Workshop Preprints

QuASoQ 2016

4th International Workshop on
Quantitative Approaches to Software Quality

co-located with APSEC 2016
Hamilton, New Zealand, December 6th , 2016

Editors:

Horst Lichter, RWTH Aachen University, Germany
Toni Anwar, UTM Johor Bahru, Malaysia
Thanwadee Sunetnanta, Mahidol University, Thailand
Konrad Fögen, RWTH Aachen University, Germany

Table of Contents

Organization

Trying to Increase the Mature Use of Agile Practices by Group Development Psychology Training — An Experiment

Lucas Gren
Chalmers and the University of Gothenburg
Gothenburg, Sweden 412–92 and
University of São Paulo
São Paulo, Brazil 05508–090
Email: lucas.gren@cse.gu.se

Alfredo Goldman
University of São Paulo
São Paulo, Brazil 05508–090
Email: gold@ime.usp.br

***Abstract*—There has been some evidence that agility is connected to the group maturity of software development teams. This study aims at conducting group development psychology training with student teams, participating in a project course at university, and compare their group effectiveness score to their agility usage over time in a longitudinal design. Seven XP student teams were measured twice (43+40), which means 83 data points divided into two groups (an experimental group and one control group). The results showed that the agility measurement was not possible to increase by giving a 1.5-hour of group psychology lecture and discussion over a two-month period. The non-significant result was probably due to the fact that 1.5 hours of training were not enough to change the work methods of these student teams, or, a causal relationship does not exist between the two concepts. A third option could be that the experiential setting of real teams, even at a university, has many more variables not taken into account in this experiment that affect the two concepts. We therefore have no conclusions to draw based on the expected effects. However, we believe these concepts have to be connected since agile software development is based on teamwork to a large extent, but there are probable many more confounding or mediating factors.**

I. Introduction

Agile Project Management and its methods evolved during the nineties on ideas from lean production and more flexible product development [1], but also from practical experience saving IT projects that were about to fail [2]. The main difference between lean production and agile project management is that both management ideas admit they do not know what the best end-product would look like far in advance [3]. The agile development processes are often intimately connected to high performing, self-managing and mature teams [4] and the way group norms are set has been shown to increase performance [5]. Agile development, as compared to plan-driven ditto, implies more communication and focus on human factors, which make the group psychology aspects of teams a key ingredient [6]. However, the agile processes do not explicitly include the temporal perspective of what happens to all teams over time from a group maturity perspective.

In this experiment, we conducted a longitudinal study of seven agile teams to see if the group development affects process agility. By giving half of the teams training in group psychology theory we hoped to see an effect on their measured agility. However, by only giving a 1.5-hour lecture, we did not see such an effect. We instead discuss reasons for our non-significant results and suggest next steps for future attempts at finding such effects in complex social systems.

We follow Jedlitschka, Ciolkowski and Pfahl's [7] guidelines on how to to report experiments on software engineering throughout this paper. We will therefore start by giving a theoretical background (Section II), describe the experiment in detail (Section III), analyze the data and show descriptive statistics and tests (Section IV), and, finally, discuss the result (Section V) and provide conclusions and suggestions for future work (Section VI).

A. Context

When software development teams transition to an agile approach (i.e. more team-based work) more of the process is dependent on how well the team cooperates [4]. The agile adoption sometimes fails due to the fact that an agile transition is a cultural change as well, which impose new constellations of teams [8], [9]. To further explore the causal relationship between the group dynamics and agile practices over time, would therefore be interesting, both from a research and an industrial perspective, in order to guide agile adoptions better.

B. Problem statement

Many aspects of group dynamics come into play in the team-based workplace [10]. There are studies showing a correlation between group maturity and agile concepts (see e.g. [11]), however, little is known of any causal relationship between them. Correlation analysis only show the connection between the two. If the mature usage of agile practices are directly dependent on group development aspects has not yet been investigated. Therefore, it would be interesting to see if group psychology training of agile software development teams could increase the adoption of concrete agile practices.

II. Background

A. Agile methods (processes)

Agile methodologies can be seen as an approach rather than a technique that mostly change the culture and values behind managing projects. There are some more concrete agile methods, but they all basically share the same values. However, in order to understand how these methods work in practice, we will now shortly present some of the agile practices and how the values are implemented.

a) eXtreme programming (XP): eXtreme programming was the first method created by the agile community and is the most researched method [12] and is considered relatively strict and controlled. The practices that implement the agile principles are [13]:

1) *The planning game.* In the beginning of each iteration, the team, managers, and customers meet and write requirements in form of user stories (written in clear natural language and in a way that everybody can understand). During these meetings the whole group estimates and prioritizes the requirements.
2) *Small releases.* Working software is up and running and delivered very fast and new versions are released continuously, from every few days to every few weeks.
3) *Metaphor.* Customers, managers, and developers model the system after a constructed metaphor or set of metaphors.
4) *Simple design.* Developers are asked to keep design as simple as possible.
5) *Tests.* The development is test-driven (TDD), i.e., the test are written before the code.
6) *Re-factoring.* The code should be revised and simplified over time.
7) *Pair-programming.* All code is written by having two developers per machine.
8) *Continuous integration.* The developers integrate new code into the system as often as possible. However, all code must pass the testing otherwise it is discarded.
9) *Collective ownership.* Developers can change code wherever necessary and the overall code is assessed.
10) *On-site customer.* A customer is in the team all the time to answer questions so the team always works according to what is needed.
11) *40-hour work week.* The team works with a sustainable pace defined as a 40 hour work week. The requirement selected for each iteration should never mean that the team needs to work overtime.
12) *Open workspace.* The team should be collocated and fit in the same room. The layout of the room should make cooperation and communication easy.

b) Scrum: Scrum is based on XP and is one of the more common methodologies and is built on embracing change and focus a lot on delivering value. In Scrum, the project has a prioritized backlog of requirements and use iterative development (called "sprints") to get basic working software for the customer to view as soon as possible. Scrum uses self-organizing teams that get coordinated through daily meetings called "scrums." The manager is called a "Scrum Master" to clarify that it is a facilitating role and not a directive one.

The Scrum methodology consists of three main phases: Pre-sprint planning, sprint (iteration), and post-sprint meeting. All work to be done is kept in a "release backlog" where from requirements (user stories) are taken to the current "sprint backlog." The requirements are usually broken down from a higher abstraction level when the sprint backlog is made. The actual sprint (usually 2–4 weeks) is when the implementation is performed. Here, the sprint backlog is frozen and the team "sprints" to complete what was planned. The team members choose tasks they want to work on themselves. "Scrum meetings" also called "Daily scrums" are 15-minute meetings every morning were the team members check status, report problems, and keep the whole team focused on a common goal. The post-meeting is done to evaluate the process and demonstrate the current system. One important aspect of Scrum is to have small working teams in order to maximize communication, minimize overhead, and maximize the sharing of informal (or tacit) knowledge. The team should also agree and be able to define when something is considered "done" [14].

c) Lean and Kanban: The flexible project management techniques and focus on customer value is not new. Within lean manufacturing these aspects have existed a long time (for more information about lean manufacturing see for example [15]). Many companies combine the process of Scrum with Kanban (Scrum-ban). It is important to note that Kanban is a signal card to pull products through the process within Lean production but has become a software development tool itself [16]. Scrum is a more strict process and can be modified by changing the WIP (work in progress) in each sprint into being connected to the work-flow state to prevent too much WIP. Kanban also allows adding items within each sprint. Another aspect is to change the sprint backlog owned by the team into a Kanban board with multiple teams with work-flow state instead. The Kanban board is never reset after a sprint and can be followed over time, and is also less dependent on collocation. Scrum only allows three different roles of the team, while Kanban does not have a limit. Therefore, larger teams in larger organization with a diversity of specializations often use Kanban or Scrum-ban when possible [17].

d) Crystal: We will not describe the Crystal methodologies in detail but, generally speaking, they are built on the assumption that the main problem in software development is poor communication. Crystal focuses on people, interaction, community, skills, talents, and communication as main effects on performance [18].

The twelve agile principles are a very high-level description of a work environment. Agile software development is an ambiguous concept with descriptions on various levels of abstraction. Many of these are obviously connected to group dynamics. The problem is that these psychological aspects are not described in detail in the methods (processes). This means that this dimension is left out for practitioners to figure our for themselves to a large extent. In order to try to operationalize agility and correlate the measurement to group maturity over time, we enforced the twelve original XP practices (described in Section II-A0a) on all the participating student teams and then opted to use the Perceptive Agile Measurement developed by So and Scholl [19] in order to measure this "agile" behavior over time. All the items are included in Section III-E.

B. Groups and Teams

A group can be defined as: "three or more members that interact with each other to perform a number of tasks and achieve a set of common goals" [20]. If the group is larger all the members might not have a common goal, which means that larger groups often consist of subgroups. Some studies have shown that smaller groups are more productive than larger groups with a threshold at around eight individuals [21]. In psychology, a "work-group" is a group that has a shared view of the group goal and has developed a structure that enables goal achievement. A team, on the other hand, is an effective work-group, however, we will use the terms somewhat interchangeably in this paper, since agile work-groups are called "teams" no matter their actual effectiveness. In social psychology, though, only 17% of all groups were considered teams according to one study [22].

