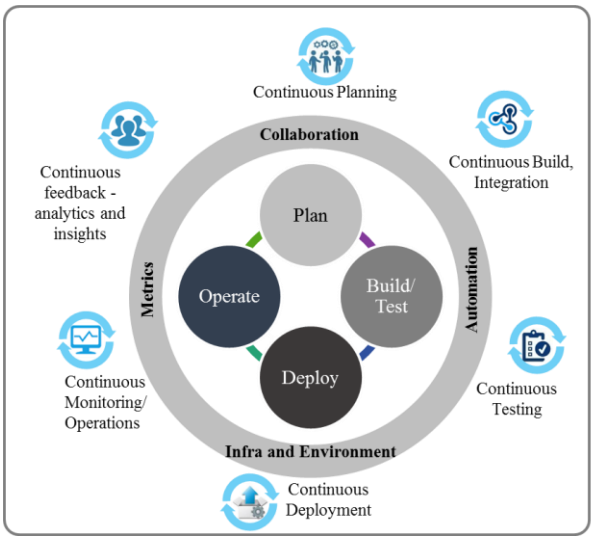

Fig. 1. DevOps framework

B. *DevOps analytics*

Traditionally the wall between development and operations team continues for organization structural and regulatory reasons; while collaboration is trying to bring both the teams together, data driven culture based on analytics brings in transparency through glass wall as shown in fig 2.

DevOps analytics is defined as: "integrated analytics on software data generated in runtime (production) and design time (test and development) to provide insights to development (Dev) and operations (Ops) teams to improve software quality, productivity to make informed decisions".

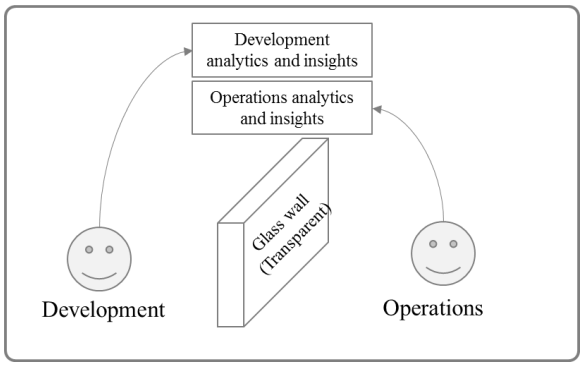

Fig. 2. Analytics enabling transparency in DevOps teams

C. *DevOps analytics implementation*

In this paper we discuss about DevOps analytics implementation for a product team. The product team has adopted agile and DevOps practices. To strengthen the product quality, the product unit decided to leverage the data to make informed decisions on the product quality and performance improvements. Also, the initiative is expected to reduce the incident resolution time of the product support team.

The product development team in scope has built CI pipeline with Jenkins as orchestrator and maven for build. The SAST and DAST is being performed using a commercial product and Ansible is used for product deployment. The product support team uses a cloud based ticket management system for product support requests and incidents.

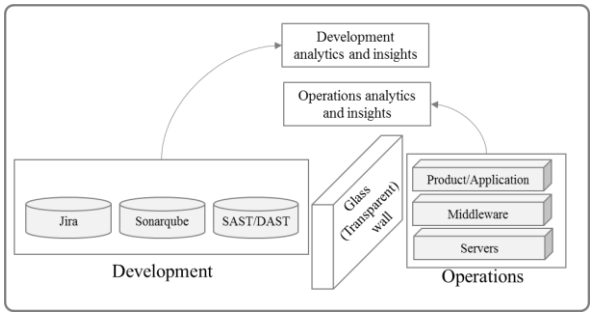

Fig. 3. Architecture representation of DevOps analytics

As shown in fig 3, the data sources from development includes features, defects (jira), code quality (sonarqube) and SAST/DAST tool. The data sources from operations includes

the ticket management tool, metrics and logs from servers, middleware and applications.

While the DevOps analytics can potentially provide answers to several questions, product team wanted to focus on most frequently asked questions [10]. To begin with, product team has identified the following focus areas and correlation across these focus areas and layers; with an objective of improving software quality and reducing incident resolution time for operations team:

- Application usage analytics
- Performance analytics
- Error analytics
- Code quality analytics
- Infrastructure analytics

Application usage analytics, helps development and testing teams to focus on the most used features and traversed paths by application users. The program files that are being used for these features, coupled with their code quality metrics will provide insights on the complexity and maintainability of these programs. Thus teams can focus on these paths to improve their software quality and use their effort effectively to improve productivity.

Similarly, understanding the performance issues and frequent errors in program execution observed in production environment, will provide insights to development team on the programming issues and fix them to improve the software quality.

IV. ARCHITECTURE

The application in scope are deployed on multiple servers and generate data in their respective servers. The development tools like sonarqube and SAST/DAST tools are in different systems. The data had to be streamed from these data sources for analytics and insights.

We have used the open source tools like File beat, Metric beat, Apache HBASE, Logstash, Solr and Banana for data aggregation, paring, transformation, storage, querying and visualization. The architecture for DevOps analytics and insights system is shown in fig 4.

The various data sources across application, middleware and infrastructure include httpd, application, database and syslogs, code quality, queue metrics are collected. The data collected are of two types log data and metric data. Log data is textual and metric is numerical data. Real-time data is being aggregated through file beat and metric beat from these data sources. File or metric beat is installed in each of these data source systems. Once aggregated data is ingested and stored in HBASE by defining canonical model. The data is then queried from Solr and visualized in Banana through dashboards for Dev and Ops teams at real-time. The visualization is defined based on the needs of Dev and Ops teams.

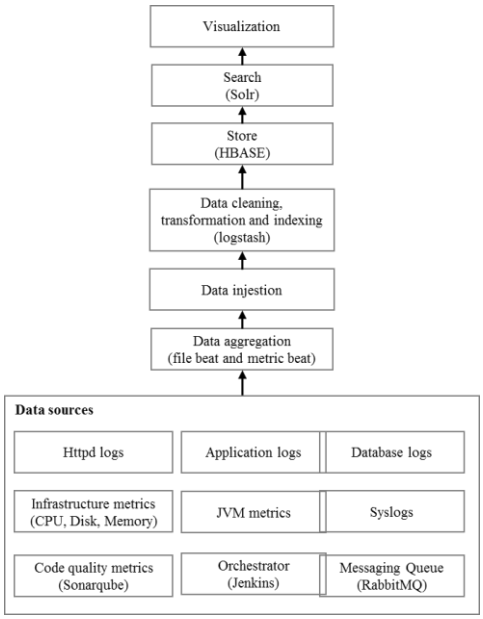

Fig. 4. Architecture representation of DevOps analytics system

V. VISUALIZATION

We initiated DevOps analytics journey with visualization and descriptive analytics of various data like application usage, performance metrics, errors in logs, code quality metrics and infrastructure metrics.

A. *Application usage analytics:*

Application usage in runtime environment is a key measure for several insights and decisions for the application owner. Some of the measures and their significance is mentioned below.

- Number of active application users: usage in terms of number of users will assist application owner in deciding if the investment is justified and should they continue, increase or reduce investment.
- Peak usage in a day: application owner or release manager can decide appropriate time of applying patch, when there is no or minimal usage, which is today intuitive in several organizations.
- Geographical location of the application users: The geographical location of application users indicates active or inactive users in the geography. This triggers causal analysis for low usage in a geography and understand the reasons for low usage of application in the geography (or) if the application owners needs to increase communication and promotional activities.

Number of active application users and their geographic location is of interest to the application user. Peak application usage is of interest to the operations head.

Usage pattern for the application in scope is shown in fig 5. The application usage typically starts at 8:30 AM IST onwards. There is no usage from midnight till 8:30 AM

IST. The maintenance window for the application should be in the lean usage period. Considering the usage and the cost of operations, the application owner can continue to invest in this application. For illustration purpose, we have represented a day's usage. By extending the duration over a period of time, application owner can determine the usage by week and month to identify patterns in the usage. Identify any anomalies in the usage pattern, caused due to increased/decreased sales of the application.

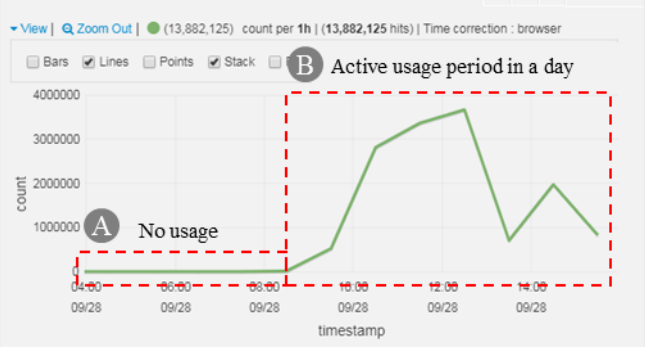

Fig. 5. Application usage analytics

Application users by geography is shown in fig 6. This indicates the number of active users from a city and country. Large community of application users from Chennai, India corroborates with the fact of their presence in Chennai.

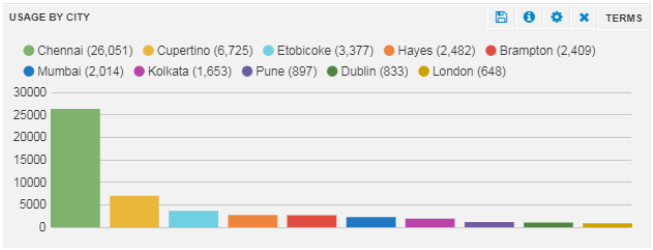

Fig. 6. Application usage by geography location

B. *Performance analytics:*

Application development team want to improve performance, by understanding methods that are being called most frequently and the methods that consume high processing time. This information from runtime environment provides insights to the programmers in optimizing their design and code.

The Top-N modules and methods of an application is shown in fig 7. Number of times the module is called is shown as 'count' and the average processing time of the method is shown as 'mean' (A). Similarly the aggregation of all methods in a module are shown as 'count' and 'mean' at the module level (B).

The development team selected the top 'N' modules and methods for code optimization. Once the performance has been improved by optimizing the processing time, they identified the other top 'N' time consuming methods. This process is repeated till the team achieves the required application performance levels. In case the development team can't achieve required levels of performance, they modify the deployment structure by deploying the components on multiple servers.

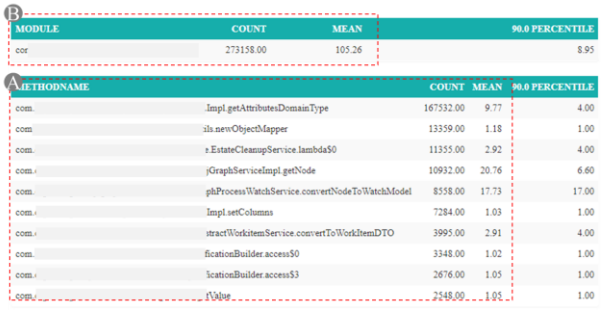

Fig. 7. Top-N time consuming modules and methods

C. *Error analytics*

There are various types of logs namely – application logs, error logs and diagnostic logs. Operations team typically browse through multiple log files at the time of incident to detect the location of the problem. This is a time consuming process and needs expertise to visualize and detect the problem. The problem gets complex in a multi-vendor scenario, in which there is no clear ownership of the incident. Error analytics will help the operations team to improve transparency and reduce the resolution time.

The application and error logs have been ingested for the application in scope. A dashboard showing the number of logs and the error logs by module are shown in fig 8. Based on the transaction id, the operations team can quickly diagnose the module and the method in which there is an error occurred and even before the user raises an incident, operations team can resolve the incident. The operations team can drill down to the log statement at real-time in a single window, instead of browsing through multiple logs to identify the location of the problem through logs as shown in fig 8.

This helps development team to understand the most frequently occurring error types and methods in which the error is found, to provide a permanent fix and reduce the number of incidents for operations team.

As a next step, we intend to generate alerts to the operations team based on the anomaly that has been detected in the logs and the error type.

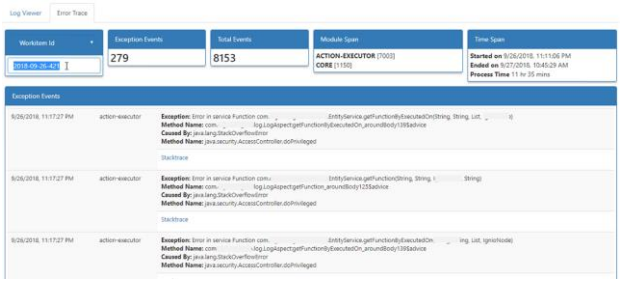

Fig. 8. Error analytics

D. *Code quality analytics*

As discussed earlier, the operations team may not have enough confidence on the build shared at a rapid pace, to be deployed in production. Hence, the build and code quality metrics like defects, vulnerabilities and complexity will provide insights and assurance to the operations teams on the software quality.

The code complexity of an application is shown in fig 9, which was planned to be released in to production. As the code complexity increases, maintainability decreases. The complexity, number of defects, number of build failures and vulnerabilities; indicate stability of the application. Based on number of incidents in production for this application in the past and corresponding quality parameters, provide stability insights to operations team.

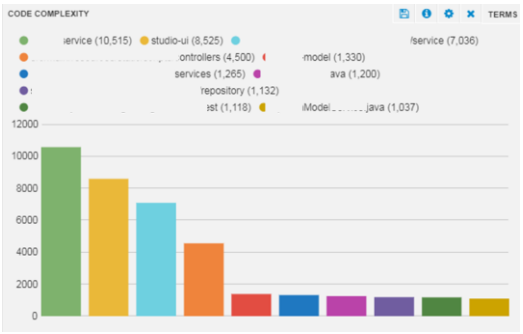

Fig. 9. Code complexity of the application

E. *Application security analytics*

A key concern from the operations team has always been lack of confidence on the application release that they receive. Some of the operations team started testing in their environment, even before they move to production. This consumes effort, resources and cost to the organization. Had it been a scenario in which, operations team has insights on the quality of development, some of such activities can be eliminated. Similarly, the security operations team would also like to scan and ensure no vulnerabilities exist (or) introduced in the application.

We have ingested application security testing tool results, both SAST (Static application security testing) and DAST (Dynamic application security testing) results which were run on a daily basis. As shown in fig 10, the trends provided interesting insights to operations team and security operations teams, where there is a drop in number of vulnerabilities.

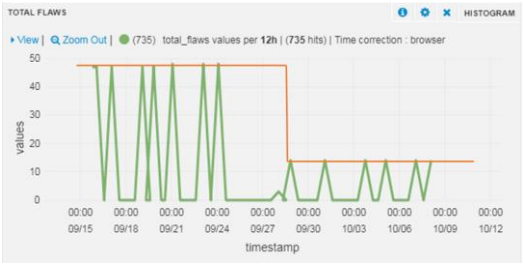

Fig. 10. Applications security vulnerabilities

F. *Infrastructure analytics*

Organizations monitor infrastructure for availability and performance issues of the servers and applications, by constantly measuring their services and resources (like CPU, memory, disk, I/O, network etc). These metrics provides insights on underlying server behavior. Since, the infra and application operations teams and corresponding tools are different; and often they operate in silos; the patterns of these metrics are not being interpreted in a holistic and cohesive way at the application level. Hence, most of the information required for incident diagnosis may not be considered for incident resolution and problem management. Hence, providing a view of the infrastructure layer metrics along with middleware and application layer metrics and logs will provide 360^0 view of system at the time of incident.

The various components of the application in scope are installed in 3 servers. A typical monitoring will highlight issue with one of the server and it may not highlight issue with the application, as it doesn't have the context of the application. The memory utilization pattern of all the three servers indicate a potential activity at the application level at highlighted time as shown in fig 11. On analysis, the application services went down during this period. Analysis across layers assists operations team in detecting the cause of anomaly quickly.

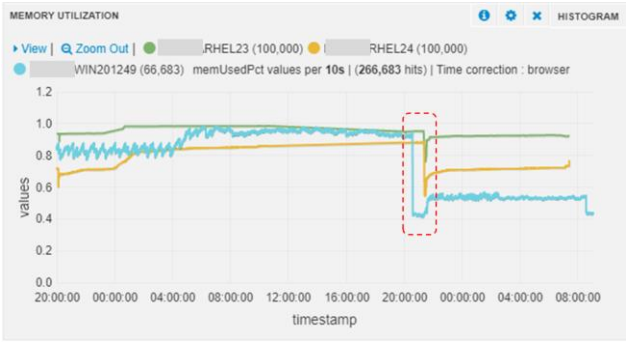

Fig. 11. Memory utilization of 3servers of an application

VI. Correlation and Anomaly Detection

Aggregation and visualization of logs and metrics alone for each metric is of little value to Dev and Ops teams, instead insights based on correlation of multiple logs and metrics will be useful. Hence, Dev and Ops teams started correlating across multiple metrics. Some such correlations are discussed below.