The group research in psychology received much attention after the second world war and before the sixties. After that, the focus in research was on the individual instead of groups [22]. The start of the human factors research in software engineering has also mostly focused on individuals and their personalities and traits for 40 years without finding any coherent results [23]. Therefore, we have reason to believe that much of what happens in software engineering is set on team-level, which means that "agility" is hard to obtain if we do not understand the group dynamics of agile teams, or as Wheelan and Hochberger [22] very adequately put it: "before one jumps to fix something, one has to know what is broken."

During so many years of research on groups in psychology, there are, of course, a diversity of group development models [24]. However, there seems to be a reoccurring patterns of what happens to all types of groups when humans get together in order to solve a task. The first researchers to aggregate models into a general group development model were Tuckman and Jensen [25] in the seventies. In the nineties Susan Wheelan did a similar aggregation of existing models that resulted in the Integrated Model of Group Development that we used in this study. However, Tuckman and Jensen's [25] model with the phases; Forming, Storming, Norming, and Performing correspond well to the stages suggest by Wheelan [22].

C. Wheelan's Integrated Model of Group Development

The Integrated Model of Group Development (or IMGD) describes four different stages that all groups go through in their journey towards becoming a well-functioning high performing team. These stages are illustrated in Figure 1 and described next. The Group Development Questionnaire (the GDQ), that is a measurement of how much energy the group is spending on each development stage, is described afterward.

a) Stage 1 — Dependency and Inclusion: During the first stage of group development (i.e. when the group is new) the group members have more focus on safety and inclusion, a dependency on the designated leader, and more of a wish for order and structure, than in more mature stages. A group at stage one can still get work done, but will focus more on figuring out who the other people are. There is a lack of structure and the group needs to become organized, being able to do efficient work, and achieve the group goals. The group members need to create a sense of belonging and lay the foundation for how to interact within the group. At this first stage, there is a lack of the feeling of belonging to a group, but after this stage people start feeling safe enough to state their ideas and contribute to how they think the group should work in order to achieve its goals. If this does not happen groups stagnate, which is often noticed when group members stop doing work between meetings and even stop attending the group meetings [22].

b) Stage 2 — Counter-Dependency and Fight: During the second stage the group starts having conflict. These differences in opinion is a must in order to create clear roles based on real competence and to make it possible to work together in a constructive manner. The group members have to go through this more turbulent stage in order to build trust. After feeling safe and therefore daring to have these conflicts, a sense of loyalty emerges, which is needed to create cohesion. Since we do not have a clear picture of goals and roles in the beginning, we need this emotional and hard work in order to get shared perceptions of values, norms, and goals, which need to be set on group-level. Since everybody needs to believe in the group values and norms for them to fill their purpose, the rules of the game need to be negotiated so that all members thoroughly believe in them. The more shallow discussions about goals probably present in the first stage, will now be more emotional or include disagreements [22].

c) Stage 3 — Trust and Structure: During the third stage the structure is getting into place and the roles are now actually based on competence instead of status, power, or safety concerns. The communication patterns are more open and also more task-oriented. In this stage the role, organization, and process negotiations are most often more mature and there will be an evident clarification and consensus regarding the group goals. The group members also spend time solidifying positive relationships, and when the tasks are adjusted to competence the likelihood of goal achievement is higher. At this stage the leader's roll goes from needing to have been more directive to being more consultative. The communication structure is also more flexible (i.e. group members talk to whomever they need). Along the group development the content of the communication is also more and more task-oriented instead of relation-oriented. Groups always need the relation-oriented communication since we always need to do the maintenance of discussing how we work together as a group. Therefore, conflict will still occur but be over much faster since the group has better conflict management techniques. Work satisfaction and cooperation increase together with cohesion and trust. At this stage the individual commitment to the group goal will be higher (i.e. we care about what the group is doing on a personal level). This means we will see a voluntary conformity with the norms and helpful deviation from the group (like sub-grouping) will be accepted if considered helpful for the group as a whole [26].

d) Stage 4 — Work and Productivity: The forth stage of group development is when the group does even better with regards to the purpose of stage three. This means that the group focuses on getting the task done well together as well as maintaining group cohesion over a longer period of time. It is important to realize that there is a large set of variables that can and will disturb the group development. Basically, all changes will have such an effect, e.g. change

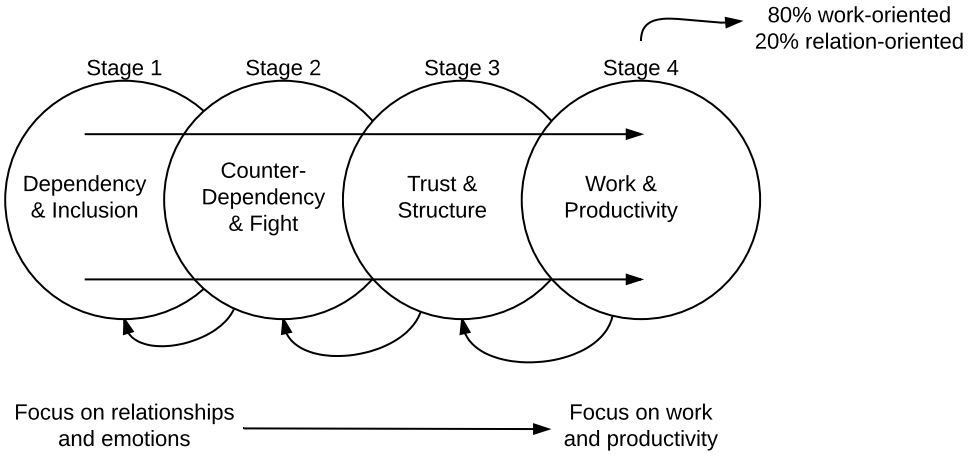


Fig. 1. The Group Development Stages [26]

of demands from the organization, losing staff, getting new staff, and so on and so forth. The challenge in stage four is therefore to try to maintain the effectiveness reached, and the most effective groups do not discuss task-related issues a hundred percent of the time, but actually, still spend around twenty percent discussing how to work together, which is key when maintaining high performance. The characteristics of the decision-making in such teams are participatory and the team encourages task-related conflicts, since they help finding better solutions to problems faced. A person who is or has been on such a team will recognize the intensity of the work and the effectiveness together with a very high interpersonal attraction between group members. People in such high performing teams often look at their work with excitement and joy and getting work done is easy and members have the feeling of being a part of the absolute best team in the world. Getting to stage four takes a lot of work both from internal group members but the group also needs to be given the right conditions from their surrounding ecosystem [26].

D. The Group Development Questionnaire (GDQ)

Wheelan [26] was not the first one who found these characteristic stages of group development, but she contributed with a tool to measure these different stages with four scales put together in a questionnaire. Her tool has made it possible to measure and therefore diagnose where a specific group is focusing its energy from a group developmental perspective. The survey has a total of 60 items and provides a powerful tool for research and interventions in teams. Scale four (GDQ4) is the "work and productivity" and has been shown to correlate with a set of effectiveness measures in different fields. Examples are that groups that have high scores on GDQ4 finish projects faster [27], students perform better on standardized test (SAT scores) if the faculty team scores high on GDQ4 [28], [29], and intensive care staff save more lives in surgery [30].

E. Technology under investigation

The group development stages have been well-known for many years in social psychology [10]. Helping teams to develop and mature in their group development have been shown to increase productivity and effectiveness in a diversity of fields (see Section II-D). Therefore, we want to see if helping teams to mature from a group psychology perspective also gets them to mature in their usage of the XP practices.

There are many group development models, but few have been scientifically validated in the way the group development questionnaire (GDQ) has [22]. Since correlations have been found between the group score on the "work and productivity" scale of the GDQ and other external effectiveness measurements, it would interesting to explore its effect on agile software development teams in their adoption of agile practices. Especially since agile methods have been shown to increase software development project success [31].

Evidence-based interventions within group development with the GDQ have been shown to increase the group maturity in teacher teams [32], and to increase the velocity of group development [33].

F. Relevance to practice

If group development training can be shown to increase the agility of software engineering teams, such aspects would be appropriate to explicitly integrate into the implementation of "agility" in all organizations conducting software development.

III. Experiment planning

A. Goals

The goal of this experiment was to see if training an agile team in group developmental psychology would increase their agility through more mature use of the agile practices.

B. Experimental units

In order to conduct an experiment the study was conducted with 43 students in an agile software development course at the University of São Paulo. Group developmental aspects apply to all group-work and therefore working with students as research subjects is a valid representation of software development conducted by developers on all knowledge levels. However, we would still be careful in generalizing a result to a larger

population than that of developers in the phase of learning an agile approach (i.e. individuals with little experience of agile software development in practice). The course were offered to 3rd year students, however, most students usually take the course in the 4th or 5th (last) year of their software engineering degree. The course is also open to graduate students who are given the possibility to take the course twice during their graduate education.

The student teams in this study comprised students enrolled in a project XP software development course called "The Laboratory of XP" at the Institute of Mathematics and Statistics at the University of São Paulo. The purpose of the course is to introduce agile methods through the use of XP. These methods included, at a minimum, the twelve practices presented in Section II-A0a. Some other staff at the university acted as customers and had to pitch their project ideas to the students, who signed up for the most interesting one from their point of view. All the teams included six to eight members and a more experienced student acting as a an agile coach for the team. The process was put together by the student teams themselves and we allowed any type of additional practices they selected as long as it was within the XP framework. As an example, we enforced collocation of a minimum of eight hours per week.

C. Experimental material

The experimental object was the agile software development team. Group norms and cooperation are set on group level and therefore the actual "team" is the relevant level of analysis.

D. Tasks

The experimental tasks applied in this experiment was for the teams (one team at a time) to listen and reflect on group development theory and discuss its applicability in connection to their own team.

E. Hypotheses, parameters, and variables

The construct used to measure agile practices and the behavior connected to them, was the mature usage of nine agile practices as defined by So and Scholl [19]:

Iterative Planning: (1) All members of the technical team actively participated during iteration planning meetings. (2) All technical team members took part in defining the effort estimates for requirements of the current iteration. (3) When effort estimates differed, the technical team members discussed their underlying assumption. (4) All concerns from team members about reaching the iteration goals were considered. (5) The effort estimates for the iteration scope items were modified only by the technical team members. (6) Each developer signed up for tasks on a completely voluntary basis. (7) The customer picked the priority of the requirements in the iteration plan.

Iterative Development: (1) We implemented our code in short iterations. (2) The team rather reduced the scope than delayed the deadline. (3) When the scope could not be implemented due to constraints, the team held active discussions on re-prioritization with the customer on what to finish within the iteration. (4) We kept the iteration deadlines. (5) At the end of an iteration, we delivered a potentially shippable product. (6) The software delivered at iteration end always met quality requirements of production code. (7) Working software was the primary measure for project progress.

Continuous Integration and Testing: (1) The team integrated continuously. (2) Developers had the most recent version of code available. (3) Code was checked in quickly to avoid code synchronization/integration hassles... (4) The implemented code was written to pass the test case. (5) New code was written with unit tests covering its main functionality. (6) Automated unit tests sufficiently covered all critical parts of the production code. (7) For detecting bugs, test reports from automated unit tests were systematically used to capture the bugs. (8) All unit tests were run and passed when a task was finished and before checking in and integrating. (9) There were enough unit tests and automated system tests to allow developers to safely change any code.

Stand-Up Meetings: (1) Stand up meetings were extremely short (max. 15 minutes). (2) Stand up meetings were to the point, focusing only on what had been done and needed to be done on that day. (3) All relevant technical issues or organizational impediments came up in the stand up meetings. (4) Stand up meetings provided the quickest way to notify other team members about problems. (5) When people reported problems in the stand up meetings, team members offered to help instantly.

Customer Access: (1) The customer was reachable. (2) The developers could contact the customer directly or through a customer contact person without any bureaucratic hurdles. (3) The developers had responses from the customer in a timely manner. (4) The feedback from the customer was clear and clarified his requirements or open issues to the developers.

Customer Acceptance Tests: (1) How often did you apply customer acceptance tests? (2) A requirement was not regarded as finished until its acceptance tests (with the customer) had passed. (3) Customer acceptance tests were used as the ultimate way to verify system functionality and customer requirements. (4) The customer provided a comprehensive set of test criteria for customer acceptance. (5) The customer focused primarily on customer acceptance tests to determine what had been accomplished at the end of an iteration.

Retrospectives: (1) How often did you apply retrospectives? (2) All team members actively participated in gathering lessons learned in the retrospectives. (3) The retrospectives helped us become aware of what we did well in the past iteration/s. (4) The retrospectives helped us become aware of what we should improve in the upcoming iteration/s. (5) In the retrospectives (or shortly afterwards), we systematically assigned all important points for improvement to responsible individuals. (6) Our team followed up intensively on the progress of each improvement point elaborated in a retrospective.

Collocation: (1) Developers were located majorly in... (2) All members of the technical team (including QA engineers, db admins) were located in... (3) Requirements engineers were located with developers in... (4) The project/release manager worked with the developers in... (5) The customer was located with the developers in...

The group maturity (or effectiveness) operationalization was done through using Scale 4 of the GDQ [22]. All the items in the GDQ scale cannot be shared here due to copyright, however, we can include three example items:

- The group gets, gives, and uses feedback about its effectiveness and productivity.
- The group acts on its decisions.
- This group encourages high performance and quality work.

The group development measurement on Scale 4 was assessed on a 5-point Likert scale (1 = low agreement to the statement and 5 = high agreement). The agile items were assesses on a 7-point Likert scale (1 = never and 7 =

always). These scales were used for the simple reason that these measurements were developed and validated using these exact scales.

Both measurements have been validated using a factor analysis [34] and a test for internal consistency (using the Cronbach's α [35]).

The formal research hypothesis for each scale is that the mean values for the scale is different between the two measurements, or $H_1 : \mu_1 \neq \mu_2$.

F. Design

We used a longitudinal research design in order to test differences in group mean value scores on the two measurements over time. The first measurement comprised seven teams and 43 student responses, and the second measurement comprised the same seven teams with 40 responses, i.e. three student were absent during the second measurement.

G. Procedure

The two measurement surveys were distributed to the teams five weeks into their software development projects (during their scheduled and collocated development sessions). The reason was to let the students actually form teams and have done some work before the first measurement. Three of the participating seven teams were randomized into the experimental group and the remaining four teams were used as a control group. The randomization was done by first writing the numbers "3" and "4" on paper slips and letting a person not connected to the experiment draw one folded slip for the research group (three groups were selected). The second step was conducted by writing all team names on other paper slips and letting the person draw three slips to be used for the research group.

On week six, the three selected teams participated in a 1.5-hour group development training with a discussion on the applicability to their own team. During the first hour of the training, the first author of this paper presented The Integrated Model of Group Development [10] and its four group developmental stages. The idea is, briefly, that there are predictable group developmental stages that all groups have to go through in order to work effectively. If team-members are aware of these there is a smaller probability of the team getting stuck on group issues, which leads to quicker and higher quality work [10]. Aspects covered were, for example, goal-setting, role clarification, decision-making, and leadership issues of groups in different development stages.

On week eleven, the second measurement was conducted using the same procedure as in the first measurement.

H. Analysis procedure

The data was analyzed using a general linear model for repeated measures (i.e. a standard repeated measures ANOVA). Such a model assumes normality in data, but since we did not find any significant result, we did not proceed to use non-parametric tests (since these are more restrictive and would therefore neither show any significance).

IV. Results

A. Descriptive statistics

Since we aimed at affecting the agile practices score by conducting group psychology training, we first looked at if we managed to increase the group dynamics score. Since that was not the case we already knew we did not succeed with the intended plan of the experiment. However, we still looked for differences in the agile practices measurement to see if they differed anyways between the two measurements. The only two significant differences we found between the first and second measurements were that the scales "Retrospectives" and "Customer Acceptance Tests." Therefore, we show the descriptive statistics for these scales as well (see Table I).

B. Data set preparation

A mean value was calculated based on the collected data for each individual, and then for the team according the agile practices as defined by So and Scholl [19]. The measured agile practices were: Iteration planning, Iterative development, Continuous integration and testing, Stand-up meetings, Customer access, Customer acceptance tests, Retrospectives and Collocation. The group development Scale 4 individual items were also turned into a mean value for each individual and then for the groups separately. Since we wanted to run the analysis on group-level we only have three mean values in the research group and four values in the control group (seven groups in total).

C. Hypothesis testing

Since we have so few data points, we cannot assess the population distribution based on our sample. However, other studies have shown this kind of data to be normally distributed [19], [22]. Also, since we did not find any significant results based on parametric tests, neither would we for non-parametric tests (since they are more restrictive). We began by testing the group effectiveness score (GDQ4 mean values) for the first and second measurements and can conclude that we did not see a significant effect (see Table II).

We then, still, ran the same analysis for all the agile practices and only found that the scales "Retrospectives" and "Customer acceptance tests" were different between the two measurements overall and not in connection to whether they were in the research group or not (see Table III and Table IV).

We conclude that the group development effectiveness measurement (GDQ Scale 4) was not different between the research group and the control group (not in the first nor the second measurement). The two agile practices "Retrospectives" and "Customer acceptance tests" where both different overall between the two measurements, but not depending on if the teams were in the research group or the control group.

V. Discussion

We did not find any of the expected results in this study. Clearly, just having 1.5 hours of training and discussion is not enough to help the group to develop, even if 1.5 hours of a workweek of 8 hours (like the students in the course had) would be equivalent to 7.5 hours of working full-time 40 hours a week. When taking a closer look at when other experiments

TABLE I. DESCRIPTIVE STATISTICS.

	Research Group	Mean	Std. Deviation	N
GDQ4 1st Measurement	Yes	3.9226	.22378	3
GDQ4 2nd Measurement	Yes	3.8579	.92728	3
GDQ4 1st Measurement	No	3.9007	.19832	4
GDQ4 2nd Measurement	No	4.0055	.33619	4
Retrospectives 1st Measurement	Yes	3.4963	1.54921	3
Retrospectives 2nd Measurement	Yes	5.8500	.42517	3
Retrospectives 1st Measurement	No	4.9280	1.05903	4
Retrospectives 2nd Measurement	No	5.9099	.54201	4
Cust. Accept. Tests 1st Measurement	Yes	3.6319	.38193	3
Cust. Accept. Tests 2nd Measurement	Yes	4.6400	.90598	3
Cust. Accept. Tests 1st Measurement	No	4.3349	.93600	4
Cust. Accept. Tests 2nd Measurement	No	4.8229	.91841	4

TABLE II. ANOVA FOR THE TWO REPEATED GDQ4 MEASUREMENTS.