A. *Correlation of deployment structure and resource utilization metrics*

A large application was deployed on multiple servers and the operations team has been facing challenges with unknown issues. To understand the cause of the problem, the operations team collected the resource metrics (CPU, disk, memory) and the deployment architecture, which is shown in fig 12.

The application was deployed on 3 servers namely server 1, server 2 and server 3. The 25 deployable components were deployed on these 3 servers. The database server was deployed

server 2, which had 32 databases. The CPU utilization of server 1 and server 3 were normal. Server 2 has been experiencing frequent high CPU utilization. Thus it became obvious that the CPU on the server 2 has been consistently high, due to which the operations team was facing issues. As a result, the operations team has been recommended to modify deployment structure (or) increase resources for improved performance.

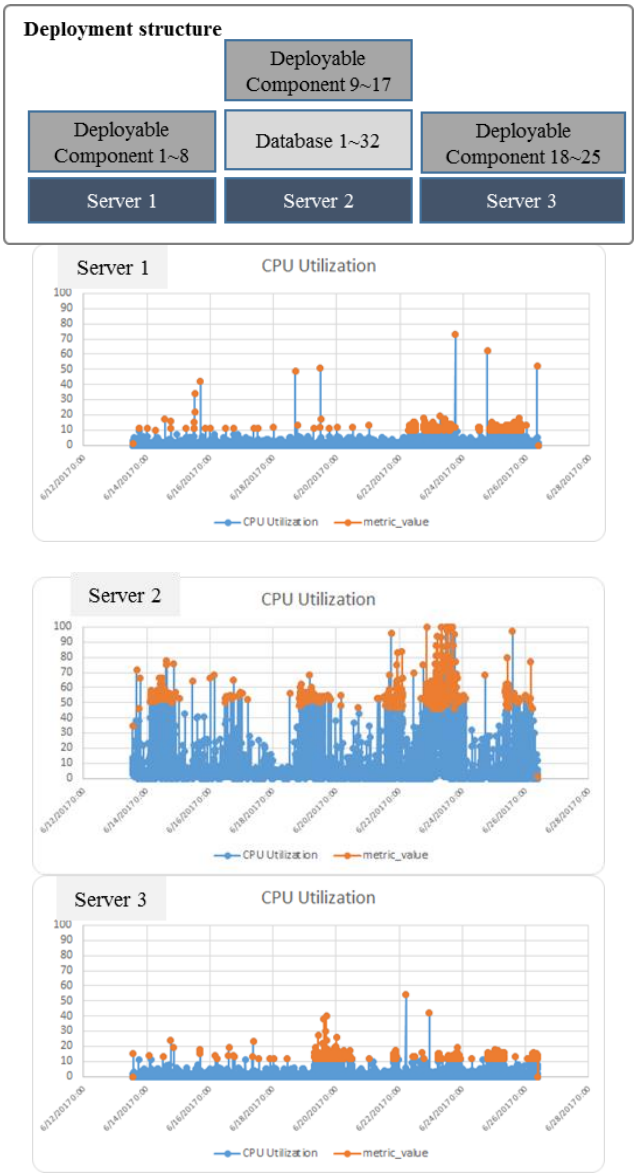

Fig. 12. Deployment architecture and resource utilization

B. *Application usage anomaly detection*

Analytics of an application in production environment has shown spike in its usage (as shown in fig 13). Considering normal behavior pattern and time of the day and week, the spike in usage is an anomaly. Operations team on quick selection of the time period on usage graph, it provided insights about - who has been accessing the application based on their IP address and the country from where the users are coming from. Operations team could quickly diagnose and arrive at a conclusion. In absence of such real-time analytics, most of these events goes unnoticed and a risk to an application and organization. Also, now it's possible for operations team to relate all possible metrics like IP address, city, pages accessed by these users and obtain insights. As a next step, we intend to build an alert management system from such incidents.

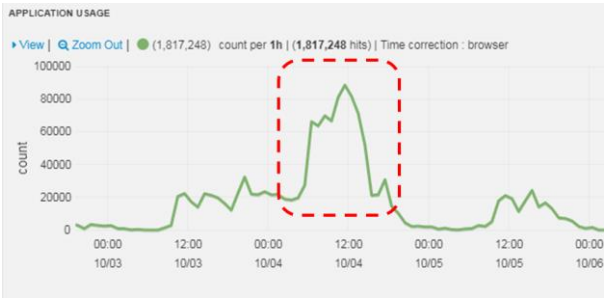

Fig. 13. Anomaly in application usage

C. *Correlation of errors, methods and programmer*

Correlation of most frequently called methods and modules to error prone modules and methods based on operations data of the application in scope became useful for development team to understand the characteristics of the errors. The methods which are called the most number of times and their corresponding error messages are shown in fig 14(a) and fig 14(b) respectively. This will provide guidance and prioritization for the programmer on the methods that needs to be refined and improve quality. Also, the type of errors occurring most of the times provides insights to development team on the category of errors they need to fix. Correlating erroneous methods with programmers of the method will provide insights on the programming abilities of the programmer and recommend necessary training/awareness to the programmer to avoid such errors. Where possible, define the code quality rules in static code analysis tools like sonarqube, so that such code quality issues are detected during the continuous integration stage.

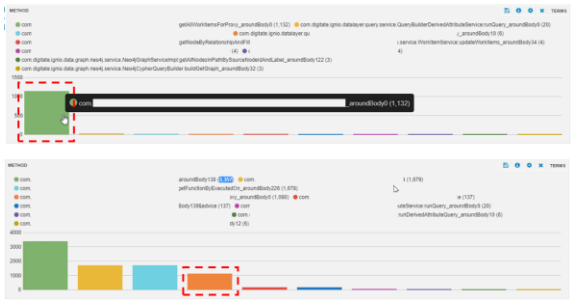

Fig. 14. (a) Errors in application logs by method (b) Total number of logs by method

D. *Correlation of performance and application complexity*

The modules that are performing slowly are correlated with code complexity. The module which has most number of the calls and takes longer processing time (fig 7) was the module that had high code complexity and defect density (fig 15).

These modules have become a bottleneck in the application. Based on these insights development team can now optimize their design to reduce the complexity, number of calls and processing time to improve overall quality of the software. Also, the complexity impacts the maintainability of the code for ops team.

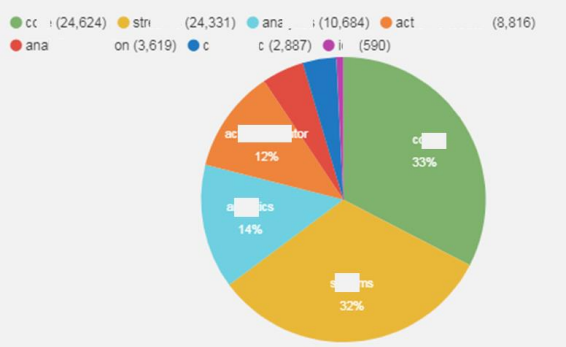

Fig. 15. Code complexity

E. Correlation of errors across layers

Applications in production generate errors/exceptions in multiple logs. At real-time it's humanly impossible to determine where the problem is by observing the logs, that too across multiple logs across layers. To assist them we have a log aggregation and correlation across layers as shown in fig 16. On clicking the specific type of error log like application log or technology log as shown in fig 17 and 18 respectively; operations team is able to quickly diagnose the problem location.

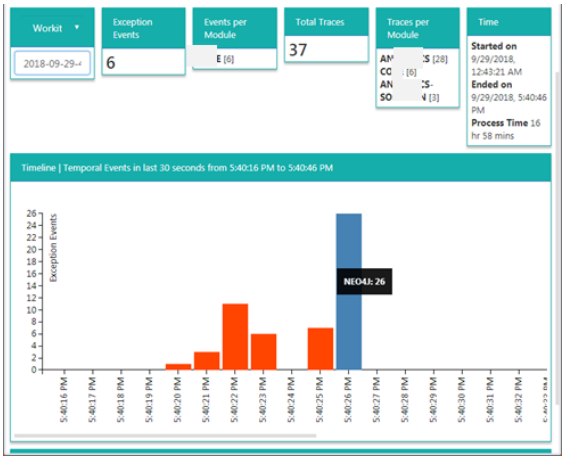

Fig. 16. Correlation of logs across layers

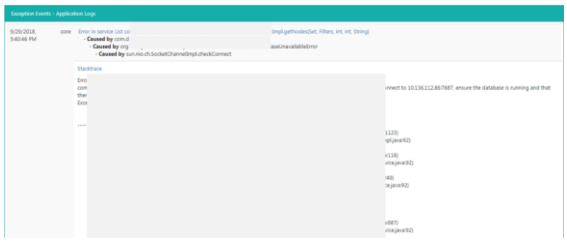

Fig. 17. Application log

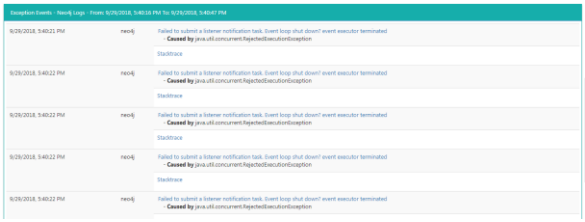

Fig. 18. Technology log

VII. LESSONS LEARNT

Some of the lessons learnt based on the challenges faced during DevOps analytics implementation are as follows:

(a) Log format: Different applications use different log formats. We started getting them modified by working with the respective application teams, to capture meaningful information for the purpose of analytics. Hence, for any organization embarking on to DevOps analytics, it becomes important to standardize the log format across applications to generate meaningful insights. Alternately, teams can adopt an approach based on application code instrumentation.

(b) Transaction details: Application in-scope for DevOps analytics is a large and complex application with multiple components (or) deployable packages. We had a challenge to trace a business transaction, end to end across all components. Hence, we needed a trace id or transaction id across all the application components for meaningful insights. We had to work with the application team to get this implemented. Applications are also required to maintain business transaction names, so that we can profile across interactions.

(c) Intelligent logging: Applications and underlying middleware create humongous logs and data. It increases data size and the processing that needs to be performed on these logs. To minimize this, we started logging transactions whose processing time is greater than a specified limit of 1 sec. This avoided superfluous logging.

(d) User id: Application usage analytics will be useful and meaningful when we capture user id. However, for certain applications this is not being enabled at httpd, due to security reasons. This needs to be discussed with the respective organization security team and enable id which is unique yet doesn't divulge enough details of the user.

(e) Tooling ecosystem: Different tools will provide different API's and each tool provides different attributes for a specific purpose. This means, we need to customize the model for each application depending on the type of tools that are being used. For instance, veracode as a SAST/DAST tool provides some information, which other tools like Fortify may not provide.

(f) Initiatives like this in organizations are always at a risk of death very soon after its initial success. Hence, to make it as a part of organization culture and refine the analytics from time to time, Dev and Ops teams should leverage analytics for diagnosis and decision making in day-to-day

activities in order for the learning to enrich the system. Thus the system can start addressing the most of the frequently asked questions [10]

(g) Typically deployment architecture is not being considered as a part of DevOps practice. We observed that the deployment architecture has an impact on the operations team and the number of alerts/incidents it generates. Considering this we recommend to include deployment architecture as a dimension in DevOps analytics book of knowledge.

VIII. CONCLUSION

DevOps analytics offers useful insights to (a) operations team based on data across layers (application, middleware and infrastructure) to resolve the incidents faster (b) application owner and operations team on the stability and quality of the release based on the data from development to operations (c) security operations team on the application vulnerabilities, resulting in to effort, resource and cost savings. Thus frequent delivery with minimal impact on application operations; and operations team ability to resolve incidents quickly, makes DevOps implementation effective. Considering the benefits that DevOps analytics offers by bringing transparency between the Dev and Ops teams, and its ability to provide insights near real-time, we intend to enrich it with additional data sources and leverage algorithms to identify anomalies and provide insights, thereby minimizing the dependency on experts in Dev and Ops teams.

IX. FUTURE WORK

As a next step, we intend to

(a) increase the scope of data sources to release management and incident management tools

(b) enhance visualization for faster resolution and highlight risk with the modules in the application.

(c) leverage machine learning algorithms to correlate, identify anomaly and provide insights to the development and operations team.

(d) anomaly detection across various metrics and logs; and alert the operations team

(e) integrate and institutionalize this as a part of development and support processes, in order to sustain this practice.

REFERENCES

[1] M. West, M. Sobejana, J.Herschmann and K.Mann, “Predicts 2017: Application development” Gartner report, November 2016.

[2] J. Thomson, L.Carvalho and A.Gillen, “IDC Futurescape: Worldwide developer and DevOps 2018 predictions”, November 2017.

[3] S. Sharma and B.Coyne, “DevOps for dummies”, 2nd Edition, John Wiley & sons, 2015.

[4] B. Snyder and B. Curtis, “Using analytics to guide improvement during an agile-DevOps transformation”, IEEE Software, vol. 35, no. 1, pp: 78-83, 2018

[5] V. Augustine, J.Hudepohl, P. Marcinczak, and Will Snipes, “Deploying software team analytics in a multinational organization”, IEEE Software,vol. 35, no. 1, pp:72-76, 2018

[6] G. Gousios,D. Safaric and J. Visser, "Streaming software analytics", Proceedings of 2nd International Workshop on BIG Data Software Engineering, pp:10-11, 2016

[7] J. Cito, “Developer Targeted Analytics: Supporting software development decisions with runtime information”, Proceedings of 31st International Conference on Automated Software Engineering (ASE), IEEE/ACM, pp:892-895, 2016

[8] O. Baysal, R. Holmes, and M. Godfrey, "Developer Dashboards: The Need for Qualitative Analytics", IEEE Software,vol. 30, no. 4, pp:46-52, 2013

[9] P. Johnson, “Searching under the Streetlight for Useful Software analytics”, IEEE Software,vol. 30, no. 4, pp:57-63, 2013

[10] T. Menzis and T. Zimmerman, “Software analytics: so what”, IEEE Software,vol. 30, no. 4, pp:31-37, 2013

Implementing Value Stream Mapping in a Scrum-based project - An Experience Report

Nayla Nasir, Nasir Mehmood Minhas
SERL Sweden, Blekinge Institute of Technology, Karlskrona, Sweden
naylanasirminhas@gmail.com, nasir.mehmood.minhas@bth.se

***Abstract*—The value stream mapping is one of the lean practices, that helps to visualize the whole process and identifies any bottlenecks affecting the flow. Proper management of the value stream can significantly contribute towards waste elimination by categorizing process activities to be either value adding or non value-adding. Lean development focuses on the value through the elimination of waste. Adding value through embracing change and customer satisfaction are also the benefits of Scrum.**

This study reports our experience regarding the implementation of VSM with Scrum. We followed the action research method, with an objective to see if VSM can contribute to the identification and reduction of wastes in a Scrum-based project.

We identified a noticeable amount of waste even with strict compliance to the Scrum practices. On the basis of identified waste, their root causes, and possible mitigation strategy we have proposed a future state map, that could help improve the productivity of the process. The results of our study are encouraging, and we suggest that adoption of VSM with Scrum could add more value to the Scrum-based projects.

Keywords: Scrum, Value Stream Mapping, Waste

I. Introduction

The focus of software development organizations is to develop high-quality software at a lower cost, faster pace, and maximum customer value. To achieve this, organizations have shifted their approach from classical development to iterative releases. In early 90's people started raising their voices to curtain lengthy processes and documentation in software development. Agile and lean are the methodologies which supported this concept. Both Agile and lean have their own well-defined set of principles. Some of the principles are shared among the two, at the same time each has its unique principles and practices [1]. The main difference between agile and lean is that agile is a bottom-up approach, whereas lean supports top-down process [2].

Scrum is a popular agile development method. It is well suited where requirements are random and complex [5]. It is described as a development process for small teams, which includes a series of time-boxed development phases, "sprints", which typically last from one to four weeks. Scrum value the customer, and encourage the participation of the customer in the sprint meetings [10]. For the overall process improvement, lean software development emphasizes two fundamental principles, namely identification of waste in the process and considering interactions between the individual parts of the software process from an end-to-end perspective. [8].

The overall goal of lean development is to achieve a continuous and smooth flow of production with maximum flexibility and minimum waste in the process. All activities and work products that do not contribute to the customer value are considered waste. Identifying and removing waste helps to focus more on the value creating activities [8]. Waste is an activity, which will not provide any value addition to the final product or customer aspects [12]. The use of lean principles and practices with agile is not new, lean practices are used for continuous improvement in the agile process. The use of Kanban (a concept related to lean) with Scrum is one of the examples [14].

One of the important lean practices is value stream mapping(VSM). It is the process of directly observing the flow of items, and summarizing them visually [16]. VSM helps managers to understand the current operational conditions and recognize improvement opportunities to maximize the performance towards perfection. Value stream mapping focuses on activities that add value to a product, at the same time it identifies the activities that do not add value [4].

This study briefly describes the main concepts associated with VSM and presents an analysis of VSM implementation in a Scrum-based project. The focus of the study is to see if the VSM can be beneficial when used in Scrum. In our project, by identifying the value-added and non value-added time in the current state of the process, we were able to identify and prioritize the various type of wastes. Furthermore, we have proposed the mitigation strategies to reduce the identified waste and thus suggested a future state of the process. Our experiment shows that the use of VSM with Scrum adds more value when comparing to the effort utilized on it.

The rest of this paper is organized as: Section II describes the background knowledge and concepts regarding value stream mapping. Section III elaborates on the method used for the study, it also enlists the research questions investigated in the report. VSM process execution is presented in Section IV. Section V presents the findings of the research. Finally, conclusions are provided in Section VI.

II. Background

This section presents the background knowledge and key concepts regarding the value stream mapping. The subsequent sections describe the following concepts: *A*) The value stream mapping, and its purpose, and *B*) How to implement VSM.

A. Value Stream Mapping

Womack et al., [15] define the value steam mapping as: *"The process of charting out or visually displaying a Value stream so that improvement activity can be effectively planned."*
Value stream mapping, if implemented properly, can have significant contributions towards waste identification and increased process efficiency [8]. To draw a value stream map, the concept of value and the value stream should be clearly understood.

What is Value? Value is the end-goal and therefore the establishment of value parameters at the start of a project is the key to achieve improved productivity and customer satisfaction. Emmitt et al., [4] suggests the following two characteristics of value:

- The perception of value is individual and personal, and is therefore subjective. Agreement of an objective best value for a group will differ from the individuals' perception of value.
- Values will change over time.

These characteristics highlight the complexity of value and thus emphasize on having a clear understanding and establishment of value before doing anything else.

The Value Stream Rother and Shook [11] define value stream to be *"all the actions (both value added and non-value added) currently required to carry a product through the main flows essential to that product: 1) the production flow from raw material to customer, and 2) the design flow from concept to launch"*

The Flow and Flow Items: To look for a bottleneck in a production system, it is important to understand what flows through that system i.e. the flow items. All work that flows through a software value stream is characterized by one, and only one, of the following flow items[6].

- Features and Defects: If we consider feature additions and defect fixes as the flow items. We can characterize work across all the people and teams in a value stream as applying to one of these units. Having visibility into every process, we could identify exactly how many people were involved in creating, deploying, and supporting a particular feature. The same goes for a defect fix. [6]
- Work on Risks: Another kind of work that is invisible to users and is pulled through the value stream is the work on risks. It includes the security, and compliance work that must be scheduled onto development backlogs. This work competes for priority against features and defects. It is not pulled by the customer because the customer usually can't see it until it is too late. [6].
- Debt Reduction: The fourth type of work that can be observed is debt reduction. If automation is not done to reduce infrastructure debt, it could impede the future ability to deliver features. This work tends to be pulled by software architects.[6].

How to Measure Flow? A number of metrics can be found in literature that can be used as measures of software delivery flow. They include lines of code, function points, work items, story points, deployments, and releases.
Each of them captures a notion of value flow from a different perspective, and has its limitations when used to depict end to end flow. [6]

Metrics for VSM: Following are the metrics that are associated Value stream mapping and can be used as a reflection on the current state of a process[13]:

- Takt time: Takt time is the rate at which a company must produce a product to satisfy customer demand.
- Cycle time: The time that elapses from the beginning of a process or operation until its done.
- Total cycle time: The total of all cycle times for each individual operation or cell in a value stream. Ideally, if there is no waste then the Total cycle time equals total value-added time.
- Queue time: The time that a work unit will wait for a downstream operation to be ready to work on it.
- Lead Time: Number of minutes, hours, or days that must be allowed for the completion of an operation or process, or must elapse before a desired action takes place.[3]

Purpose of Value Stream Mapping: Value stream management aims for perfection by maximizing flow [11]. It contributes to continuously refine and adjust the software process to improve its performance in terms of lead-time, quality of the software product and reduction of change requests [8]. VSM allows to see the whole by letting you visualize more than a single process level. It not only allows to identify the waste but also helps to see the sources of waste by highlighting the bottlenecks [11].

B. Implementing VSM

Value stream management is a process for planning and linking lean initiatives through systematic data capture and analysis. Value stream management consists of eight steps [13].

1) Commit to lean:
   The first step to improvement is willingness to change. Successful VSM implementation requires a commitment from management to maximize the visibility of operations and accept any changes as suggested by this process visualization activity.
2) Choose the value stream:
   A flow item must be selected as a target for improvement. This will define the scope of the VSM process.
3) Learn about lean:
   Having the knowledge of key lean concepts, such as maintaining a consistent work-flow, the lean waste and its types, and the importance of continuous improvement (kaizen) etc, will help to categorize the activities as value adding or non value adding, hence identifying the waste.
4) Map the current state:
   The current state is determined by gathering information on current practices, and a current state map is drawn by visually presenting this information.

5) Determine the lean metrics:
   On the basis of the current state map, the values for the lean metrics (i.e the lead time, the cycle time, the queue time) are derived to reflect the current efficiency of the process.
6) Map the future state:
   Highlighting the non value adding activities helps to identify and prioritize wastes. A future state map is presented after addressing the root causes of waste with an intention to improve the process.
   The Development of current and future state is an overlapping effort. Future state improvement ideas will come while working on current state, and working on future state may highlight important current state ideas that have been overlooked.
7) Create Kaizen plans:
   The next step is to devise plans for achieving the future state by incorporating the changes as suggested in the future state map.
8) Implement Kaizen plans:
   The final step is the implementation of Kaizen plans that describe how to achieve future state and how to cope with this transformation.

III. Research methodology

We adopted action research as main research method for this study. Action research is a participatory approach concerned with developing practical knowledge [9]. This methodology leads to a better understanding of practical and theoretical outcomes of a process through direct participation, rather than based on perceptions and interests of an external researcher. We carried out a Scrum based project with a small team of six members. The participants were assigned to the following roles, 1) Scrum master, 2) product owner, 3) business analyst, 4) developers, and 5) QA. All the members of the team are the students of masters in software engineering.
The research activity is based on research questions presented in Table I.

TABLE I
Research questions and Objectives.

ID	Research Question	Objective
RQ1	Is VSM capable of visualizing the current state of a Scrum-based project?	To investigate if VSM can identify the current state (waste) in a Scrum-based project.
RQ2	What is the impact of using VSM in a Scrum-based project?	To determine the effectiveness of VSM, regarding waste reduction in the future state of a Scrum-based project.

A. Methods Used:

To extract knowledge about the VSM, we carried out a literature review and studied literature presented in various research articles. After having a clear understanding of VSM, we utilized the acquired knowledge to observe the outcomes of value stream management in a practical scenario. The steps taken for this study are illustrated in Figure 1.

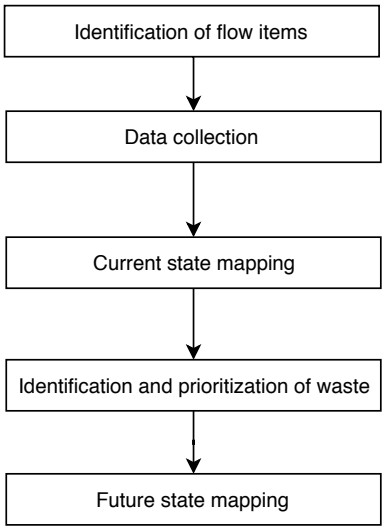

Fig. 1. Steps followed to implement VSM

The VSM activity was performed on a Scrum based process. The first author of the report was the member of the team executing the process, and served as a Scrum Master. There were two iterations in the process, but the perception of value changed for the customer after the demo of first sprint and the project scope was redefined. The tasks delivered at the end of the fist sprint were rendered as waste as the customer was not going to utilize the outcomes of that work. The sprint under observation consisted of 12 working days, and the value stream was mapped for the same period. A working day was considered to be of four hours as the team members were allocated fifty percent of their capacity to this project.

IV. Process Execution and Results

For the implementation of VSM, we followed the steps defined by Tapping et al. [13]. A brief description of these steps is presented in Section II-B. The steps adopted for our experiment are presented in Figure 1.

A. Identification of Flow Items

The first step is to determine the flow items for the value stream and use some measures for them. Kersten [6] defines four types of flow items (Features, Defects, Work on Risk, and Debt Reduction) that can be used for value stream mapping. Further, he emphasizes that only one of these four can be selected to draw a value stream. The flow items for the intended VSM are the features prescribed by the user stories, and these features are represented as tasks (PF-23 to PF-35), in the sprint backlog.

B. Data Collection:

An online Scrum management tool was used as a virtual Scrum board, for visibility and availability purposes. This tool was also used for time based data collection. All the team

TABLE II
SPRINT EXECUTION DETAIL

Sprint 2(26/09/2018 to 10/10/2018) ……(12 Working Days, Each working day=4 hours)									
DATE	ANALYSIS	Task ID	DESIGN	DEVLOPMENT		CODE REVIEW		QA	DONE
				CODING	READY TO REVIEW	IN PROCESS	WAITING FOR QA		
26/09/2018	Tasks defined, priorotized, assigned and added to sprint Backlog (4 hrs)								
27/10/2018		PF-23	PF-23 3:00 hr						
28/09/2018		PF-25 PF-27 PF-28	F-25 01:00 hr	PF-23 1:00 hr PF-27 00:30 hr PF-28 00:30 hr					
10/01/18		PF-34 PF-26 PF-30 PF-29 PF-33	PF-34 3:00 hr	PF-25 01:00 hr PF-26 00:30 hr PF-30 00:30hr PF-29 00:30 hr PF-33 00:30 hr	PF-23 PF-27 PF-28 PF-26 PF-30				
10/02/18				PF-29 00:30 hr PF-33 00:30 hr PF-25 01:00 hr	PF-23 PF-27 PF-28 PF-26 PF-30				
10/03/18		PF-31		PF-31 02:00 hr PF-25 00:00 hr PF-33 00:00 hr PF-29 01:00hr		PF-23 01:00 hr PF=27 00:30Hr PF-28 01:00hr PF=26 00:30Hr exception 01:00hr			
10/04/18		PF-32		PF-32 01:00 hr PF-31 00:00 hr PF-25 01:00 hr PF-33 00:00 hr PF-29 00:30hr PF-30 00:30hr	PF-28		PF-26 PF-23 PF-27		
10/05/18		PF-32		PF-32 00:00 hr PF-31 00:00 hr PF-25 00:00 hr PF-33 00:00 hr PF-30 00:00hr	PF-28 PF-29		PF-26 PF-23 PF-27		
10/08/18				PF-32 00:30 hr PF-31 00:00 hr PF-25 00:30 hr PF-33 00:30 hr PF-30 00:30hr	PF-28 PF-29			PF=26 01:00Hr PF-23 01:00 hr PF=27 01:00Hr	
10/09/18				PF-32 00:30 hr PF-31 00:30 hr PF-25 00:00 hr PF-33 00:30 hr PF-30 00:30hr	PF-28 PF-29 PF-32 PF-31 PF-33				PF-26 PF-23 PF-27
10/10/18				PF-25 01:00 hr	PF-30 PF-25	PF-32 00:30 hr PF-31 00:30 hr PF-33 00:30 hr PF-30 00:30hr PF-29 00:30hr PF-28 00:30 hr PF-25 00:30 hr		PF-32 00:30 hr PF-31 00:30 hr PF-33 00:30 hr	PF-32 PF-31
10/11/18								PF-29 00:30hr PF-28 01:00 hr PF-30 00:30hr PF-25 01:00 hr	PF-33 PF-29 PF-28 PF-30 PF-25

members reported the actual time using the tool every time they worked on a task. This input was further validated in the daily stand-up meetings. The data on actual time spent on each task was also maintained in an excel file, which depicts the day to day progress (See Table II). The tasks are color-coded which makes it easy to visualize the progress of each task across different operations in the process. Time spent on each task was recorded, and if there were multiple tasks in an operation on a particular day, time spent on task switching was also recorded. There are columns to show the time for which tasks waited until they were retrieved by the next operation.

Presenting a detailed picture of sprint execution,the excel file can be used to conduct an analysis and identify the value added Time (VAT), as well as the non value added time (NVAT) per task, for each working day. The Data Collected during this phase is summarized in Table III.

C. Current State Mapping

This activity is completed in two steps, firstly on the basis of analyzed VAT and NVAT, we calculated the required metrics for VSM and then we generated the current state map.

a) VSM Metrics:

1) **Cycle Time:** The cycle time is a measure of the time required to complete one cycle of an operation or to complete a function. It consists of both value added and non-value added time within an operation. Value added time (VAT) is referred to the time spent on the activities that add some value to the customer. Whereas, non-value added time (NVAT) is the one that is spent on activities that could be essential for the process but adds no value to the customer. The cycle time is calculated for each task and the cycle time for the individual

TABLE III
CYCLE TIMES AND LEAD TIMES FOR THE CURRENT STATE

Tasks ID	Analysis Cycle time NVAT	Analysis Cycle time VAT	Design Cycle time NVAT	Design Cycle time VAT	Development Cycle time NVAT	Development Cycle time VAT	Queue Time	Code Review Cycle time NVAT	Code Review Cycle time VAT	Queue Time	Quality assurance Cycle time NVAT	Quality assurance Cycle time VAT	Lead time
PF-23	0,08	0,25	2	2	0,66	1,00	1,60	0,00	1,00	1,33	3,00	1,00	13,92
PF-25	0,08	0,25	0	4	1,88	5,00	0,00	0,00	0,50	0,00	0,50	1,00	13,21
PF-26	0,08	0,25	0	0	0,20	0,50	1,60	0,00	1,00	1,33	0,33	1,00	6,29
PF-27	0,08	0,25	0	0	0,66	0,50	1,60	0,00	0,50	1,33	0,33	1,00	6,26
PF-28	0,08	0,25	0	0	0,66	0,50	8,67	0,17	1,50	0,00	0,25	0,50	12,58
PF-29	0,08	0,25	0	0	1,28	2,50	5,00	0,17	0,50	0,00	0,25	1,00	11,03
PF-30	0,08	0,25	0	0	1,47	1,50	1,60	0,17	1,50	0,00	0,25	0,50	7,31
PF-31	0,08	0,25	0	0	1,02	2,50	0,00	0,17	0,50	0,00	0,83	0,50	5,85
PF-32	0,08	0,25	0	0	0,77	2,00	0,00	0,17	0,50	0,00	0,83	0,50	5,10
PF-33	0,08	0,25	0	0	1,88	2,00	0,00	0,17	0,50	0,00	0,83	0,50	6,21
PF-34	0,34	0	4	0	No progress here for these tasks								4,34
PF-35	0,34	0	0	0									0,34
Total	1,48	2,5	6	6	10,477	18	20,07	1,00	8	4,00	7,41	7,50	92,43
Cycle time(CT) per opertion	3,98		12		28,48			9,00			14,91		68,37
Average CT/operation=	0,33		4		2,848			0,90			1,49		
Total VAT for the system(hrs)=	42,00		Takt time= Total working hours/no. of tasks=		48/12=	4 hrs							
Total NVAT for the system (hrs)=	26,37												
Total Queue time for the system (hrs)=	24,07		Product cycle efficiency=		45%								

operations is presented as a sum of cycle time for each task. For example,referring to Table III, the analysis was performed by the product owner in one working day (i.e. 4 Man-Hrs). The cycle time calculated to be 4 hours, of which 1.48-hour is the NVAT and 2.5-hour is the VAT. In Table III VAT for PF-34 and PF-35 is zero because these tasks were unfinished from the customer perspective as these tasks did not progress to the subsequent operations (i.e., Development to QA). Cycle times for other operations is Design 12 hours, Development 28.48 hours, Code review 9 hours, and Quality Assurance 14.91 hours.