Source	GDQ4	Type III Sum of Squares	df	Mean Square	F	Sig.
GDQ4	Linear	.001	1	.001	.010	.925
GDQ4 * research_group	Linear	.025	1	.025	.178	.691
Error(GDQ4)	Linear	.694	5	.139		

TABLE III. ANOVA FOR THE TWO REPEATED RETROSPECTIVES MEASUREMENTS.

Source	Retrospective	Type III Sum of Squares	df	Mean Square	F	Sig.
Retrospective	Linear	9.537	1	9.537	19.597	.007
Retrospective * research_group	Linear	1.613	1	1.613	3.314	.128
Error (Retrospective)	Linear	2.433	5	.487		

TABLE IV. ANOVA FOR THE TWO REPEATED CUSTOMER ACCEPTANCE TESTS MEASUREMENTS.

Source	Cat	Type III Sum of Squares	df	Mean Square	F	Sig.
Cat	Linear	1.919	1	1.919	7.018	.045
Cat * research_group	Linear	.232	1	.232	.848	.399
Error(Cat)	Linear	1.367	5	.273		

succeeded in significantly helping the groups to develop and capture their increased maturity, it turns out they did a much larger intervention then was applied in this study. Jacobsson and Wramsten Wilmar [32], for example, gave the groups eight different interventions of group development assessment and enforced improvement points that the group had to work on until the next workshop, plus the students were fully dedicated to only one course. In hindsight, we probably would have needed something similar in order to move the groups forward in the research group. There is also, of course, the possibility of software engineering teams' agility not being as dependent on group maturity as we might think.

A. Threats to validity

We believe the layout of the experiment has potential. Of course, even if we would have found a significant difference, we would still have to have been careful when generalizing the results due to the very small sample size. Regarding construct validity the first author was present during the data collection and could answer any potential questions regarding the questionnaire. However, since we did not want the control group to get training of group development we provided all participants with as little information as possible before the first survey, since we did not want to introduce bias. The trade-off is of course that the participants could have misinterpreted the questions and failed to answer in connection to our intended operationalization of constructs. Regarding learning effects between measurement, the GDQ has been shown to be stable for repeated measurements as such [22]. We have no such studies for the agile practices, which means that we might have seen a learning effect when students answer that part of the survey.

In order to prevent hypothesis guessing, we only informed the participant that the research was about looking at connections between group psychology and agile practices and not more detail on how we expected them to be connected. The internal validity is considered quite high in this experiment since we used validated scales as defined and validated quantitatively by other researchers [19], [22]. However, inter-group communication between the research groups and the control groups is also a threat our experimental research design.

We draw no inference from this experiment. We do not want to state that group development causes more mature use of agile practices, nor the opposite.

B. Lessons learned

The largest lesson learned from this experiment is evidently to check the level of intervention effort needed to move groups forward in their development before conducting this kind of an experiment. We still do not known the effort needed, but the span is more then one 1.5 hours workshop with a second measurement two months later, and less than six to eight workshops of 2–3 hours during a full year with connected action plans and follow-up. By having more time with the teams we could have focused even more concretely on, for example, goal-setting, role clarification, decision-making, functional sub-grouping, or leadership issues, like in [32].

VI. Conclusions and future work

We obtained an insignificant result of this experiment. We therefore have no conclusions to draw based on the expected effects. However, we believe these concepts could still be connected since agile software development is based on teamwork to a large extent. We evidently need a larger intervention effort and, of course there could also be more confounding or mediating factors we have not thought of in the context of agile software development teams.

We would like to redo this experiment with more resources and be able to give the teams in the research group eight times more workshops with connected action plans in order to see if we can get a similar effect as has been shown with teacher teams [32]. It would, of course, be advantageous to include as many teams as possible and at multiple universities and companies to increase the statistical power of the experiment.

References

[1] H. Takeuchi and I. Nonaka, "The new new product development game," *Harvard business review*, vol. 64, no. 1, pp. 137–146, 1986.

[2] J. Sutherland, *Scrum: The art of doing twice the work in half the time*. Random House Business, 2014.

[3] K. Schwaber and M. Beedle, *Agile software development with scrum*. Upper Saddle River, NJ: Prentice Hall, 2002.

[4] G. Melnik and F. Maurer, "Direct verbal communication as a catalyst of agile knowledge sharing," in *Agile Development Conference, 2004*. IEEE, 2004, pp. 21–31.

[5] A. Teh, E. Baniassad, D. Van Rooy, and C. Boughton, "Social psychology and software teams: Establishing task-effective group norms," *IEEE Software*, vol. 29, no. 4, pp. 53–58, 2012.

[6] P. Lenberg, R. Feldt, and L.-G. Wallgren, "Human factors related challenges in software engineering: An industrial perspective," in *Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering*.

[7] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to advanced empirical software engineering*. Springer, 2008, pp. 201–228.

[8] J. Iivari and N. Iivari, "The relationship between organizational culture and the deployment of agile methods," *Information and Software Technology*, vol. 53, no. 5, pp. 509–520, 2011.

[9] C. Tolfo and R. Wazlawick, "The influence of organizational culture on the adoption of extreme programming," *Journal of systems and software*, vol. 81, no. 11, pp. 1955–1967, 2008.

[10] S. Wheelan, *Group processes: A developmental perspective*, 2nd ed. Boston: Allyn and Bacon, 2005.

[11] L. Gren, R. Torkar, and R. Feldt, "Group maturity and agility, are they connected? A survey study," in *Proceedings of the 41st EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, 2015.

[12] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Information and software technology*, vol. 50, no. 9, pp. 833–859, 2008.

[13] D. Cohen, M. Lindvall, and P. Costa, "An introduction to agile methods," *Advances in Computers*, vol. 62, pp. 1–66, 2004.

[14] K. Schwaber, "Scrum development process," in *Business Object Design and Implementation*. Springer, 1997, pp. 117–134.

[15] W. Feld, *Lean manufacturing: Tools, techniques, and how to use them*. Boca Raton, Fla.: St. Lucie Press, 2001.

[16] M. Poppendieck, "Lean software development," in *Companion to the proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 165–166.

[17] C. Ladas, "Scrumban," *Lean Software Engineering-Essays on the Continuous Delivery of High Quality Information Systems*, 2008.

[18] A. Cockburn, *Agile software development: The cooperative game*, 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2007.

[19] C. So and W. Scholl, "Perceptive agile measurement: New instruments for quantitative studies in the pursuit of the social-psychological effect of agile practices," in *Agile Processes in Software Engineering and Extreme Programming*. Springer, 2009, pp. 83–93.

[20] J. Keyton, *Communicating in groups: Building relationships for group effectiveness*. New York: McGraw-Hill, 2002.

[21] S. Wheelan, "Group size, group development, and group productivity," *Small Group Research*, vol. 40, no. 2, pp. 247–262, 2009.

[22] S. Wheelan and J. Hochberger, "Validation studies of the group development questionnaire," *Small Group Research*, vol. 27, no. 1, pp. 143–170, 1996.

[23] S. Cruz, F. da Silva, and L. Capretz, "Forty years of research on personality in software engineering: A mapping study," *Computers in Human Behavior*, vol. 46, pp. 94–113, 2015.

[24] S. Wheelan and R. Mckeage, "Developmental patterns in small and large groups," *Small Group Research*, vol. 24, no. 1, pp. 60–83, 1993.

[25] B. Tuckman and M. Jensen, "Stages of small-group development revisited," *Group & Organization Management*, vol. 2, no. 4, pp. 419–427, 1977.

[26] S. Wheelan, *Creating effective teams: A guide for members and leaders*, 4th ed. Thousand Oaks: SAGE, 2013.

[27] S. Wheelan, D. Murphy, E. Tsumura, and S. F. Kline, "Member perceptions of internal group dynamics and productivity," *Small Group Research*, vol. 29, no. 3, pp. 371–393, 1998.

[28] S. Wheelan and F. Tilin, "The relationship between faculty group development and school productivity," *Small group research*, vol. 30, no. 1, pp. 59–81, 1999.

[29] S. Wheelan and J. Kesselring, "Link between faculty group: Development and elementary student performance on standardized tests," *The journal of educational research*, vol. 98, no. 6, pp. 323–330, 2005.

[30] S. Wheelan, C. N. Burchill, and F. Tilin, "The link between teamwork and patients' outcomes in intensive care units," *American Journal of Critical Care*, vol. 12, no. 6, pp. 527–534, 2003.

[31] P. Serrador and J. K. Pinto, "Does agile work? – A quantitative analysis of agile project success," *International Journal of Project Management*, vol. 33, no. 5, pp. 1040–1051, 2015.

[32] C. Jacobsson and M. Wramsten Wilmar, "Increasing teacher team effectiveness by evidence based consulting," in *Proceedings of the 14th European Congress of Work and Organizational Psychology (EAWOP)*, May 13–16 2009.