2) **Average Cycle Time:** As the cycle time for each task varied depending upon the complexity of the task, so an average cycle time of an operation is calculated by dividing the cycle time of the operation by the number of tasks. For instance, in the development operation the cycle time was 28.47 and the total number of tasks in the operation were 10. So, the average cycle time is calculated to be 2.8. Average cycle time for each operation is presented in Table III.
3) **Total Cycle Time**:
   The Total cycle time for the system is calculated by adding the cycle time of all operations (analysis to quality assurance). Total average cycle time calculated for our system is 68.37 hours.
4) **Queue Time:** Queue time is the time for which a task waits between two successive operations. There were two queues in our system, one between development and code review and the other was between code Review and Quality Assurance. The task with the highest queue time is PF-28, that waited for code review for 13.67 hours, because of its interdependence with two tasks (i.e., PF-29 and PF-30). The Queue time is calculated for each task, and the total queue time between development and code review is 20.07 hours, and the total queue time between code review and quality assurance is 4 hours. The queue time for the whole process is 24.07 hours.
5) **Lead time:** Lead time is the measure of time required to perform all the activities from the initiation of a task until it is completed. The lead time includes both the queue time and the cycle time. If there is no waiting time between the operations , then the lead time equals the total cycle time for the process.
   In our project the lead time is calculated to be 92.43, it is the sum of the total cycle time and the total queue time.
6) **Takt Time**: The Takt time is used to synchronize the rate of production with the rate of demand. It establishes a cadence for the work flow, and contributes towards efficiency through continuous adjustments. It is computed by dividing the available time by the number of tasks. There were 12 working days with four hours for each day (i.e., 48 work hours), and tasks to be accomplished were 12. The Takt time is calculated to be 4 hours (total work hours / total number of tasks). That indicates that each task needs to be worked by the team for four hours.

b) The Current State Map: A Current State Map was generated based on the data presented in Table III, and is shown in Figure 2. We considered the customer as the start point for our VSM because the first step in value stream management is to establish a clear perception of the value as defined by the customer. The process of implementing the current state map was initiated as soon as an agreement was made on user stories. Each cell of the current state map represents an operation. The number of people carrying out the operation is also mentioned within each cell. An activity is considered to be value adding if it contributes to customer value, and all non value adding activities are considered a waste. The value added time (VAT) for an operation is represented by the trough of the time-line and the crust shows the waste represented as a sum of NVAT and queue time. The

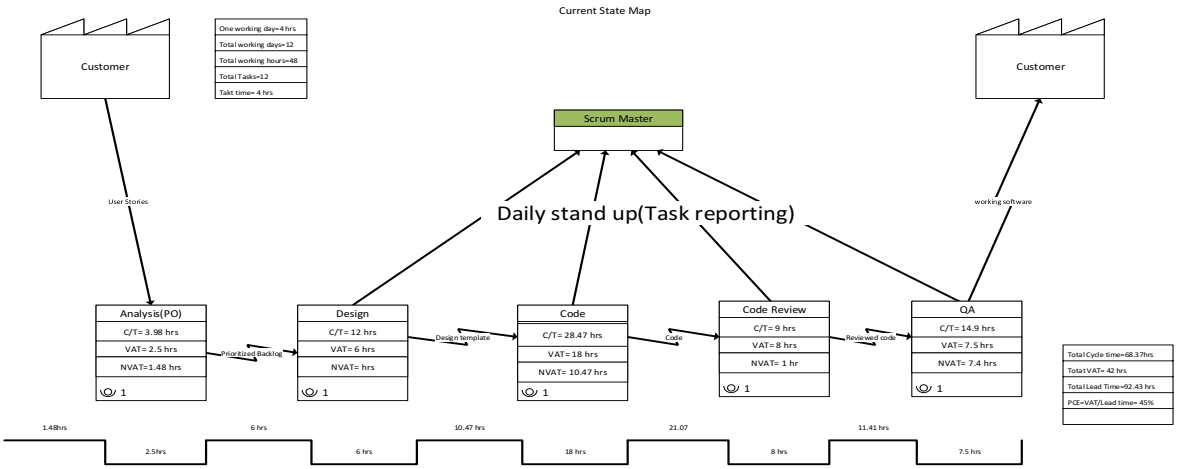

Fig. 2. Current State Map

TABLE IV
WASTE IDENTIFIED AND PRIORITIZED IN TERMS OF IMPACT

Priority	Type of Waste	Task Involved	Impact	Root Cause	Mitigation Strategy
1	Queue Time (Delays & Movements)	PF-23, PF-26, PF-27, PF-28, PF-30	24.07 Hrs.	i)Upstream operations are producing items at an inconsistent rate and the downstream operations cannot pull them efficiently, creating a push. ii) Interdependence of tasks	i) Merge operations where possible (e.g., coding and coding review) ii) Reduce task inter-dependency.
2	Task Switching	PF-25, PF-29, PF-30, PF-31, PF-32, PF-33	21.69 Hrs.	Multitasking within operations due to interdependence of tasks	Use work in progress (WIP) limit
3	Partially done work	PF-34, PF-35	4.68 Hrs.	Time constraint	Eliminate higher priority waste
4	Defects	PF-30	0.5 Hrs.	Exception handling was not addressed during development	Pair Programming

flow of information between cells is in electronic form.The takt time , lead time and process cycle efficiency are also presented in the CSM. The purpose of VSM is to visualize the complete value stream, and it should represent the complete work flow, right from the inception of demand to the delivery of the product to the customer. So customer is considered to be the endpoint for the VSM discussed in this study.

D. Identification of Waste

There are seven types of waste in Lean, as identified by Taiichi Ohno – the founder of the Toyota production process [7]. Cawley et al. [1] presented a connection between waste in lean manufacturing and waste in Lean software development. They listed the following wastes for the software development: 1) Extra features/code, 2) Delays, 3) Task switching, 4) Extra processes, 5) Partially done work, 6) Movements, 7) Defects, and 8) Unused employee creativity. On the basis of this categorization, we identified four waste categories in our project, namely queue time (delays and movement), task switching, partially done work, and defects. Identified waste categories along with the tasks involved are presented in Table IV. We have prioritized the waste on the basis of impact (in terms of time). In our project the top priority waste is the queue time with 24.07 hours, and the lowest priority waste is defects with 0.5 hours. We also identified the causes of the waste regarding our project (See TableIV). Finally, we listed the possible solutions to reduce the identified waste.

E. Future State Mapping

The future state map suggests improvements in the process. The goal is to prepare a possible course of action to address the root causes of the waste found in the current state of the process. In the light of solutions presented in Table IV, we reviewed the current state and suggested an ideal state of sprint execution with significant reduction in lead time, queue time and NVAT. For instance, to eliminate Queue time we can merge two operations, i.e., Code and Code Review by using the concept of pair programming. This will help to rectify errors during the development operation, and the delays between these two operations can be removed. It will also help to reduce the defect rate. Similarly, our next priority was to address the issue of task switching. For this, we can define the limit for work in progress(WIP) within an operation at any given time. Limiting work in progress can regularize the flow and will result in less multitasking and a notable reduction in NVAT within the operations. We propose to set this limit to a maximum of three tasks in development and quality assurance. It will create a pull by allowing the retrieval of a task when a successive operation is ready to work on it. Subsequently, the queue time between operations can be reduced to zero.

TABLE V
CYCLE TIME AND LEAD TIME FOR FUTURE STATE

Tasks ID	Analysis Cycle time NVAT	Analysis Cycle time VAT	Design Cycle time NVAT	Design Cycle time VAT	Code+ Review Cycle time NVAT	Code+ Review Cycle time VAT	Queue Time	Quality assurance Cycle time NVAT	Quality assurance Cycle time VAT	Lead time
PF-23	0,08	0,25	2	2	0,5	2	0	0	1	7,83
PF-25	0,08	0,25	0	3,5	1,3	5,5	0	0,33	1	11,96
PF-26	0,08	0,25	0	0	0,3	1	0	0,33	1	2,96
PF-27	0,08	0,25	0	0	0,3	1	0	0,33	1	2,96
PF-28	0,08	0,25	0	0	0,12	1,5	0	0	0,5	2,45
PF-29	0,08	0,25	0	0	0,25	3	0	0,08	1	4,66
PF-30	0,08	0,25	0	0	0,13	3	0	0	0,5	3,955
PF-31	0,08	0,25	0	0	0,25	3	0	0,08	0,5	4,16
PF-32	0,08	0,25	0	0	0,75	3	0	0,08	0,5	4,66
PF-33	0,08	0,25	0	0	0,75	2,5	0	0	0,5	4,08
PF-34	0,08	0,25	0	4	0,75	3	0	0	1	9,08
PF-35	0,08	0,25	0	0	0,5	2	0	0	1	3,83
Total	0,96	3	2	9,5	5,90	30,5	0	1,23	9,5	62,585
Cycle time(CT) per opertion	3,96		11,5		36,40			10,73		62,585
Average CT/operation=		0,33	3,83		3,03			0,89		8,090

Total VAT for the system(hrs)=	52,5	Takt time= Total working hours/no. of tasks=	4,00
Total NVAT for the system (hrs)=	10,09		
Total Queue time for the system (hrs)=	0	Product cycle efficiency=	84%

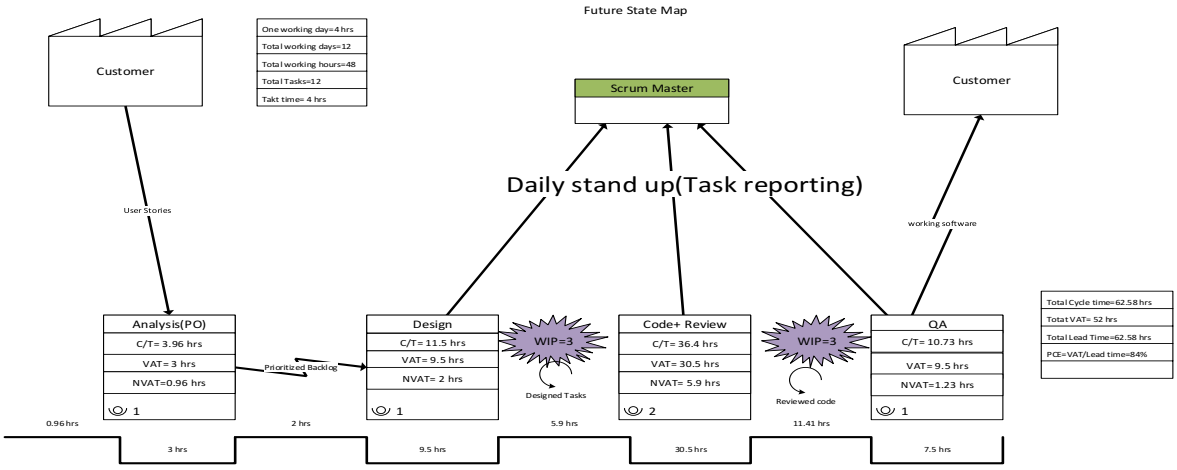

Fig. 3. Future state map

Finally, there will be no partially done work as there will be enough time to complete all the tasks. The suggested changes are presented in the Table V. It can be seen that the lead time is equal to the total cycle time as there are no delays between the operations. The proposed future state map is provided in Figure 3. The map includes calculations for takt time, lead time, and process cycle efficiency. The customer is the start, and the end point of the future state map as well.The cells symbolize the operations within our process, along with the details of cycle time and the number of workers for each operation. The Kaizen burst symbol represents suggested improvements. We used it between design and development, and then development and QA, to represent the improvements we have proposed in the form of WIP. Also, the flow between these operations is represented by a pull symbol instead of a push.

V. DISCUSSION

RQ1: IS VSM capable of visualizing the current state of a Scrum-based project?

The first objective of this study was to investigate the capability of VSM regarding the identification of current state (waste) in a Scrum-based project. The current state map presents the current set of activities been performed during a process, and is identified on the basis of detailed study of the actual process. We applied the action research method and carried out a project with a team of six people practicing Scrum as their development methodology. Being aware of the overall process flow, the Scrum Master (first author of the study) was responsible for value stream management. Required data for VSM was extracted from an online Scrum board with a little effort, and its validation was the part of every day stand-up. The detailed analysis of activities in terms of time, and value to the customer, helped to categorize them as value adding or

non value adding. Also some bottlenecks were highlighted that affected the overall productivity and increased the lead time for the process . The data of the current state is presented in the Table III, and visual representation of the current state is presented in Figure 2.

The current state map helped us to recognize the waste within and between the operations. We categorized the waste by using the list of waste provided by Cawley et al., [1]. The current process cycle efficiency is 45%, which could be improved by working on wastes. From our experience, we can conclude that implementation of VSM in a Scrum-based project does not require any additional effort, as most of the required data is available within the Scrum management process. Furthermore, we infer that VSM could be utilized with Scrum without affecting its practices.

RQ2: What is the impact of using VSM in a Scrum-based project?

The second objective of the study was to determine the effectiveness of the VSM, regarding waste reduction in the future state of a Scrum-based project. VSM not only helped us to identify the bottlenecks in the process, but also highlighted the root causes of the waste and allowed us to prepare a mitigation strategy. Table IV presents the prioritization of identified waste along with the root causes and mitigation strategy for waste reduction in each category. By utilizing the proposed mitigation strategy, we have created a future state map for the process. A set of data was created for an ideal state of sprint execution, see Table V, and this data was used to propose the future state. The visual representation of future state is shown in Figure 3. The future state map, shows a significant reduction in the lead time (i.e.92.43 hours to 62.58 hours), while also completing the partially done tasks in the current state of the process. This implies more productivity and thus a more satisfied customer, which is a primary goal of agile software development.

The waste is also reduced (i.e., 50.44 hours to 10.09 hours) by utilizing the team to maximize the time spent on value adding activities and eliminating the waiting time between process. The process cycle efficiency has improved visibly from 45% to 84% which implies a better utilization of available resources. The results we obtained by the implementation of VSM in our project are encouraging. Considering the difference, while comparing wastes and productivity in the current and the future state of the process, it can be stated that VSM can serve as an effective method to improve the efficiency of a Scrum team to an even higher level of performance. Value stream management allowed the team to realize impediments in the continuous flow of tasks, and to discover new ways of becoming more effective.

VI. Conclusion

This paper presents a research activity employing the value stream management to determine its effectiveness as a waste reduction mechanism in Scrum. It was inferred that VSM can be effectively used to reflect the overall flow, to identify the bottlenecks and to distinguish between value adding and non value adding activities.

Significant amount of waste was identified, along with its root causes. Based on this knowledge, mitigation strategies were defined to reduce or eliminate waste so as to increase efficiency, and to deliver tasks with better quality and less lead time. Addressing the wastes can contribute towards sustainable and more predictable development cycles and can enhance the Scrum experience. It was also observed that implementation of VSM in a Scrum based project was a fruitful activity as it requires very little effort when compared to its contribution to overall process improvement.

Based on current state of the process, we have proposed a future state map. Our next focus will be to identify the steps needed to develop the kaizen plans for the implementation of the future state of our process. And then to measure the success as achieved by the implementation of these plans in a practical scenario.

Acknowledgement

The authors would like to thank all the participants of the project. This work has been supported by EASE, the Industrial Excellence Centre for Embedded Applications Software Engineering (reference number 2015-03235).

References

[1] Oisín Cawley, Xiaofeng Wang, and Ita Richardson. Lean software development–what exactly are we talking about? In *Lean Enterprise Software and Systems*, pages 16–31. Springer, 2013.

[2] Robert N Charette. Challenging the fundamental notions of software development. *Cutter Consortium, Executive Rep*, 4, 2003.

[3] Business Dictionary. Lead Time. http://www.businessdictionary.com/definition/lead-time.html, 2018. [Online; accessed 10-oct-2018].

[4] Stephen Emmitt, Dag Sander, and Anders Kirk Christoffersen. Implementing value through lean design management. In *Proceedings of the 12th International Conference*, pages 361–374, 2004.

[5] Harleen K Flora and Swati V Chande. A systematic study on agile software development methodologies and practices. *International Journal of Computer Science and Information Technologies*, 5(3):3626–3637, 2014.

[6] Mik Kersten. What flows through a software value stream? *IEEE Software*, 35(4):8–11, 2018.

[7] Taiichi Ohno. *Toyota production system: beyond large-scale production*. crc Press, 1988.

[8] Kai Petersen and Claes Wohlin. Software process improvement through the lean measurement (spi-leam) method. *Journal of systems and software*, 83(7):1275–1287, 2010.

[9] Peter Reason and Hilary Bradbury. *Handbook of action research: Participative inquiry and practice*. Sage, 2001.

[10] Linda Rising and Norman S Janoff. The scrum software development process for small teams. *IEEE software*, 17(4):26–32, 2000.

[11] Mike Rother and John Shook. Learning to see. *Lean Enterprise Institute, Cambridge, MA*, 1999.

[12] RS Russell and BW Taylor. Operations management, 2nd edn, 1999.

[13] Don Tapping, Tom Luyster, and Tom Shuker. *Value stream management: Eight steps to planning, mapping, and sustaining lean improvements*. Productivity Press, 2002.

[14] Xiaofeng Wang, Kieran Conboy, and Oisin Cawley. "leagile" software development: An experience report analysis of the application of lean approaches in agile software development. *Journal of Systems and Software*, 85(6):1287–1299, 2012.

[15] James P Womack, Arthur P Byrne, Orest J Fiume, Gary S Kaplan, and John Toussaint. Going lean in health care. *Cambridge, MA: Institute for Healthcare Improvement*, 2005.

[16] Jim Womack and Dan Jones. *Seeing the whole value stream*. Lean Enterprise Institute, 2011.

Prioritization in Automotive Software Testing: Systematic Literature Review

Ankush Dadwal, Hironori Washizaki, Yoshiaki Fukazawa
Department of Computer Science and Engineering
Waseda University
Tokyo, Japan
ankush.dadwal@toki.waseda.jp

Takahiro Iida, Masashi Mizoguchi, Kentaro Yoshimura
Control Platform Research Department
Center for Technology Innovation - Controls
Hitachi, Ltd. Research & Development Group
takahiro.iida.ac@hitachi.com

***Abstract*—Automotive Software Testing is a vital part of the automotive systems development process. Not identifying the critical safety issues and failures of such systems can have serious or even fatal consequences. As the number of embedded systems and technologies increases, testing all components becomes more challenging. Although testing is expensive, it is important to reduce bugs in an early stage to maintain safety and to avoid recalls. Hence, the testing time should be reduced without impacting the reliability. Several studies and surveys have prioritized Automotive Software Testing to increase its effectiveness. The main goals of this study are to identify: (i) the publication trends of prioritization in Automotive Software Testing, (ii) which methods are used to prioritize Automotive Software Testing, (iii) the distribution of studies based on the quality evaluation, and (iv) how existing research on prioritization helps optimize Automotive Software Testing.**

***Index Terms*—Automotive Software Testing, Prioritizing, Systematic Literature Review**

I. Introduction

Currently the automotive industry is undergoing a major transition. Automakers have been adding new functions and systems to meet the market's demand for an ever-growing amount of software-intensive functions. However, these new functions and systems have some negative aspects. One is that automakers must enhance their testing techniques because vehicle complexity is increasing. Testing typically consumes more than half of all development costs [4]. While testing a single software system is difficult, testing without prioritization is even more challenging due to the exponential number of products and the number of features. Today, software determines more than 90% of the functionality in automotive systems and software components are no longer handwritten [6].

Test case prioritization is a method to prioritize and schedule test cases. In this technique, test cases are run in the order of priority to minimize time, cost, and effort during the software testing phase. Every organization has its own methods to prioritize test cases. The automotive safety standard ISO26262 requires extensive testing with numerous test cases. To achieve a high productivity, the availability of quality assurance systems must be high [7].

Herein we use a systematic literature review to evaluate relevant publications on prioritization in the automotive industry. A systematic review aims to assess scientific papers in order to group concepts around a topic. Through analysis criteria, it allows the quality of research to be evaluated. Herein the system review aims to identify common techniques in automotive testing and to define new challenges.

The paper is structured as follows. Section II describes related works. The systematic literature review approach is detailed in Section III. Section IV presents the results obtained from the systematic review. Section V addresses potential threats to validity. Finally, Section VI lists the conclusions and the definitions for future work.

II. Related Works

Automakers have experienced the impact of the evolution of technology on automotive testing. Today, testing all systems manually is not only cost-intensive and time-consuming but nearly impossible. Automating the testing phase would significantly reduce the cost of software development [14]. Literature about the prioritization efforts in the automobile industry is scarce. Herein we focus on known techniques and their applicability to the investigated domain.

In the past few decades, numerous studies [5], [6], [7], [8], [9], [10], [13], [15], [16], [17], [19], [20], [24], [27] have demonstrated that vehicles are becoming increasingly more complex and more connected. For example, an empirical study, which aimed to investigate the potential regarding quality improvements and cost savings, employed data from 13 industry case studies as part of a three-year large-scale research project. This study identified major goals and strategies associated with (integrated) model-based analysis and testing as well as evaluated the improvements achieved [25].

The only study we found that reviews the literature about the benefits and the limitations of Automated Software Testing is presented by Mantyla et al, [3] (2012). Their review, which included 25 works, tries to close the gap by investigating academic and practitioner views on software testing regarding the benefits and the limits of test automation. They found that while benefits often come from stronger sources of evidence (experiments and case studies), limitations are more frequently reported in experience reports. Second, they conducted a survey of the practitioners' view. The results showed that the main benefits of test automation are reusability, repeatability, and reduced burden in test executions. Of the respondents,

45% agreed that the available tools are a poor fit for their needs and 80% disagreed with the vision that automated testing would fully replace manual testing.

III. Methodology

We started the systematic literature review by specifying our scope and searching only documents in the domain of automotive software testing that discuss issues related to prioritization in the field of testing. Topics that focus only on software testing without prioritizing the test cases are excluded. In this research, we followed the guidelines suggested in papers [1], [2] . This method is divided into three steps.

A. Research Questions

This study strives to answer the following research questions:

RQ1. What are the **publication trends** of *prioritization in Automotive Software Testing*?
This research question should characterize the interest and ongoing research on this topic. Additionally, it will identify relevant venues where results are being published and the contributions over time.

RQ2. What are the **methods used** for *prioritization in Automotive Software Testing*?
This research question should elucidate the different methods used for prioritization in Automotive Software Testing. The goal here is to determine the main methods and tools used by researchers.

RQ3. How are the studies distributed based on a **quality evaluation** of *prioritization in Automotive Software Testing*?
This research question should reveal the quality distribution of the selected primary studies and evaluate them accordingly.

RQ4. How does **existing research** on prioritization help with the *optimization of Automotive Software Testing*?
This research question should classify existing and future research on prioritization in Automotive Software Testing and assess current research gaps. This is the most important and challenging question as it aims to compile problems that have yet to solved.

B. Search and Selection Process

The search and selection process is a multi-stage process (Fig. 1). This multi-stage process allows us to fully control the number and characteristics of the studies that are considered during various stages.

As mentioned in [1] and [2], we used three of the largest scientific databases and indexing systems in software engineering: ACM Digital Library, IEEE Xplore, and Scopus. These were selected because they are common, effective in systematic literature reviews in software engineering, and capable of exporting the search results. Further, these databases provide mechanisms to perform keyword searches. We did not specify a fixed time frame when conducting the search. To cover as many significant studies as possible, the systematic literature search query was very generic and considered the object of our research (i.e., Prioritization in Automotive Software Testing).

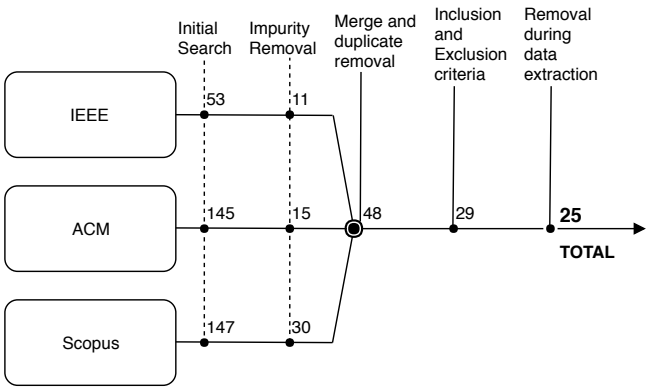

Fig. 1. Paper selection process

1) **Initial Search***:* We performed a search in three of the largest and most complete scientific databases and indexing systems in software engineering: ACM Digital Library, IEEE Xplore, and Scopus. We searched these databases using a search string that included the important keywords in our four research questions. Further, we augmented the keywords with their synonyms, producing the following search string:

((”automobile” OR ”automotive” OR ”car”)
AND
(”software” OR ”program” OR ”code”)
AND
(”prioritization” OR ”priority” OR ”case selection”)
AND
(test*))

For consistency, we executed the query on titles, abstracts, and keywords of papers in all the data sources at any time and any subject area.

2) **Impurity Removal***:* Due to the nature of the involved data sources, the search results included some elements that were clearly not research papers such as abstracts, international standards, textbooks, etc. In this stage, we manually removed these results.

3) **Merge and Duplicate Removal***:* Here we combined all studies into a single dataset. Duplicated entries were matched by title, authors, year, and venue of publication.

4) **Inclusion and Exclusion Criteria***:* We considered all the selected studies and filtered them according to a set of well-defined selection criteria. The inclusion and exclusion criteria of our study are:

Inclusion criteria:

- Studies focusing on software testing specific to the automotive industry.
- Studies providing a solution for prioritizing Automotive Software Testing.
- Studies in the field of software engineering.
- Studies written in English.

Exclusion criteria:

- Studies that focus on the automotive industry, but do not explicitly deal with software testing.
- Studies where software testing is only used as an example.
- Studies not available as full-text.
- Studies not presented in English.
- Studies that are duplicates of other studies.

5) **Removal during Data Extraction***:* When reviewing the primary studies in detail to extract information, all the authors agreed that four studies were semantically beyond the scope of this research. Consequently, they were excluded.

C. Data extraction

Relevant information was extracted to answer the research questions from the primary studies. We used data extraction forms to make sure that this task was carried out in an accurate and consistent manner. The data was collected and stored in a spreadsheet using MS Excel to list the relevant information of each paper. This technique helps extract and view data in a tabular form.

The following information was collected from each paper:

- Publication title
- Publication year
- Publication venue
- Problems faced by the authors
- Testing method used
- Limitations in field
- Detail of the proposed solution
- Results obtained
- Rating of quality issues
- Verification and validation
- Future work suggested by the authors
- Conclusions
- Answers to research questions

IV. Analysis Results

This section presents the analysis and each sub-section answers the previously presented research questions. We used the R software environment and Microsoft Excel to perform basic statistical operations and draw charts.

A. Publication Trends (RQ1)

Figure 2 presents the distribution of publications over time. The most common publication types are conference papers (17/25) followed by workshop papers (5/25), journals (2/25), and symposiums (1/25). The high number of conference papers may indicate that prioritization of automotive software testing is maturing. A small but constant number of publications were published until 2014. However, prioritization has become an important and eye-catching aspect in terms of research since 2014. The interest in prioritization of automotive software testing has rapidly increased in the last few years.

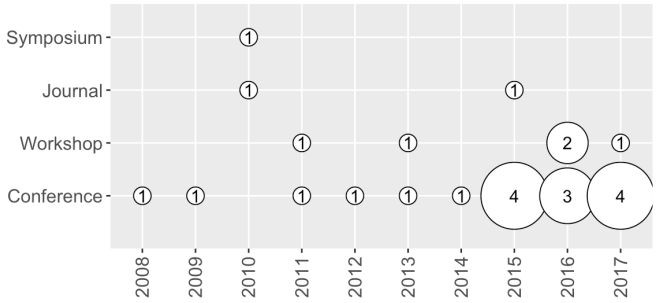

Fig. 2. Primary studies distributed by type of publication over the years

TABLE I
APPLIED RESEARCH STRATERGIES

Res. strategies	#Studies	Studies
Solution Proposal	20	P4, P5, P6, P7, P8, P9, P10, P11, P12, P16, P17, P18, P19, P20, P21, P22, P23, P24, P26, P27
Evaluation Research	15	P5, P8, P10, P11, P13, P15, P16, P17, P18, P19, P20, P21, P22, P26, P28
Validation Research	14	P5, P8, P9, P10, P11, P12, P16, P18, P20, P21, P22, P24, P26, P28
Opinion Paper	4	P13, P14, P25, P27
Survey Paper	1	P25

Studies published before 2015 refer to slightly different perspectives on prioritization than more recent papers. The number of papers has drastically increased since 2014. . [25] [11] [4] [6] [10] used model-based testing to improve the prioritization by increasing the effectiveness. On the other hand, [6] [10] showed potential improvements and proposed new model-based methods.

Many researchers provided solution proposals (20/25) and evaluation research (15/25) (Table I), indicating that today's researchers focus on industry and practitioner-oriented studies (e.g., industrial case study, action research). Another common research strategy is validation research (14/25), highlighting the fact that there is some level of evidence (e.g., simulations, experiments, prototypes, etc.) supporting the proposed solutions. However, Table I also shows that few studies employ surveys (1/25), suggesting that future studies should fill this gap.

B. Methods Used (RQ2)

Due to the requirement of connected services for vehicles, an interesting method is model-based testing. Figure 3 depicts a histogram of the distribution of the most common techniques in the literature. The most common testing methods are model-based testing (7/25), regression testing (6/25), and black box testing (5/25), followed by hardware in the loop testing

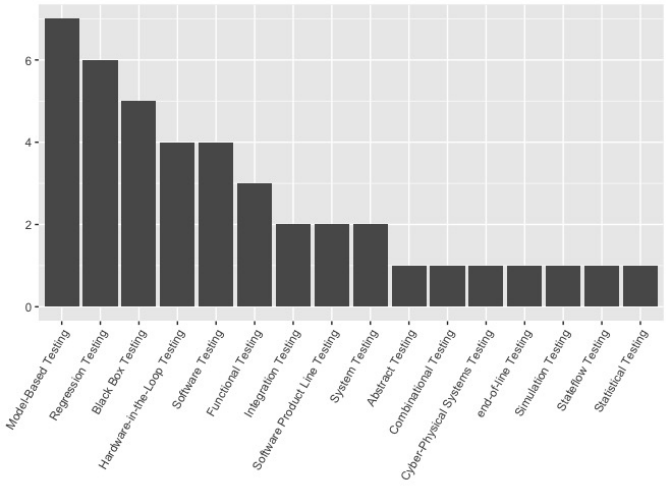

Fig. 3. Testing Methods

(4/25), software testing (4/25), functional testing (3/25), and other.

Approaches that use model-based testing are found in [4] [6] [9] [10] [11] [25] [26]. Techniques listed as "OTHER" refer to the use of integration testing [10] [24], software product line testing [4] [11], system testing [5] [8], abstract testing [17], combinational testing [18], cyber-physical system testing [13], end-of-line testing [7], simulation testing [12], stateflow testing [15], and statistical testing [20].

Different methods or combination of methods are used in multiple studies (Table II). These papers frequently target model-based testing (7/25), regression testing (6/25), and black box testing (5/25). Model-based development is an efficient, reliable, and cost-effective paradigm to design and implement complex embedded systems. The software determines more than 90% of the functionality of automotive systems and up to 80% of the automotive software can be automatically generated from models [6]. Additionally, model-based testing is a common solution to test embedded systems in automotive engineering. Regression testing is undertaken every time a model is updated to verify quality assurance, which is time-consuming as it reruns an entire test suite after every minor change. Test case selection for regression testing after new releases is an important task to maintain the availability [7]. Typically studies focus on black-box testing scenarios because the source code is often unavailable in the automotive domain such as an OEM-supplier scenario [19]. Hardware in the loop testing (4/25) and software testing (4/25) are the next most used methods. The most common method of testing the software and the Electronic Control Units (ECU) is the use of Hardware-In-the-Loop (HIL) simulation [27]. Software testing presents an approach to automatically generate test cases for a software product. Functional testing (3/25) strives to demonstrate the correct implementation of functional requirements and is one of the most important approaches to gain confidence in the correct functional behavior of a system [28]. Integration testing (2/25), software product line testing (2/25), and system testing (2/25) are used as the time donation when the testing process is limited. A negative highlight of this systematic review is the fact that only one paper directly employs a simulation testing method [12]. If a simulation environment can imitate the key criteria of the real-world

TABLE II
APPLIED TESTING METHODS

Testing Methods	#Studies	Studies
Model-Based Testing	7	P4, P6, P9, P10, P11, P25, P26
Regression Testing	6	P5, P6, P7, P9, P11, P19
Black-Box Testing	5	P5, P8, P9, P16, P19
Hardware in the Loop Testing Testing	4	P8, P16, P27, P28
Software Testing	4	P14, P21, P23, P27
Functional Testing	3	P16, P22, P28
Integration Testing	2	P10, P24
Software Product Line Testing Testing	2	P4, P11
System Testing	2	P5, P8
Abstract Testing	1	P17
Combinational Testing	1	P18
Cyber-Physical Systems Testing	1	P13
End-of-Line Testing	1	P7
Simulation Testing	1	P12
Stateflow Testing	1	P15
Statistical Testing	1	P20

TABLE III
RATING OF REVIEWED ARTICLES

Reference	QI1	QI2	QI3	QI4	Total
4	Y	Y	Y	Y	4
5	Y	P	P	Y	3
6	Y	P	Y	P	3
7	Y	Y	P	Y	3.5
8	Y	P	P	Y	3
9	Y	Y	P	P	3
10	Y	Y	N	P	2.5
11	P	Y	P	Y	3
12	Y	P	Y	P	3
13	Y	Y	Y	Y	4
14	Y	P	Y	Y	3.5
15	Y	P	Y	P	3
16	Y	Y	P	Y	3.5
17	Y	Y	Y	Y	4
18	P	P	Y	Y	3
19	Y	Y	Y	Y	4
20	Y	N	P	P	2
21	Y	P	P	Y	3
22	Y	Y	P	Y	3.5
23	P	N	Y	N	1.5
24	Y	Y	Y	Y	4
25	Y	Y	Y	P	3.5
26	Y	Y	P	Y	3.5
27	Y	Y	Y	N	3
28	Y	P	Y	P	3
				Average	3.2