[33] C. Jacobsson and O. Persson, "Group development, what's the speed limit? – Two cases of student groups," in *Proceedings of the 7th Nordic Conference of Group and Social Psychology (GRASP)*, May 20–21 2010.

[34] L. Fabrigar and D. Wegener, *Exploratory Factor Analysis*, ser. Series in understanding statistics. OUP USA, 2012.

[35] L. Cronbach, "Coefficient alpha and the internal structure of tests," *Psychometrika*, vol. 16, no. 3, pp. 297–334, 1951.

Predicting Quality of Service (QoS) Parameters using Extreme Learning Machines with Various Kernel Methods

Lov Kumar
NIT Rourkela, India
lovkumar505@gmail.com

Santanu Kumar Rath
NIT Rourkela, India
skrath@nitrkl.ac.in

Ashish Sureka
ABB Corporate Research, India
ashish.sureka@in.abb.com

***Abstract*—Web services which are language and platform independent self-contained web-based distributed application components represented by their interfaces can have different Quality of Service (QoS) characteristics such as performance, reliability and scalability. One of the major objectives of a web service provider and implementer is to be able to estimate and improve the QoS parameters of their web service as its clients application are dependent on the overall quality of the service. We hypothesize that the QoS parameters have a correlation with several source code metrics and hence can be estimated by analyzing the source code. We investigate the predictive power of 37 different software metrics (Chidamber and Kemerer, Harry M. Sneed, Baski & Misra) to estimate 15 QoS attributes. We develop QoS prediction models using Extreme Learning Machines (ELM) with various kernel methods. Since the performance of the classifiers depends on the software metrics that are used to build the prediction model, we also examine two different feature selection techniques i.e., Principal Component Analysis (PCA), and Rough Set Analysis (RSA) for dimensionality reduction and removing irrelevant features. The performance of QoS prediction models are compared using three different types of performance parameters i.e., MAE, MMRE, RMSE. Our experimental results demonstrate that the model developed by extreme learning machine with RBF kernel achieves better results as compared to the other models in terms of the predictive accuracy.**

***Index Terms*—Extreme Learning Machines, Predictive Modeling, Quality of service (QoS) Parameters, Software Metrics, Source Code Analysis, Web Service Definition Language (WSDL)**

I. Research Motivation and Aim

Web services are distributed web application components which can be implemented in different languages, deployed on different client and server platforms, are represented by interfaces and communicate using open protocols [1][2]. Web service implementers and providers need to comply with common web service standards so that they can be language and platform independent and can be discovered and used by other applications [1][2]. Applications and business solutions using web services (which integrate and combine several services) expect high Quality of Service (QoS) such as performance, scalability and reliability as their application is dependent on the service. Measuring quality of service attributes and characteristics of web services and understanding their relationship with source code metrics can help developers control and estimate maintainability by analyzing the source code [3][4][5][6]. The work presented in this paper is motivated by the need to investigate the correlation between QoS attributes such as response time, availability, throughput, reliability, modularity, testability and interoperability and source code metrics such as classic object oriented metrics (Chidamber and Kemerer) as well as other well-known metrics such as Baski & Misra and Harry M. Sneed metrics. Specifically, our research aim is to study the correlation between 15 web service quality attributes and 37 source code metrics and then build machine learning based predictive models for estimating the quality of a given service based on the computed source code metrics. Our aim is to conduct experiments on a real-world dataset and also examine the extent to which feature selection techniques such as Principal Component Analysis (PCA) and Rough Set Analysis (RSA) can be used for dimensionality reduction and filter irrelevant features.

II. Related Work, Research Contributions and Research Framework

Related Work: Coscia et al. investigate the potential of obtaining more maintainable services by exploiting Object-Oriented metrics (OO) values from the source code implementing services [3]. Their approach proposed the use of OO metrics as early indicators to guide software developers towards obtaining more maintainable services [3]. Coscia and Crasso et al. present a statistical correlation analysis demonstrating that classic software engineering metrics (such as WMC, CBO, RFC, CAM, TPC, APC and LCOM) can be used to predict the most relevant quality attributes of WSDL documents [4]. Mateos et al. found that there is a high correlation between well-known object-oriented metrics taken in the code implementing services and the occurrences of anti-patterns in their WSDLs [5]. Kumar et al. use different object-oriented software metrics and Support Vector Machines with different type of kernels for predicting maintainability of services [6]. Their experimental results demonstrate that maintainability of SOC paradigm can be predicted by application of 11 object-oriented metrics [6]. Olatunji et al. develop an extreme learning machine (ELM) maintainability prediction model for objectoriented software systems [7].

Research Contributions: The main research contribution of the study presented in this paper is the application of 37 source code metrics (Chidamber and Kemerer, Harry M. Sneed, Baski & Misra) for predicting 15 Quality of Service (QoS) or maintainability parameters for web services by employing Extreme Learning Machines (ELM) using various kernel methods and two feature selection techniques (Principal Component Analysis and Rough Set Analysis). To the best of our knowledge, the research presented in this paper is the first such in-depth empirical study on publicly available well-known dataset.

Research Framework: Figure 1 displays our research framework and methodology. The framework consists of multiple steps. As shown in Figure 1, we first compute the QoS parameters for the web services in our dataset. We compute 37 source code metrics belonging to 3 different metrics suite. We apply two different feature selection methodology (Rough Set Analysis and Principal Component Analysis) for the purpose of dimensionality reduction and removing irrelevant features. We apply Extreme Learning Machines (ELM) with three different kernel functions (linear, polynomial and RBF). We create 6 sets of metrics suite, 2 feature selection techniques and 3 kernel functions and evaluate the performance of all the combinations resulting in a comprehensive and in-depth experimental evaluation. Finally, we evaluate the performance of various models using wide used estimator evaluation metrics and conduct statistical tests to identify best learning algorithms.

III. Experimental Dataset

We use a subset of QWS Dataset[1] for our experimental analysis. The QWS Dataset provided by Al-Masri et al. includes a sets of 2507 Web services and their 9 QWS parameters (such as response time, availability, throughput, compliance and latency) which are measured using Web service benchmark tools [8][9]. Al-Masri et al. collect the Web services using their Web Service Crawler Engine (WSCE) and majority of the Web services are obtained from public sources. We observe that 524 out of 2507 Web Service have their corresponding WSDL file. Baski et al. present a suite of metrics to evaluate the quality of the XML web service in terms of its maintainability [10]. We apply the Baski and Misra metrics suite tool on the 524 WSDL files and obtained successful parsing for 200 files. We use the metrics proposed by Baski et al. as predictor variables. We could not include 324 WSDL files as part of our experimental dataset as we were unable to parse them for computing Baski and Misra metrics. Hence, we finally use 200 Web services for the experiments presented in this paper. Redistribution of the data on the web is not permitted according to the dataset usage guidelines and hence we provide a list[2] of the 200 Web services used in our study so that our research can be reproduced and replicated for benchmark or comparison. Figure 2 shows a scatter plot for the number of Java files for the 200 WSDL files in our dataset. The X-axis represents the WSDL File ID and the Y-axis represent the number of Java files. Figure 2 shows that there are several web services implemented using more than 100 Java files.

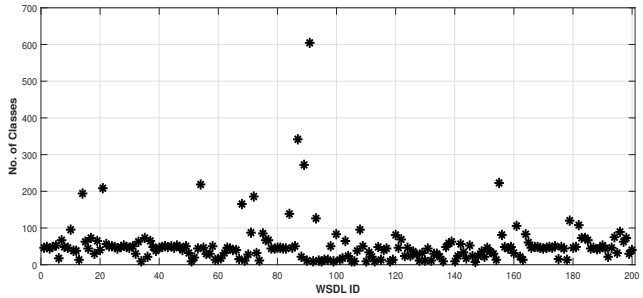


Fig. 2. Scatter Plot for the Number of Java Files for the 200 WSDL Files in Experimental Dataset