TABLE IV
VALIDATION,TECHNIQUES/TOOLS, GAPS, AND MAIN OUTCOMES OF STUDIES

Validation	No. of Papers	Study	Techniques/Tools	Gaps	Main Outcome(s)
Industrial Case Study	14 (56%)	P5	Test case selection based a on Stochastic model	Find better heuristic clustering approaches	Regression effort can be minimized
		P8	Test case selection based on a component and communication model	Change deployed libraries	82.3% reduction in tested functions
		P10	Taster tool, Proposed new framework for MBT	Focus on the optimization of the classification structure used within the priority assignment procedure	New framework for the MBT
		P11	Dissimilarity-based TCP	Investigate the fault detection capabilities of the approach	Dissimilarity-based TCP issues are solved
		P15	Test selection algorithms	Develop optimal guidelines to divide test oracle budget across the output-based selection algorithms	Output-based algorithms consistently outperform coverage-based algorithms in revealing faults
		P16	Evolutionary Testing Framework, modularHiL, MESSINA	Investigate how to configure the evolutionary testing system to reduce the number of pre-tests	Solution provide complements systematic testing in that it generates test cases for situations that would otherwise be unforeseen by testers
		P17	Migration from traditional to abstract testing	Extend the formalism to handle non-functional requirements	Abstract testing is comparable in effectiveness
		P18	Equivalence Class Partitioning (ECP), Boundary Value Analysis (BVA), Choice Relationship Framework (CRF)	Investigatee efficient test case generation and discover more feasible tools and empirical studies to work	Efficient reduction in the final effective number of test cases by 42 (88% reduction)
		P19	Test case combination	Six approaches are presented to improve test efficiency	Machine learning approach in black-box testing
		P20	Combination of test models in MATLAB/Simulink	*NO GAPS MENTIONED*	Higher coverage is achieved compared with manually created test cases
		P21	Automatically generate test cases	Refine the functional coverage model	Improvement actions are identified for test case generation
		P22	Proposed unified model	Implement a large survey on software specifications in Johnson Controls company	More than 90% of the requirements are represented by the model
		P26	End-to-end test framework	Investigate the nature of the test model and the relevance to the generated test cases	Automation of executable test script generation
		P28	Search-based testing principles	Further investigate safety requirements	Promising approach to ensure safety requirements
Technique Comparison	6 (24%)	P4	Similarity-Based Product Prioritization w.r.t Deltas	Include more Solution space information (Such as Source code)	Improvement in the effectiveness of SPL testing
		P6	New model-based method for test case prioritization	Implement performance evaluation with a large-scale case study	Future regression testing can be sped up
		P7	Combined fault diagnosis and test case selection	Evaluate using an end-of-line test system at a real assembly line	System can find test cases to increase the test coverage
		P13	Weight-based search algorithms	Study weight tuning, different fitness functions, and cost-effectiveness measures	Results suggest that all the search algorithms outperform Random Search
		P23	Model slicing technique for optimal test case generation	*NO GAPS MENTIONED*	Complexity of Simulink models can be reduced
		P24	OUTFIT tool	Evaluate it with other domains such as medical and avionic systems	Potential defects can be effectively identified
Statistical Evaluation	1 (4%)	P9	Dependences between the components of embedded systems	Study exhaustiveness of the path search and correctness of path search	Reduction in test-cases for regression testing
Simulation	1 (4%)	P12	Parallelly execute loosely coupled segments	Fully automate the process of segmentation and instrumentation	Reduce the simulation testing time for both successful and failed runs
Others	3 (12%)	P14	Three approaches to automatically generate MC/DC test cases	Focus on MC/DC test case generation from formal specifications	Approaches can be combined to support different kinds of decisions
		P25	Survey paper	Publish more detailed description of the applied evaluation approach	Improvements that are possible with MBAT technologies
		P27	Study on requirements	*NO GAPS MENTIONED*	Synect provides easy test requirement specifications and management of the test results

environment, it should be used to provide early feedback on the vehicle's design.

C. Quality Evaluation (RQ3)

According to [29], a ranking was created to rate papers based on the relevance to the topic and the quality of the paper. Quality Issues (QI) are:

- QI1 Is the paper's goal clear?
- QI2 Does the assessment approach match the goals?
- QI3 Can the method be replicated?
- QI4 Are results shown in detail?

For each quality issue, articles were rated as: Yes (Y) when the issue is addressed in the text, Partial (P) when the issue is partially addressed in the text, and No (N) when the issue is not addressed in the text. These ratings were scored as Yes = 1 point, Partial = 0.5, and No = 0. Table III shows the papers that were analyzed in this SLR and their respective scores based on the Quality Issues discussed above.

D. Existing Research (RQ4)

Here we discuss the recurring problems that are targeted by primary studies, which methods described in RQ2 are validated, and the gaps mentioned in the research.

Recurring problems are time consumption (15/25), cost (13/25), and complexity (14/25) followed by test case selection (3/25) and quality improvement (3/25) (Table V). Because the testing time is expensive, it should be reduced without an uncontrolled reduction of reliability. The entire test suite must be rerun each time the system is updated or modified. Consequently, each modification makes the testing process more time-consuming. Automotive systems are becoming more complex due to a higher rate of integration and shared usage. The high complexity results in numerous interfaces, and many signals must be processed inside the system [9]. Testing activities can account for a considerable part of the software production costs. However, only two studies discuss improving efficiency [18] and safety [26] which is a negative highlight.

Table IV presents the studies within each category, Techniques/Tools, gaps, and the main study outcomes. Most studies are focus on industrial case studies (n = 14) and technique comparisons (n = 6).

TABLE V
TARGET PROBLEMS

Problems	#Studies	Studies
Time Consumption	15	P5, P6, P8, P9, P10, P12, P13, P14, P15, P16, P17, P19, P20, P24, P27
Complexity	14	P5, P6, P7, P8, P9, P10, P13, P15, P16, P17, P19, P20, P24, P27
Cost	13	P5, P6, P8, P9, P10, P13, P15, P16, P17, P19, P20, P24, P27
Test Case Selection	3	P7, P11, P28
Quality Improvement	3	P22,P25,P26
Test Generation	2	P21,P23
Problem Space Information	1	P4
Improving Efficiency	1	P18
Safety	1	P26

TABLE VI
VALIDATION

Techniques	#Studies	Studies
Industrial Case Study	14	P5, P8, P10, P11, P15, P16, P17, P18, P19, P20, P21, P22, P26, P28
Technique Comparison	6	P4, P6, P7, P13, P23, P24
Statistical Evaluation	1	P9
Simulation	1	P12
Others	3	P14, P25, P27

Table VI lists the techniques used to validate the selected studies. The most common are industrial case studies (14/25) followed by technique comparisons (6/25), statistical evaluations (1/25), simulations (1/25), and others (3/25). Technique comparisons include studies that propose and then compare a new method to an old one. Others include three studies, which [14] [27] talk about the advantages, limitations, and requirements of different approaches. [25] is a survey paper from 13 industry case studies.

V. Threats to validity

The analysis was conducted by a single person. Thus, one threat is that some information may be omitted. Moreover, the analysis is limited by the analytical skills of that single person.

VI. Conclusion

This paper overviews the Prioritization in Automotive Software Testing. The results should help companies that are planning to incorporate prioritization into their strategies. Researchers can also benefit because this study depicts the limitations and gaps in current research. Additionally, the trends in other embedded and non-embedded domains must be investigated as this should provide a more detailed picture and lessons learned regarding prioritization in Automotive Software Testing. Future work includes (i) a qualitative study to better understand test execution, test case generation, test case selection, and test analysis and (ii) addressing the identified research gaps.

References

[1] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[2] B. Kitchenham and P. Brereton, "A systematic review of systematic review process research in software engineering," *Inf. Softw. Technol.*, vol. 55, no. 12, pp. 2049–2075, Dec. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2013.07.010

[3] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *2012 7th International Workshop on Automation of Software Test (AST)*, June 2012, pp. 36–42.

[4] M. Al-Hajjaji, S. Lity, R. Lachmann, T. Thüm, I. Schaefer, and G. Saake, "Delta-oriented product prioritization for similarity-based product-line testing," in *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)*, May 2017, pp. 34–40.

[5] I. Alagöz, T. Herpel, and R. German, "A selection method for black box regression testing with a statistically defined quality level," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 114–125.

[6] A. Morozov, K. Ding, T. Chen, and K. Janschek, "Test suite prioritization for efficient regression testing of model-based automotive software," in *2017 International Conference on Software Analysis, Testing and Evolution (SATE)*, Nov 2017, pp. 20–29.

[7] S. Abele and M. Weyrich, "A combined fault diagnosis and test case selection assistant for automotive end-of-line test systems," in *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*, July 2016, pp. 1072–1077.

[8] S. Vöst and S. Wagner, "Trace-based test selection to support continuous integration in the automotive industry," in *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, May 2016, pp. 34–40.

[9] P. Caliebe, T. Herpel, and R. German, "Dependency-based test case selection and prioritization in embedded systems," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, pp. 731–735.

[10] L. Krejčí and J. Novák, "Model-based testing of automotive distributed systems with automated prioritization," in *2017 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 2, Sept 2017, pp. 668–673.

[11] R. Lachmann, S. Lity, M. Al-Hajjaji, F. Fürchtegott, and I. Schaefer, "Fine-grained test case prioritization for integration testing of delta-oriented software product lines," in *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*, ser. FOSD 2016. New York, NY, USA: ACM, 2016, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/3001867.3001868

[12] M. A. Al Mamun and J. Hansson, "Reducing simulation testing time by parallel execution of loosely coupled segments of a test scenario," in *Proceedings of International Workshop on Engineering Simulations for Cyber-Physical Systems*, ser. ES4CPS '14. New York, NY, USA: ACM, 2007, pp. 33:33–33:37. [Online]. Available: http://doi.acm.org/10.1145/2559627.2559635

[13] A. Arrieta, S. Wang, G. Sagardui, and L. Etxeberria, "Test case prioritization of configurable cyber-physical systems with weight-based search algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16. New York, NY, USA: ACM, 2016, pp. 1053–1060. [Online]. Available: http://doi.acm.org/10.1145/2908812.2908871

[14] S. Kangoye, A. Todoskoff, M. BARREAU, and P. GERMANICUS, "Mc/dc test case generation approaches for decisions," in *Proceedings of the ASWEC 2015 24th Australasian Software Engineering Conference*, ser. ASWEC ' 15 Vol. II. New York, NY, USA: ACM, 2015, pp. 74–80. [Online]. Available: http://doi.acm.org/10.1145/2811681.2811696

[15] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Effective test suites for mixed discrete-continuous stateflow controllers," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 84–95. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786818

[16] P. M. Kruse, J. Wegener, and S. Wappler, "A highly configurable test system for evolutionary black-box testing of embedded systems," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '09. New York, NY, USA: ACM, 2009, pp. 1545–1552. [Online]. Available: http://doi.acm.org/10.1145/1569901.1570108

[17] F. Merz, C. Sinz, H. Post, T. Gorges, and T. Kropf, "Bridging the gap between test cases and requirements by abstract testing," *Innov. Syst. Softw. Eng.*, vol. 11, no. 4, pp. 233–242, Dec. 2015. [Online]. Available: http://dx.doi.org/10.1007/s11334-015-0245-7

[18] J. S. Eo, H. R. Choi, R. Gao, S. y. Lee, and W. E. Wong, "Case study of requirements-based test case generation on an automotive domain," in *2015 IEEE International Conference on Software Quality, Reliability and Security - Companion*, Aug 2015, pp. 210–215.

[19] R. Lachmann and I. Schaefer, "Towards efficient and effective testing in automotive software development," in *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)*, vol. P-232, 2014, pp. 2181–2192, cited By :2. [Online]. Available: www.scopus.com

[20] S. Siegl, K. S. Hielscher, and R. German, "Modeling and statistical testing of real time embedded automotive systems by combination of test models and reference models in matlab/simulink," in *2011 21st International Conference on Systems Engineering*, Aug 2011, pp. 180–185.

[21] R. Awedikian and B. Yannou, "Design of a validation test process of an automotive software," *International Journal on Interactive Design and Manufacturing*, vol. 4, no. 4, pp. 259–268, 2010, cited By :2. [Online]. Available: www.scopus.com

[22] R. Awedikian, B. Yannou, P. Lebreton, L. Bouclier, and M. Mekhilef, "A simulated model of software specifications for automating functional tests design," in *Proceedings DESIGN 2008, the 10th International Design Conference*, 2008, pp. 561–570, cited By :1. [Online]. Available: www.scopus.com

[23] Z. Jiang, X. Wu, Z. Dong, and M. Mu, "Optimal test case generation for simulink models using slicing," in *Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS-C 2017*, 2017, pp. 363–369. [Online]. Available: www.scopus.com

[24] D. Holling, A. Hofbauer, A. Pretschner, and M. Gemmar, "Profiting from unit tests for integration testing," in *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, 2016, pp. 353–363, cited By :1. [Online]. Available: www.scopus.com

[25] M. Klas, T. Bauer, A. Dereani, T. Soderqvist, and P. Helle, "A large-scale technology evaluation study: Effects of model-based analysis and testing," in *Proceedings - International Conference on Software Engineering*, vol. 2, 2015, pp. 119–128, cited By :4. [Online]. Available: www.scopus.com

[26] J. Lasalle, F. Peureux, and J. Guillet, "Automatic test concretization to supply end-to-end mbt for automotive mechatronic systems," in *2011 International Workshop on End-to-End Test Script Engineering, ETSE 2011 - Proceedings*, 2011, pp. 16–23, cited By :4. [Online]. Available: www.scopus.com

[27] A. Bansal, M. Muli, and K. Patil, "Taming complexity while gaining efficiency: Requirements for the next generation of test automation tools," in *AUTOTESTCON (Proceedings)*, 2013, pp. 123–128, cited By :1. [Online]. Available: www.scopus.com

[28] F. Lindlar and A. Windisch, "A search-based approach to functional hardware-in-the-loop testing," in *Proceedings - 2nd International Symposium on Search Based Software Engineering, SSBSE 2010*, 2010, pp. 111–119, cited By :7. [Online]. Available: www.scopus.com

[29] U. Kanewala and J. M. Bieman, "Testing scientific software: A systematic literature review," *Inf. Softw. Technol.*, vol. 56, no. 10, pp. 1219–1232, Oct. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2014.05.006

Use-Relationship Based Classification for Software Components

Reishi Yokomori
Nanzan University
Nagoya, Japan
yokomori@se.nanzan-u.ac.jp

Norihiro Yoshida
Nagoya University
Nagoya, Japan
yoshida@ertl.jp

Masami Noro
Nanzan University
Nagoya, Japan
yoshie@nanzan-u.ac.jp

Katsuro Inoue
Osaka University
Osaka, Japan
inoue@ist.osaka-u.ac.jp

***Abstract*—In recent years, the maintenance period of the software system is increasing. The size of the software system has grown, and the number of classes and the relationship between classes are also increasingly complicated. If we can categorize software components based on information such as functions and roles, we believe that these classified components can be understood together, and are useful for understanding the system. In this paper, we proposed a classification method for software components based on similarity of use relation. For each component, a set of components used by the component was analyzed. And then, for each pair of components, the distance was calculated from the coincidence of the two sets. A distance matrix was created and components were classified by hierarchical cluster analysis. We applied this method to jlGui consisting of 70 components. 8 clusters of 36 components were extracted from the 70 components. Characteristics of the extracted clusters were evaluated, and the content of each cluster was introduced as a case study. In 7 clusters out of the 8 clusters, components of the cluster were strongly similar with each other from the viewpoint of their functions. Through these experiments, we confirmed that our method is effective for classifying components of the target software, and is useful for understanding them.**

***Index Terms*—component, use-relations, hierarchical cluster analysis**

I. Introduction

In recent years, the maintenance period of the software system has increased, by also experiencing the entry of new developers. In order to participate in the maintenance of such a software system, it is the first important task to become accustomed to the environment surrounding the software system [1], and it is necessary to understand the configuration of the target software. In the process of understanding, the developer first understands the diagram and documents representing the composition of the software and then reads the source code of each component of the software in order to grasp the details of each subsystem and the overall image.

In reading the source code, the developer browses the source code of another component one by one on an editor or a source code browser like SPARS-J [2], by referring to the package hierarchy and the name of another class appearing in the source code, and so on. In this way, the source codes of a lot of components are read strategically in order to understand efficiently. In fact, when the developer encounters a component similar to the components the developer confirmed once, it often happens that reworking occurs in order to remember its functions and role. This type of rework needs time, however, it makes their understanding deeper. However, as the number of files and classes increases through a long period of maintenance, the frequency of the rework in understanding increases, and the content to be understood becomes also complicated. The effort required for understanding the internal structure by developers participating later will be much greater than the effort expected by the original developer. By analyzing the source code by the statement unit, a lot of methods for extracting code clone [3], extracting coding-pattern [4], and extracting coding-rule [5] have been proposed. As the scale of the software system increases, the relationship between statements becomes complicated, and the analysis cost gradually increases, so we would like to propose a method with a coarser granularity, based on the anlysis by the class unit.

In the evaluation experiment for the component retrieval system SPARS-J [2], it was effective to guide the component to be browsed next based on its use relationships, and it was popular among developers. If we can categorize software components based on their functions and roles by obtaining information from analysis of use-relationships, we can understand these classified components together and this classification would be useful for understanding the target software. For example, when realizing a specific feature, it is often necessary to use certain components together. We compare use-relations of two components (-A, -B), that means components that component-A uses and components that component-B uses are compared. In the case where there are many common components in the two component set, we consider that component-A and -B are supposed to implement the same specific function or very similar function. We think that understanding of these components can be performed efficiently by also considering components that are commonly used by these components. By being able to understand efficiently, we believe that developers devote much effort to other work and can carry out with high quality. As a result, we think it can improve productivity and quality of maintenance.

In this paper, we proposed a classification method for software components based on similarity of use relation. In this method, for each component, a set of components used by the component was analyzed. And then, for each pair of components, the distance between the two components was

calculated from the coincidence of the two sets. In this way, a distance matrix of the target software components was created and software components were classified by hierarchical cluster analysis.

We implemented a prototype for the proposed method, and its analysis target is a software system described in Java. The prototype system treats the Java source files as a component and excludes components defined outside the software, such as components in libraries. Our system generates a distance matrix from the analysis result of Classycle's analyzer [6], and hierarchical cluster analysis is executed by R. We conducted two evaluation experiments. In the first experiment, we applied it to the actual software system of open source project and evaluated the similarity of the components in the obtained cluster from several viewpoints. As a result, many of the components in the cluster showed strong similarity in terms of their functionality, and more than half of similar components were joined as clusters. In the second experiment, we introduced how the components in the clusters were combined, as a case study. Some clusters were composed of a collection of components that have a simple structure and a simple use relationship, and some clusters were composed of a collection of components that realize multiple functions by using several other components. In the case study, knowing the components commonly used by the components in the cluster was effective in understanding the contents of the components in the cluster. Through these experiments, we introduce that the obtained clusters have sufficient accuracy to be used for software understanding, and our method is effective to classify and understand the components of the target software.

The main contributions of our research are as follows:

- We have hypothesized that "When two components use many common components, these two components are implemented for the same or similar purpose", and confirmed it by experiment.
- We showed that the proposed method can combine most of the functionally related components as clusters.
- We showed that the obtained clusters have sufficient accuracy for software understanding, and our method is effective to classify and understand the components of the target software.
- We showed that the components in the cluster could be used as examples when adding new components that use the same components.

Section II introduces the component graph as a background. Section III introduces our method and implementation. Section IV conducts two experiments. Finally, Section V discusses the results and improved ways and introduces related works.

II. Background

A. Component Graph and Use Relation between Components

In general, a component is a modular part of a system that encapsulates its content and whose manifestation is replaceable within its environment [7]. We model software systems by using a weighted directed graph, called a *Component Graph* [2]. In the component graph (V, E) , a node $v \in V$ represents a software component, and a directed edge $e_{xy} \in E$ from node x to y represents a *use relation* meaning that component x uses component y. This *use relation* indicates that a component realizes its own function by utilizing features of other components. If you need to understand how features in a particular component are specifically implemented, you can use use-relations to understand which parts of the software you need to reference together.

B. Classification methods using metrics that represent the characteristics of software components

For a lot of components extracted from multiple software systems, Kobori suggested a method for extracting the same or similar components as the target components [8]. In this method, metrics representing the characteristics of software components are obtained from the source code of each component. These metrics are the number of methods and the number of tokens that represent the scale of the component, and the cyclomatic complexity that represents the complexity of control structure of the component, and so on. In the method, components that have the same or similar values in these metrics are extracted. Through experiments, they showed that extracted components are mainly copied components or copied and modified components from the original ones. This method is a method assumed for using in the software component retrieval system. In the software component retrieval system, it collects a lot of software system. The method could help to extract the almost identical (copied or copied and modified) components across a lot of software systems. Our method is to classify components in one software system, based on the fact that components are using a lot of common components. The purpose of our method is to obtain a set of components using similar functions and to obtain a set of components whose role and features in the software are similar.

III. Proposed Method

When implementing a feature in a new component, we also use the features realized by other existing components as necessary. From this, it can be considered that which components are used by a certain component can also be information for estimating the features realized in the component. In some cases, when using a certain feature, it is necessary to use some other components together. In such a case, there is a commonality in the set of using components among the components that use the feature. Based on the fact that the using components are similar, we thought that these components are similar in function and role. Therefore, we make a hypothesis that "When two components use many common components, these two components are implemented for the same or similar purpose", and confirm it by experiments. We propose a method for classifying components based on the commonality of components that are used by the components. We thought that this method can extract several components groups that are recognizable as collections of components that are similar in function or role. Through evaluation experiments

to an open source project, we introduce characteristics of the obtained component groups and components in the obtained component groups have a strong relationship.

A. Implementation of the analysis tool

We implemented a classification tool for Java source files of a target software system. The system constructs a component graph for Java source files of the target software system and produces a distance matrix used by the hierarchical cluster analysis on R. For extracting use relations, we used Classycle's analyzer [6], that is an analyzing tool for java class and package dependencies. In the analysis of the Classycle's analyzer, we get the following as use relations: inheritance of class, declaration of variables, a creation of instances, method calls, and reference of fields, and inner classes and use relations about inner classes are merged into the major class in the Java source files. Figure 1 is an overview of the system, and the following is the analysis procedure:

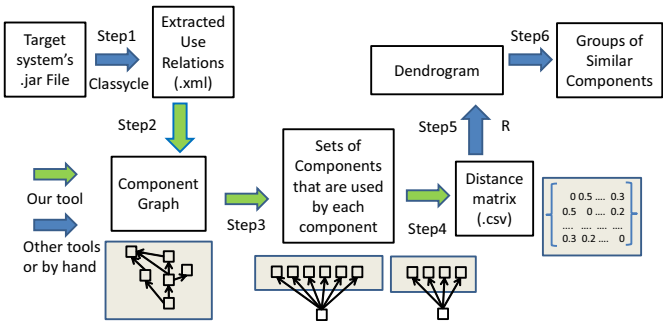

Fig. 1. Overview of the Proposed Method

1. Classycle's analyzer [6] provides the use relations of the target system.
2. Build a component graph based on the use-relations.
3. For each component, a set of components used by the component is obtained from the graph.
4. For each pair of components, a distance between the two components is calculated from a degree of coincidence of the two sets. A distance matrix for the target software components is produced.
5. By using the distance matrix, hierarchical cluster analysis is performed on R.
6. The dendrogram is obtained by R. We decide a threshold value and clusters under the threshold value are recognized as groups of similar components.

B. Distance between two Components

Consider two components A and B. L_{A} represents a set of components that are used by A, and L_{B} represents a set of components that are used by B, respectively.

At first, we define the similarity of components A and B $sim(L_{\mathrm{A}}, L_{\mathrm{B}})$ by using the Jaccard coefficient.

$$sim(L_{\mathrm{A}}, L_{\mathrm{B}}) = \frac{|L_{\mathrm{A}} \cap L_{\mathrm{B}}|}{|L_{\mathrm{A}} \cup L_{\mathrm{B}}|}$$

This value has a range of 0 to 1, and the higher this value, the more likely that L_{A} and L_{B} are similar. And then, we define the distance of components A and B $dist(L_{\mathrm{A}}, L_{\mathrm{B}})$ as following; (For representation, the distance was 10 timed, and it does not have other special meaning.)

$$dist(L_{\mathrm{A}}, L_{\mathrm{B}}) = (1 - sim(L_{\mathrm{A}}, L_{\mathrm{B}})) * 10$$

C. About Linkage Criterion for Hierarchical Clustering

In the hierarchical clustering, there are candidates to compute the distance for a cluster consists of multiple objects, so the user needs to decide on the linkage criterion to use. For our study, we decided to use the linkage criterions provided by R, since it was intended to create a prototype of an analytical environment. The linkage criterions are roughly divided into two methods: methods to recalculate a distance directly from distances of multiple objects, such as a single linkage, a complete linkage, and UPGMA, and methods to recalculate a distance from the center or the center of gravity of the cluster, such as Ward, and centroid method. From them, we decided to use the former methods. In general clustering, it is considered that the latter methods can be utilized because a set of vectors is used as input and hierarchical clustering is done by calculating the distance between vectors. However, in our analysis, only the distances between components are given as input, so it is meaningless to consider the center of the cluster.

We introduce three types of results, based on the single linkage method, the complete linkage method, and the UPGMA, as a preliminary experiment, and examine which linkage criterion should be used. The target system is GoGui that is a graphical interface program for playing Go and is composed of 181 Java source files. Figure 2 is a dendrogram of the hierarchical clustering using the single linkage method. In the case of the single linkage method, if a component has a similarity to any component in the cluster, the component is also merged into the cluster. So a very large cluster is easily formed, however, similarities between components in a cluster would not be strong. Figure 3 is a dendrogram of the hierarchical clustering using the complete linkage method. In the case of the complete linkage method, a component is merged into the cluster only if the component has similarities with all components in the cluster. A lot of small clusters are formed, however, similarities between components in each cluster would be strong.

Figure 4 is a dendrogram of the hierarchical clustering using the UPGMA. We can confirm a lot of small clusters as in the case of Figure 3, and we can also confirm that these clusters are merged into a large cluster as in the case of Figure 2. In this way, the dendrogram obtained by UPGMA is a balanced result. By adjusting a threshold height for getting clusters, we can obtain very similar clusters obtained by the other two approaches. In this study, we use the UPGMA as a linkage criterion in the hierarchical clustering. The problem of deciding the threshold height for getting cluster is considered to be a problem to be tackled in the future, and we will show how the results change by the upper or lower thresholds in the experiments. As another linkage criterion, we can also

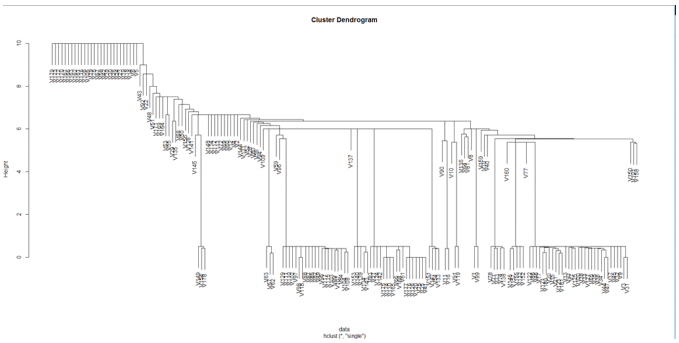

Fig. 2. A Dendrogram of GoGui Components (SLM)

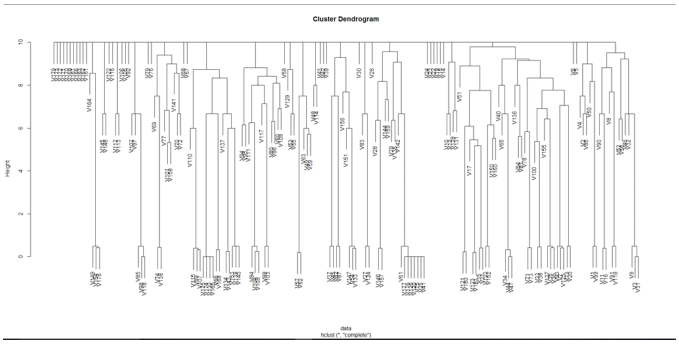

Fig. 3. A Dendrogram of GoGui Components (CLM)

consider a method to recalculate a group of using components for each cluster and recalculate the distance between clusters. This also would be a problem to be tackled in the future, however, we consider that it contributes to the improvement of accuracy and the overall trend does not change significantly.

IV. Experiments

For the experiments, we applied our method to jlGui Ver.3.0. It is a graphical music player which supports Java Sound capabilities and consists of 70 Java source files. We obtained a dendrogram of hierarchical clustering using UPGMA. By showing the results of the following two experiments, it is shown that the obtained clusters were composed of components with strong similarity.

1 Can we say that the components in the resulting cluster are strongly related components?
Initially, we prepared several situations that we can imagine when the roles and objectives of the components are similar. Considering some conditions that similar components satisfy, we treated them as criteria to show similarity. For each criterion, we examined how many components in the cluster meets the criteria. It corresponded to the accuracy. Then, we examined how many similar components exist outside the cluster. It corresponded to the recall.

2 What kind of clusters have been obtained?
For some clusters, we introduced the collection of components in the clusters and the role of each component, as a case study. We will show the results classified by our method were realistic ones.

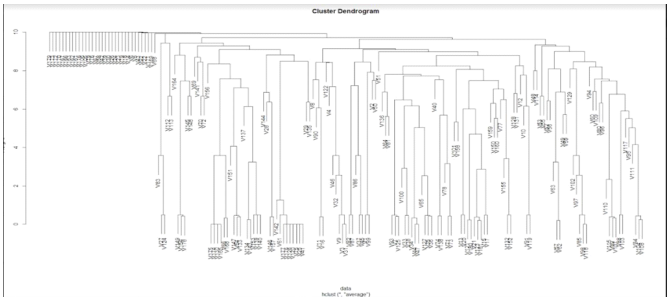

Fig. 4. A Dendrogram of GoGui Components (UPGMA)

A. About Criterion for similarity

At first, we prepared several situations that we can imagine when the roles and objectives of the components are similar.

1) Many of the components that have common processing to achieve similar functionality are derived from the same class or implement the same interface. The processing performed by these components and the APIs provided by them is often quite similar. These components share similar code fragments, so they would be comprehensible at once.
2) Components that realize different objectives for the same target should be considered as a group associated with the target. Common processing would be performed as pre- or post-processing. So these components share similar code fragments and would be comprehensible at once to understand how to use the target.
3) Components that are considered to realize a large feature are also considered as a functional group when observing from a vantage point. The targets of these components are often slightly different, but it is possible to verify the relationship between them. These components will help you to get a rough understanding of what features are provided for a particular element.
4) If the same design pattern is deployed in multiple locations in a software system, you can see that there are multiple components that have a specific role on the design pattern in the software. We believe that using the same design pattern has a common intention, so these components are considered to be comprehensible to understand a part of software design. In addition, these components sometimes share similar code fragments to perform the common processing.

In such cases, we thought that the components used by these components had a commonality. Besides, the similarity of components may be confirmed by the similarity of package classification and their names. Based on those, seven criteria were set to confirm that the two components are similar, and they were used for calculating the precision and the recall.

C1 The two components were derived from the same class or implemented the same interface.

C2 The two components belonged to the same package.
C3 The two components used the same word as a part of the name.
C4 The two components shared similar code fragments.
C5 When comparing the components used by the two components, the proportion of matching components was over 50%.
C6 The two components were thought to constitute a part of a specific functional group.
C7 There were several methods with the same signature in the two components.

B. About precision and recall

In order to calculate the precision, we randomly extracted a component from each cluster and treated it as component A. Next, at the given criterion, components related to component A were searched from the components in the cluster. We calculated the number of related components in the cluster divided by the number of components in the cluster and we regarded it as the precision for the criteria. If there is no related component without component A, it is represented by $\times$.

The recall was calculated only for items where related components existed. At the given criteria, components related to component A were searched from all components in the software. We calculated the number of related components in the cluster divided by the number of related components in the software and we regarded it as the recall for the criteria. If there were no related components in the cluster without component A, we express it by $\times$. For both cases, the component A was also counted for the number of components.

C. Experiment 1 : Can we say that the components in the resulting cluster are strongly related components?

1) Overview: For 70 jlGui components, we performed the hierarchical clustering based on UPGMA. Figure 5 is a dendrogram of the analysis, and we cut the dendrogram by a red line on Figure 5 and got the eight sets of components joining under the red line as clusters. These clusters were numbered as shown in Figure 5, and we investigated the precision and recall for each criterion for each cluster.