[1]http://www.uoguelph.ca/~qmahmoud/qws/

[2]http://bit.ly/1S8020w

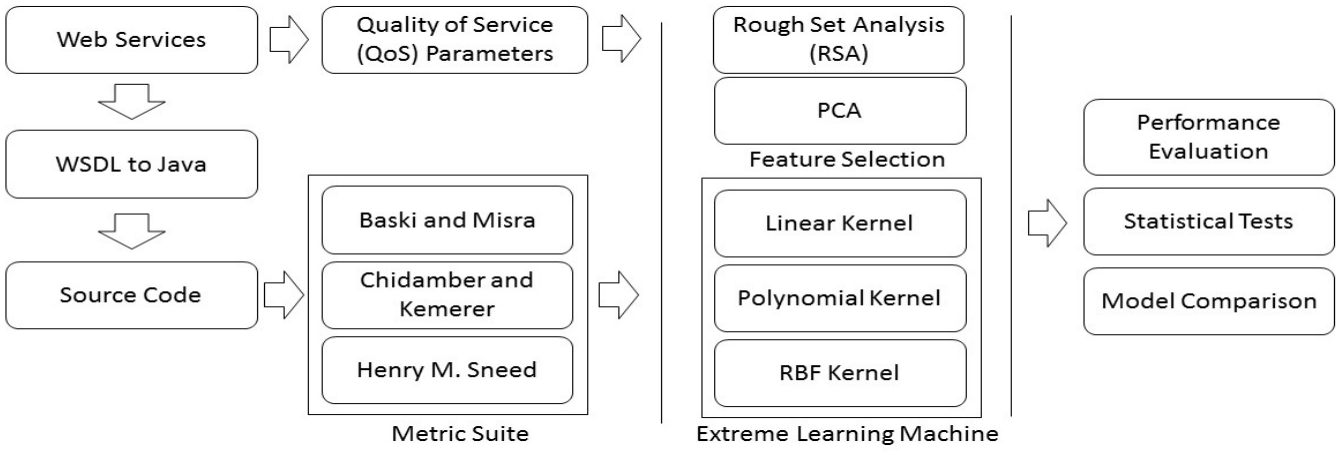


Fig. 1. Research Methodology and Framework

IV. Dependent Variables QoS Parameters

Table I shows the descriptive statistics of 9 QoS parameters provided by the creators of QWS dataset. The owners of QWS dataset provide QoS parameter values for all the 2507 web services. However, Table I displays the descriptive statistics computed by us for the 200 web services used in our experimental dataset. Table I reveals substantial variation or dispersion in the parameter values across 200 web services which shows variability in the quality across services. Sneed et al. describes a tool supported method for measuring web service interfaces [11]. The extended version of their tool can be used to compute maintainability, modularity, reusability, testability, interoperability and conformity of web services. We calculate these values for the 200 web services in our dataset and assign them as dependent variables. Table II displays the descriptive statistics for the QoS parameters calculated using Sneed's Tool. Hence, we have a total of 15 dependent variables.

TABLE I
Descriptive statistics of QoS parameters provided by QWS dataset

Parameter	Min	Max	Mean	Median	Std Dev	Skewness	Kurtosis
Response Time	57.00	1664.62	325.11	252.20	289.33	3.15	13.12
Availability	13.00	100.00	86.65	89.00	12.57	-2.63	12.06
Throughput	0.20	36.90	7.04	4.00	6.94	1.57	5.79
Successability	14.00	100.00	90.19	96.00	13.61	-2.57	10.81
Reliability	33.00	83.00	66.64	73.00	9.61	-0.60	2.96
Compliance	67.00	100.00	92.19	100.00	9.78	-0.90	2.61
Best Practices	57.00	93.00	78.81	82.00	7.70	-0.68	2.67
Latency	0.74	1337.00	42.81	12.20	106.23	9.56	112.68
Documentation	1.00	96.00	29.37	32.00	26.97	1.06	3.31

TABLE II
Descriptive statistics of QoS parameters calculated using Sneeds Tool

Parameter	Min	Max	Mean	Median	Std Dev	Skewness	Kurtosis
Maintainability	0.00	77.67	31.07	28.17	24.29	0.37	2.02
Modularity	0.10	0.81	0.22	0.17	0.13	2.02	7.10
Reusability	0.10	0.90	0.38	0.35	0.17	0.32	2.94
Testability	0.10	0.66	0.19	0.16	0.09	2.58	10.71
Interoperability	0.14	0.90	0.51	0.41	0.23	0.65	2.01
Conformity	0.43	0.98	0.79	0.87	0.15	-0.47	1.57

V. Predictor Variables - Source Code Metrics

Chidamber and Kemerer Metrics: We compute several size and structure software metrics from the bytecode of the compiled Java files in our experimental dataset using CKJM extended[3] [12][13]. CKJM extended is an extended version of tool for calculating Chidamber and Kemerer Java Metrics and many other metrics such as weighted methods per class, coupling between object classes, lines of code, measure of functional abstraction, average method complexity and McCabe's Cyclomatic Complexity. We use the WSDL2Java Axis2 code generator[4] which comes built-in with an Eclipse plug-in to generate Java class files from the 200 WSDL files in our experimental dataset. We then compile the Java files to generate the bytecode for computing the size and structure software metrics using the CKJM extended tool. The minimum number of Java files are 7 and the maximum is 605. The mean, median, standard deviation, skewness and kurtosis is 52.39, 45.50, 59.06, 5.43 and 43.94 respectively. Table III displays the descriptive statistics for 19 size and structure software metrics computed using CKJM Extended Tool for the 200 Web services in our dataset. The mean value of AMC as 61.94 means that the mean of the average method size calculated in terms of the number of Java binary codes in the method for each class is 62. We compute the standard deviation for all the 19 metrics to quantify the amount of dispersion and spread in the values. We observe (refer to Table III) that few metrics such as DIT, NOC, MFA, CAM, IC and CBM have low standard deviation which means that the data points are close to the mean. However, we observe that LCOM, LCO, AMC and CC have relatively high values of standard deviation which means that the data points are dispersed over a wider range of values.

TABLE III
Descriptive statistics of Object-Oriented Metrics

Metrics	Min	Max	Mean	Median	Std Dev	Skewness	Kurtosis
WMC	9.48	13.57	11.01	10.96	0.48	0.81	6.54
DIT	0.87	1.02	0.98	0.98	0.02	-2.07	9.72
NOC	0.00	0.13	0.01	0.01	0.02	2.64	12.55
CBO	4.10	12.55	10.70	11.01	1.33	-1.15	5.18
RFC	12.78	44.55	40.35	41.48	4.13	-3.13	15.15
LCOM	74.03	405.49	120.94	108.67	45.70	2.99	13.96
Ca	0.64	3.92	2.91	2.99	0.62	-0.50	2.72
Ce	3.49	9.50	8.24	8.37	0.90	-1.30	6.01
NPM	4.88	9.27	6.55	6.47	0.48	1.07	7.90
LCOM3	1.18	1.50	1.32	1.31	0.06	0.27	2.75
LCO	76.14	493.64	399.18	411.50	54.03	-2.61	11.71
DAM	0.21	0.45	0.37	0.37	0.04	-0.45	4.53
MOA	0.02	2.28	0.60	0.53	0.28	2.00	11.03
MFA	0.00	0.02	0.00	0.00	0.00	1.92	8.43
CAM	0.39	0.43	0.40	0.40	0.01	0.22	4.54
IC	0.00	0.05	0.01	0.01	0.01	0.79	3.68
CBM	0.00	0.05	0.01	0.01	0.01	0.79	3.68
AMC	7.68	82.86	61.94	64.37	10.75	-1.69	6.97
CC	18.17	71.39	42.77	43.74	9.58	-0.29	3.67

TABLE IV
Descriptive statistics of Harry M. Sneed's Metrics Suite

Metrics	Min	Max	Mean	Median	Std Dev	Skewness	Kurtosis
Data Complexity	0.10	0.81	0.28	0.27	0.17	0.60	2.59
Relation Complexity	0.10	0.90	0.87	0.90	0.07	-7.72	83.58
Format Complexity	0.14	0.72	0.60	0.64	0.09	-1.05	5.24
Structure Complexity	0.15	0.90	0.61	0.63	0.17	-0.13	2.63
Data Flow Complexity	0.10	0.90	0.87	0.90	0.10	-5.64	39.14
Language Complexity	0.16	0.88	0.61	0.56	0.21	0.03	1.78
Object Point	42.00	4581.00	299.32	200.00	483.31	5.67	41.67
Data Point	29.00	3124.00	222.75	152.00	347.48	5.16	34.85
Function Point	6.00	776.00	53.73	32.00	94.21	5.36	35.33
Major Rule Violation	2.00	109.00	26.39	10.00	26.13	0.62	1.97
Medium Rule Violation	2.00	16.00	5.02	5.00	1.94	0.56	6.71
Minor Rule Violation	2.00	586.00	49.51	35.50	63.14	4.26	30.60

Harry M. Sneed Metrics: Sneed's tool implements metrics for quantity, quality and complexity of web service interfaces. The values of all the metrics are statically computed from a service interface in WSDL as the suite of metrics is based on the WSDL schema element occurrences [11]. We compute six interface complexity metrics for all the 200 web services in our dataset. The six interface complexity metrics are computed between a scale of 0.0 to 1.0. A value between 0.0

[3]http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

[4]https://axis.apache.org/axis2/java/core/tools/eclipse/wsdl2java-plugin.html

TABLE V
DESCRIPTIVE STATISTICS OF BASKI AND MISRA METRICS SUITE

Metrics	Min	Max	Mean	Median	Std Dev	Skewness	Kurtosis
OPS	0.00	108.00	7.76	5.00	13.74	5.41	35.37
DW	0.00	2052.00	114.63	62.00	216.46	6.56	53.80
MDC	0.00	17.00	4.40	5.00	2.62	1.97	8.87
DMR	0.00	1.00	0.53	0.50	0.22	0.50	3.32
ME	0.00	3.80	1.73	2.12	0.73	-0.43	3.58
MRS	0.00	72.00	3.65	2.60	6.25	7.99	79.12

and 0.4 represents low complexity and a value between 0.4 and 0.6 indicates average complexity. A value of more than 0.6 falls in the range of high complexity wherein any value above 0.8 reveals that there are major issues with the code design [4][11]. Table IV shows the minimum, maximum, mean, median and standard deviation of the size complexity values for all the web services on our dataset. In addition to 6 interface complexity metrics, we measure 6 more metrics using the extended version of the tool provided to us by the author himself: object point, data point, function point, major, medium and minor rule violations. Table IV displays the descriptive statistics for the 12 metrics for all the web services in our dataset.