TABLE I
THE PRECISION FOR JLGUI CLUSTERS

	NC	C1	C2	C3	C4	C5	C6	C7
#1	5	4/5	4/5	4/5	×	5/5	5/5	4/5
#2	4	4/4	4/4	4/4	3/4	4/4	4/4	4/4
#3	9	4/9	7/9	7/9	×	7/9	6/9	8/9
#4	4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
#5	5	4/5	5/5	5/5	3/5	5/5	5/5	5/5
#6	3	3/3	×	3/3	3/3	3/3	3/3	3/3
#7	4	×	2/4	2/4	×	4/4	3/4	×
#8	2	×	×	×	×	×	×	×

The Table I is a summary of the precision. For each cluster, the table shows the number of components in the cluster (NC) and the precision for each criterion, represented by the number of similar components in the cluster and the number of components in the cluster. In cluster #1-#7, the high percentage was shown on most criteria, and you can confirm strong relationships among components in the clusters. For these components, the structure, location, and origin of the name were very similar and provides similar methods, so these components can be recognized as functional groups. Some of these components shared similar code fragments, others do not. In experiment 2, we detailed the contents of these clusters. On the other hand, we could not confirm any strong relationship between the components in cluster #8. The cluster #8 consisted of two components, one was the delegation component for the music playlist and the other was the component to search music files. Since the two components used the **Playlist** and **Playlistitem** in common, there was a common point that the two components manipulate the playlist. However, we could not confirm any relevance other than that.

TABLE II
RECALL FOR JLGUI CLUSTERS

	NC	C1	C2	C3	C4	C5	C6	C7
#1	5	4/4	4/6	4/5	×	5/5	5/5	4/5
#2	4	4/5	4/7	4/7	3/8	4/8	4/7	4/7
#3	9	4/8	7/21	7/10	×	7/12	6/10	8/10
#4	4	4/8	4/10	4/10	4/4	4/10	4/9	4/13
#5	5	4/8	5/10	5/10	3/3	5/10	5/9	5/13
#6	3	3/8	×	3/5	3/7	3/6	3/5	3/3
#7	4	×	2/4	2/6	×	4/17	3/7	×
#8	2	×	×	×	×	×	×	×

Next, the Table I is a summary of the recall for each criterion for each cluster. For each cluster, the table shows the number of components in the cluster (NC) and the recall for each criterion, represented by the number of similar components in the cluster and the number of similar components in the software. From the result, we confirmed that most of the related components were collected into clusters #1. Clusters #4 and #5 seemed to be integrated because the correct set of components were almost identical for them. Approximately half of the related components were collected into clusters #2, #3 and #6. There was no big difference between the criteria, and the overall trend was high. Overall, we found that many of the components considered to be similar were included in the clusters. However, as in the case of #7 and #8, there were cases where no strong association was found.

D. Experiment 2 : What kind of clusters have been obtained?

As a case study, we will explain components in the cluster #1-#7 of the Figure 5. First, we introduced the components obtained by the red lines in the Figure 5, and explained about components that were commonly used. We also introduced which components were newly added when moving the line upward. Through these introductions, we will show that the results classified by this method are realistic.

1) Cluster #1: 5 components: Four of the five components were components such as **OggVorbisInfo**, which handled information for several compression types, such as Ogg Voris,

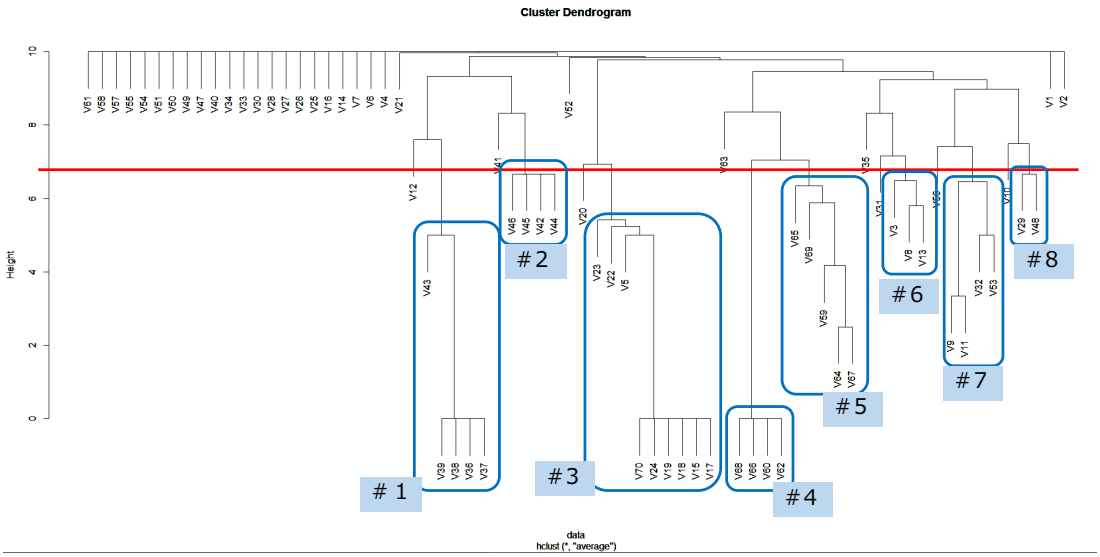

Fig. 5. A Dendrogram for jlGui (UPGMA)

Mpeg, Flac, APE etc. and these were derived from **TagInfo**. The fifth component was **EmptyDialog**. This also used **TagInfo**, and was a part of the dialog related components. These five components were clustered because they used **TagInfo** in common. And any other components were not used in common. If you move the red line upward, **Playlistitem** representing each playlist item will be joined to the cluster.

2) Cluster #2: 4 components: Four components were components such as **OggVorbisDialog**, which handled dialogs for several compression types, such as Ogg Voris, Mpeg, Flac, APE etc. and these were derived from **TagInfoDialog**, and used a corresponding component in Cluster #1. These four components were clustered because they used **TagInfoDialog** in common. And any other components were not used in common. If you move the red line upward, **TagInfoFactory**, that was a factory class for **TagInfo** and **TagInfoDialog** joined to the cluster #2, and finally, cluster #1 and #2 were integrated.

3) Cluster #3: 9 components: Nine components were several types of GUI components, such as icons, bars, sliders, popups, buttons, and so on. These components were derived from the swing components in the standard library for GUI. These components were clustered because they used **AbsoluteConstraints** in common. And any other components were not used in common. **AbsoluteConstraints** was a component for managing information about size and position on the screen. There were 15 components using **AbsoluteConstraints**, and they were scattered across Cluster #3, #6 and #8. If you move the red line upward, **ActiveJLabel** joined the cluster #3 because it also used **AbsoluteConstraints**.

4) Cluster #4: 4 components: Four components were components like **VisualizationPreference**, which handled the preferences of some items. These components were derived from **PreferenceItem**, and these components were clustered because these components used **PreferenceItem**. And any other components were not used in common. If you move the red line upward, cluster #4 and #5 were integrated. Components in Cluster #4 only used **PreferenceItem**, not other components.

5) Cluster #5: 5 components: As in the case of cluster #4, five components were components like **Preferences**, which handled the preferences. These components were also derived from **PreferenceItem**, and these components were clustered because these components used **PreferenceItem**. And any other components were not used in common. If you move the red line upward, cluster #4 and #5 were integrated. Components in Cluster #5 used **PreferenceItem** and several other components and this was a difference.

6) Cluster #6: 3 components: Three components were components that realized user interfaces, such as **PlayerUI**, **PlaylistUI** and **EqualizerUI**. These components were clustered because these components used 8-10 GUI components in common, including **Loader**, **Skin** and components in Cluster #3. If you move the red line upward, **Skin** and **StandalonePlayer** joined to the cluster #6, because these components managed a core feature using several GUI components.

7) Cluster #7: 4 components: First, **BasePlaylist** and **PlaylistFactory** joined the cluster #7. These two components used **PlayList** and **Config**, and realized playlist-related features. After that, **SkinLoader** and **FileUtil** joined the cluster #7. The latter two components didn't use **PlayList**, however, they used **Config** and other classes in common. If you move the red line upward, components using **Config** joined to the cluster. There were 15 components using **Config**, which were scattered across Cluster #5, #6 and #7.

In this way, most of the components in the cluster were functionally related components, and these components were divided into two large categories. The components of the first category had a simple structure and a simple use-relation and realized a single function by derivation. The purpose of these

components was consistent with the purpose of commonly used components. The components of the second category used several components and realized multiple functions. The purposes of the commonly used components were integrated and it becomes the purposes of the components. When we have to realize another screen for the software, these components might be useful as examples. In this way, understanding the commonly-using components was essential to understand the roles of the components in the cluster.

Table III lists the jlGui components in order of the number of components that use the component. Most of the components in the list were mentioned in our case study, so we thought the classification result was appropriate as a whole.

TABLE III
THE NUMBER OF COMPONENTS USING EACH JLGUI COMPONENT

Components	Used by
javazoom.jlgui.player.amp.skin.AbsoluteConstraints	15
javazoom.jlgui.player.amp.util.Config	15
javazoom.jlgui.player.amp.PlayerUI	10
javazoom.jlgui.player.amp.tag.TagInfo	10
javazoom.jlgui.player.amp.util.ui.PreferenceItem	9
javazoom.jlgui.player.amp.tag.ui.TagInfoDialog	7
javazoom.jlgui.player.amp.playlist.Playlist	6
javazoom.jlgui.player.amp.playlist.PlaylistItem	6
javazoom.jlgui.player.amp.skin.Skin	6

V. CONSIDERATION

A. Evaluation of Experiments

We classified jlGui components based on the similarity of the components used by each component and confirmed that many of the components in the obtained clusters showed strong relationships with each other. It is effective to understand the role of the components in the cluster based on the role of commonly used components and this approach is effective as a systematic method to efficiently understand the inside of the software. As in the situation that components using **AbsoluteConstraints** distributed in several clusters, components that use the same components may spread across multiple clusters. In that case, we think that the use-relations may be different depending on the role difference, so these components are classified into different clusters.

However, there are cases where the roles of commonly used components do not directly indicate the role of the components, and the roles of commonly used components are ambiguous. There seem to be some useless cases in this way, but we consider that our method is effective as a whole.

B. Refining Approaches for the Proposed Method

In the case of jlGui, there were about 20 components which did not use any classes in jlGui, and these were not classified. In the maximum case, about 30% to 50% of the components may not be classified. Some components were realized by not using components in jlGui but using only library components. So the number of classified components would be increased if we also consider use relations to the external components. Classification would be affected by how use-relations to the external components are treated in the calculation of similarity. So we would like to confirm whether consideration of external components is effective or not, and propose a refined method.

Moreover, for Cluster #1 and #2, and for #4 and #5, an integrated cluster may be appropriate categorization when we consider functional groups in a broad sense. In some cases, the result would be improved by moving the threshold height up and increasing the size of the cluster. On the other hand, in other cases, the result would be improved by moving the threshold height down and selecting components carefully. We would like to confirm the appropriate threshold trend.

C. Usages of the Proposed Method

First of all, as a method of using the proposed method, we would like to use it as a method for the efficient grasp of components and inside of the software. Next, we also believe that some contributions can be made to maintenance work. For example, we consider that it can also be used for checking for omissions of correction in maintenance work. As a method of assisting such checking work, you may use the commit history or similar code fragments, and we expect the same effect as those. Moreover, we think that it is effective to suggest components with similar use-relationships, as reference examples, when adding new components to implement additional features in maintenance. Finally, we consider that our method supplies a good viewpoint to express the software's evolutionary process. By comparing the dendrograms of each version in the middle of development, you can grasp which part of the software is growing and can grasp the direction of evolution and take countermeasures for it.

D. Comparison with Other Existing Methods

Currently, a comparison is not done as an experiment, and comparison with existing classification method is a problem to be tackled in future. In the feeling of calculating the recalls of Experiment 1, our method seems to perform better classification within the scope of analysis between classes, compared to the result of classification based on package hierarchy and based on similar code fragments. In the case of package-based classification, classification is done with a larger granularity, so a certain package contains several types of components. Our method can classify them components based on their purposes. In the case of code clone-based analysis, as the modification after copying increases, a similar code fragment becomes a gaped clone and it becomes difficult to grasp. Descriptions using other classes remain at the root, so the method is often highly resistant to modification.

E. Threats to Validity

In this study, we applied our method to only one software system, so it is necessary to discuss generalization by applying to various software systems. The main concern is whether the same tendencies can be confirmed in other domain software and whether similar tendencies can be confirmed in different

software systems of different sizes. The purposes of components in one software system are considered consistent, so it may have derived good results. In order to work well, a certain degree of commonality is required for target components, so it might not work well if the target of the analysis was multiple software systems. In this regard, the classifications may be effective if components generated by a certain product line were analyzed. There is a possibility that the approach of the class definition in Java and our classification method have successfully engaged. Whether our method works well for software written in other programming languages is unknown.

F. Related Works

From the viewpoint of use-relation analysis, architecture recovery is an active research field. Zhang represented the object-oriented system using a class graph and proposed a clustering algorithm to restore the high-level software architecture [9]. Constantinou represented the hierarchical relationships between components as a D-layer and examined the relationship between the architecture layer and the design metrics [10]. Kula et al. proposed the Software Universe Graph (SUG) Model as a structured abstraction of the evolution of software systems and their library dependencies over time and demonstrated its usefulness by showing several views of visualizations [11].

There are many studies to extract patterns from the call history of API on the source code. Zhong at el. proposed a system to extract call sequences of API from open source software's repositories [4], and showed that the system is useful for learning how to use the library's API. Li at el. proposed a system to extract function call sequences from C codes as programming rules, and the system provided a mechanism for detecting a portion that violates the rules [5]. Our method differs from these methods in that the granularity of our analysis is between components. We believe that it can extract informative information even in component-based approach.

As a method to extract similar code fragments, developers can use code clone analysis. Mondal analyzed the stability of several kinds of cloned codes, and they reported that Type3 clones, known as gapped clones, have higher stability than other clones [12]. Antoniol analyzed the evolution of code duplications in 19 versions of the Linux kernel [13]. Yoshida et al. proposed an approach to support clone refactoring based on code dependency among code clones [14].

VI. Conclusion

In this paper, we proposed a classification method for software components based on similarity of use relation. For each pair of components, the distance was calculated from the coincidence of their using components, and the hierarchical clustering is performed based on the distances. From the result of experiments, we confirmed that many components of the cluster were strongly similar with each other from the viewpoint of their functions, and the purpose of commonly used components greatly helps to understand the role of components in the cluster. We confirmed that our method is effective for classifying components, and is useful for understanding them. For future works, we would like to compare properties with other classification methods including stochastic block model, confirm its generality, and improve accuracy. Moreover, we would like to realize supporting tools for software understanding and for maintenance activities.

Acknowledgment

This research is supported by Nanzan University Pache Research Subsidy I-A-2 for the 2018 academic year. We would like to thank H. Kagami, N. Mase, and M. Sawai. This research is based on their bachelor thesis [15]. We were supervisors of it and added experiments and evaluations.

References

[1] B. Dagenais, H. Ossher, R. K. E. Bellamy, M. P. Robillard, and J. P. de Vries, "Moving into a new software project landscape," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering (ICSE2010)*, 2010, pp. 275–284.

[2] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 213–225, 2005.

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

[4] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP 2009)*, 2009, pp. 318–343.

[5] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *proceedings of the 10th European software engineering conference*, 2005, pp. 306–315.

[6] Classycle, http://classycle.sourceforge.net/.

[7] C. Krueger, "Software reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, 1992.

[8] K. Kobori, T. Yamamoto, M. Matsushita, and K. Inouee, "Classification of java programs in spars-j," in *proceedings of the International Workshop on Community-Driven Evolution of Knowledge Artifacts*, 2003.

[9] Q. Zhangs, D. Qiu, Q. Tian, and L. Sun, "Object-oriented software architecture recovery using a new hybrid clustering algorithm," in *Proceedings of the Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, 2010, pp. 2546–2550.

[10] E. Constantinou, G. Kakarontzas, and I. Stamelos, "Towards open source software system architecture recovery using design metrics," in *Proceedings of the 15th Panhellenic Conference on Informatics*, 2011, pp. 166–170.

[11] R. G. Kula, C. D. Roover, D. M. German, T. Ishio, and K. Inoue, "A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem," in *25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER2018)*, 2018, pp. 288–299.

[12] M. Mondal, C. Roy, M. Rahman, R. Saha, J. Krinke, and K. Schneider, "Comparative stability of cloned and non-cloned code: An empirical study," in *Proceedings of the 27th ACM Symposium on Applied Computing*, 2012, pp. 1227–1234.

[13] G. Antoniol, U. Villano, E. Merio, and M. D. Penta, "Analyzing cloning evolution in the linux kernel," *Information and Software Technology*, vol. 44, no. 13, pp. 755–765, 2002.

[14] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "On refactoring support based on code clone dependency relation," in *Proceedings of the 11th IEEE International Software Metrics Symposium*, 2005, pp. 16:1–16:10.

[15] H. Kagami, N. Mase, and M. Sawai, "Evaluation of software parts classi cation method based on commonality of parts to be used and use source," in *Bachelor Thesis of Nanzan University*, 2018, (In Japanese).