Baski and Misra Metrics: We compute 6 metrics proposed by Baski and Misra [10]. Their metrics are based on the analysis of the structure of the exchanged messages described in WSDL which becomes the basis for measuring the data complexity. Their metric suitbe is based on WSDL and XSD schema elements occurrences. Table V reveals the descriptive statistics of the 6 metrics: Data Weight of a WSDL (DW), Distinct Message Ratio (DMR), Distinct Message Count (DMC), Message Entropy (ME), Message Repetition Scale (MRS) and Operations Per Service (OPS).

VI. CODE METRICS - CORRELATION ANALYSIS

We compute the association between 37 metrics consisting of dependent and independent variables using the Pearson's correlations coefficient (r). The coefficient of correlation r measures the strength and direction of the linear relationship between two variables. Figure 3 displays our experimental results on correlation analysis between the 37 metrics. In Figure 3, a Black circle represents an r value between 0.7 and 1.0 indicating a strong positive linear relationship. A white circle r rvalue between 0.3 and 0.7 indicate a weak positive linear relationship. A black square r represents a value between -1 and -0.7 indicating a strong negative linear relationship. A white square r represents a value between -0.7 and -0.3 indicating a weak negative linear relationship. A blank circle represents no linear relationships between the two variables. For example, based on Figure 3, we infer that there is a strong positive linear relationship between OPS and four other variables MRS, OP, DP and FP. On the other hand, we observe a weak linear relationship between ILC and IDC as well as ISC and IDC. Figure 3 reveals association between different suite of metrics and not just associations between metrics within the same suite. For example, DMR is part of Baski and Misra metrics suite. DMR has a strong negative correlation with ISC (Structure Complexity), OP (Object Point), DP (Data Point), FP (Function Point) and MERV (Medium Rule Violation) which is part of Harry M. Sneed metrics suite.

VII. FEATURE EXTRACTION AND SELECTION USING PCA AND RSA

We investigate the application of Principal Component Analysis (PCA) and Rough Set Analysis (RSA) as a data preprocessing step for feature extraction and selection [14]. Our

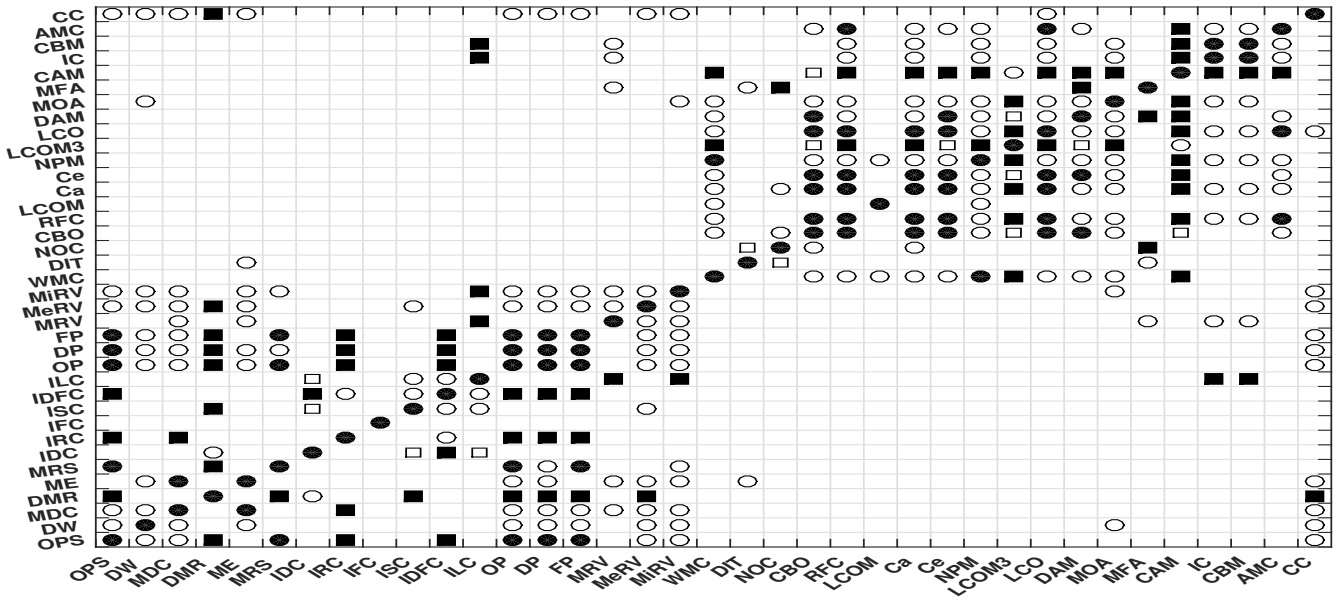


Fig. 3. Pearson's Correlation Coefficient between 37 Metrics

objective behind using PCA and RSA is to identify features which are relevant in-terms of high predictive power and impact on the dependent variable and filter irrelevant features which have little or no impact on the classifier accuracy [14]. We apply PCA and varimax rotation method on all source code metrics. The experimental results of PCA analysis is shown in Table VI. Table VI reveals the relationship between the original source code metrics and the domain metrics. For each PC (Principal Components), we provide the eigenvalue, variance percent, cumulative percent and source code metrics interpretation (refer to Table VI). In PCA the order of the eigenvalues from highest to lowest indicates the principal components in the order of significance. Among all Principal Components, we select only those which have Eigen value greater than 1. Our analysis reveals that 9 PCs have Eigen value greater than 1 (refer to Table VI). Table VI shows the mapping of each component to the most important metric for that component. Table VII shows the optimal subset of features for every dependent variable derived from the original set of 37 source code metrics based features after applying RSA. We apply the RSA procedure 15 times (one for each dependent variable). Table VII reveals that it is possible to reduce the number of features substantially and several features from the original set are found to be uncorrelated.

TABLE VI
FEATURE EXTRACTION USING PRINCIPAL COMPONENT ANALYSIS - DESCRIPTIVE STATISTICS

PC	Eigenvalue	% variance	Cumulative %	Metrics Interpretation
PC1	6.4	17.3	17.3	CBO, RFC, Ca, Ce, LCOM3, LCO, DAM, CAM
PC2	5.8	15.76	33.06	OPS, MRS, IRC, IDFC, OP, DP, FP
PC3	3.67	9.94	43.00	DW, MDC, MeRV, MiRV, CC, ME
PC4	3.39	9.16	52.17	DMR, IDC,ISC, ILC
PC5	3.34	9.03	61.2	IC, CBM, MOA
PC6	2.5	6.77	67.98	IFC, DIT, NOC, MFA
PC7	2.23	6.02	74.00	WMC, NPM
PC8	2.14	5.79	79.79	MRV, AMC
PC9	1.36	3.7	83.5	LCOM

TABLE VII
SOURCE CODE METRICS (FEATURE) SELECTION OUTPUT USING ROUGH SET ANALYSIS (RSA)

QoS	Selected Metrics
Response Time	DMR, SC, LC, WMC, Ca, LCOM3, MFA, CAM, IC, CC
Availability	FC, SC, LC, MeRV, MiRV, WMC, Ca, LCOM3, MFA, CAM, IC, CC
Throughput	ME, FC, SC, LC, MRV, MeRV, MiRV, Ce, MOA, MFA, CAM, CBM, CC
Successability	ME, FC, SC, DFC, LC, MRV, WMC, LCOM3, LCO, DAM, MOA, CAM
Reliability	FC, SC, DFC, LC, WMC, LCOM3, LCO, MOA, MFA, CAM, CBM
Compliance	ME, FC, SC, DFC, LC, MRV, WMC, MiRV, Ca, CC, DAM, MOA, CAM, NPM
Best Practices	ME, FC, SC, DFC, LC, MRV, MiRV, WMC, Ca, NPM, MOA, MFA, CAM, CC
Latency	DMR, ME, DC, FC, DFC, LC, MRV, NOC, NPM, LCO, MOA, CAM, IC
Documentation	ME, FC, SC, DFC, LC, MRV, MeRV, WMC, Ca, NPM, CAM, IC, CC
Maintainability	DP, Ce, LCOM3, MOA, MFA, CAM, CBM
Modularity	DMR, ME, SC, DFC, LC, DP, MRV, MiRV, WMC, Ca, MOA, IC, AMC
Reusability	MDC, DMR, FC, SC, DFC, LC, LCOM, LCOM3
Testability	ME, FC, SC, LC, MiRV, DIT, NOC, CC, RFC
Interoperability	SC, LC, MeRV, MiRV, WMC, DIT, CBO, MFA, CC
Conformity	ME, FC, DFC, LC, MRV, WMC, Ca, CAM

VIII. APPLICATION OF EXTREME LEARNING MACHINES (ELMS)

Huan et al. mention that Extreme Learning Machines (ELMs) have shown to outperform computational intelligence techniques such as Artificial Neural Networks (ANNs) and Support Vector Machines (SVMs) in-terms of learning speed and computational scalability [15]. ELM has demonstrated good potential to resolving regression and classification problems [15] and our objective is to investigate if ELMs can be successfully applied in the domain of web service QoS prediction using source code metrics. Selection of an appropriate kernel function depending on the application domain and dataset is an important and core issue [16].

Ding et al. mention that there is a correlation between the generalization performance and learning performance with the kernel function [16] as in the case of traditional neural networks [16]. Hence, we investigate the performance of the ELM based classifier using three different kernel functions: linear, polynomial and RBF. ELMs can be used with different kernel functions and one can create hybrid kernel functions also. The most basic, simplest and fastest is the linear kernel function which is used as a baseline for comparison with more complicated kernel functions such as polynomial and RBF. Table VIII shows the performance of the ELM based classifier with linear kernel function. Table IX shows the performance of the ELM based predictive model with second degree polynomial kernel. The polynomial kernel is more sophisticated than the linear kernel and uses non-linear equations instead of the linear equations for the purpose of regression and classification and is expected to result in better accuracy in comparison to the classifier with linear kernel. The Radial Basis Function kernel (RBF or Gaussian) is a popular kernel function and widely used in Support Vector Machine (SVM) learning algorithm. We use linear kernel to investigate if the data is linearly separable but also use polynomial and RBF kernel to examine if our data is not linearly separable (computing a non-linear decision boundary). We employ four different performance metrics (MAE, MMRE, RMSE and r-value) to study the accuracy of the classifiers. The Mean Absolute Error (MAE) measures the difference between the predicted or forecasted value and the actual values (average of the absolute errors). Table VIII and IX reveals that the forecast for several predictive model is very accurate as the MAE value is less than 0.05. For example, the MAE value for HMS, AM and PCA metrics for predicting Conformity is 0.03. Table VIII and IX reveals that in general the predictive accuracy for response time, latency, modularity and conformity is better than the predictive accuracy of other QoS parameters.

Kitchenham et al. mention that Mean Magnitude of Relative Error (MMRE) is a widely used assessment criterion for evaluating the predictive accuracy and overall performance of competing software prediction models and particularly the software estimation models [17]. MMRE computes the difference between actual and predicted value relative to the actual value. Table X shows that the MMRE values for ELM with RBF kernel is between 0.30 to 0.35 for response time, availability and successability and indicates good estimation ability of the classifier. Table VIII and Table IX reveals that the MMRE values for conformity QoS parameter are as low as 0.05, 0.06, 0.10 and 0.11. Root Mean Square Error (RMSE) or Root Mean square Deviation root-mean-

TABLE VIII
PERFORMANCE MATRIX FOR ELM WITH LINEAR KERNEL

	Response Time	Availability	Throughput	Successability	Reliability	Compliance	Best Practices	Latency	Documentation	Maintainability	Modularity	Reusability	Testability	Interoperability	Conformity
	MAE														
BMS	0.11	0.19	0.18	0.21	0.25	0.28	0.19	0.11	0.23	0.20	0.11	0.16	0.12	0.16	0.18
HMS	0.10	0.16	0.18	0.19	0.25	0.27	0.19	0.11	0.22	0.15	0.08	0.15	0.07	0.14	0.06
OOM	0.09	0.14	0.18	0.17	0.25	0.25	0.18	0.11	0.22	0.12	0.15	0.15	0.14	0.22	0.18
AM	0.10	0.15	0.18	0.18	0.26	0.27	0.20	0.11	0.21	0.11	0.07	0.13	0.06	0.11	0.06
PCA	0.10	0.16	0.19	0.18	0.26	0.26	0.21	0.12	0.21	0.16	0.09	0.14	0.10	0.14	0.09
RSA	0.11	0.17	0.18	0.20	0.25	0.26	0.19	0.11	0.23	0.27	0.16	0.17	0.15	0.25	0.26
	MMRE														
BMS	0.37	0.35	0.73	0.38	0.66	0.61	0.37	0.55	0.90	0.76	0.41	0.61	0.46	0.35	0.28
HMS	0.36	0.33	0.73	0.36	0.69	0.61	0.39	0.56	0.93	0.60	0.29	0.60	0.26	0.31	0.11
OOM	0.34	0.32	0.74	0.35	0.71	0.61	0.37	0.55	0.89	0.47	0.54	0.57	0.55	0.50	0.33
AM	0.36	0.34	0.71	0.37	0.71	0.64	0.47	0.54	0.90	0.42	0.22	0.46	0.20	0.25	0.10
PCA	0.34	0.34	0.74	0.36	0.69	0.59	0.49	0.60	0.88	0.60	0.30	0.59	0.37	0.33	0.16
RSA	0.36	0.34	0.71	0.37	0.69	0.57	0.36	0.54	0.88	1.09	0.60	0.67	0.54	0.52	0.44
	RMSE														
BMS	0.16	0.26	0.22	0.28	0.29	0.33	0.24	0.15	0.29	0.26	0.17	0.22	0.17	0.22	0.22
HMS	0.15	0.22	0.22	0.24	0.29	0.31	0.23	0.15	0.28	0.21	0.12	0.19	0.11	0.18	0.08
OOM	0.15	0.21	0.22	0.23	0.29	0.30	0.22	0.14	0.28	0.16	0.22	0.19	0.21	0.28	0.23
AM	0.15	0.22	0.22	0.24	0.29	0.32	0.24	0.14	0.27	0.14	0.11	0.16	0.10	0.15	0.08
PCA	0.15	0.22	0.22	0.25	0.29	0.31	0.25	0.16	0.27	0.21	0.13	0.20	0.14	0.19	0.13
RSA	0.17	0.26	0.22	0.29	0.30	0.31	0.23	0.14	0.29	0.34	0.22	0.23	0.21	0.31	0.32
	r-value														
BMS	0.34	0.30	0.30	0.45	0.50	0.49	0.47	0.45	0.41	0.84	0.81	0.50	0.90	0.88	0.79
HMS	0.35	0.33	0.39	0.38	0.21	0.28	0.62	0.21	0.23	0.88	0.90	0.83	0.94	0.88	0.98
OOM	0.66	0.62	0.57	0.27	0.65	0.48	0.64	0.48	0.33	0.94	0.74	0.78	0.51	0.67	0.77
AM	0.32	0.35	0.37	0.26	0.34	0.30	0.34	0.58	0.27	0.94	0.98	0.90	0.93	0.92	0.99
PCA	0.40	0.32	0.37	0.39	0.41	0.40	0.29	0.36	0.37	0.87	0.94	0.79	0.86	0.90	0.99
RSA	0.29	0.77	0.49	0.45	0.37	0.48	0.55	0.31	0.43	0.11	0.36	0.48	0.70	0.77	0.43

TABLE IX
PERFORMANCE MATRIX FOR ELM WITH POLYNOMIAL KERNEL

	Response Time	Availability	Throughput	Successability	Reliability	Compliance	Best Practices	Latency	Documentation	Maintainability	Modularity	Reusability	Testability	Interoperability	Conformity
BMS	0.10	0.14	0.18	0.17	0.24	0.26	0.19	0.11	0.23	0.17	0.11	0.16	0.11	0.15	0.12
HMS	0.11	0.15	0.19	0.17	0.26	0.28	0.19	0.11	0.23	0.12	0.05	0.11	0.04	0.10	0.03
OOM	0.12	0.15	0.21	0.18	0.27	0.28	0.19	0.11	0.23	0.12	0.13	0.14	0.15	0.18	0.12
AM	0.13	0.20	0.20	0.23	0.28	0.33	0.23	0.13	0.28	0.10	0.06	0.12	0.05	0.11	0.03
PCA	0.11	0.15	0.20	0.18	0.24	0.26	0.19	0.12	0.23	0.10	0.06	0.13	0.07	0.11	0.03
RSA	0.10	0.14	0.18	0.16	0.24	0.26	0.18	0.11	0.22	0.25	0.14	0.16	0.14	0.23	0.23
	MMRE														
BMS	0.36	0.32	0.75	0.35	0.68	0.62	0.39	0.58	0.94	0.71	0.41	0.67	0.44	0.36	0.20
HMS	0.38	0.33	0.76	0.35	0.71	0.64	0.40	0.57	0.93	0.52	0.19	0.40	0.18	0.19	0.05
OOM	0.42	0.34	0.81	0.37	0.77	0.64	0.37	0.56	0.94	0.44	0.48	0.49	0.57	0.41	0.22
AM	0.48	0.41	0.82	0.46	0.73	0.70	0.54	0.67	1.18	0.42	0.25	0.41	0.18	0.21	0.05
PCA	0.36	0.35	0.80	0.37	0.69	0.59	0.47	0.60	0.93	0.38	0.24	0.50	0.28	0.23	0.06
RSA	0.36	0.31	0.74	0.33	0.68	0.59	0.36	0.57	0.91	1.04	0.52	0.67	0.53	0.51	0.41
	RMSE														
BMS	0.15	0.20	0.22	0.23	0.28	0.31	0.22	0.15	0.29	0.23	0.16	0.22	0.17	0.20	0.16
HMS	0.16	0.21	0.23	0.23	0.30	0.33	0.23	0.16	0.29	0.18	0.08	0.17	0.07	0.13	0.04
OOM	0.18	0.22	0.26	0.25	0.32	0.34	0.23	0.15	0.30	0.15	0.20	0.19	0.21	0.25	0.18
AM	0.19	0.27	0.25	0.32	0.33	0.41	0.28	0.17	0.36	0.16	0.10	0.16	0.07	0.17	0.05
PCA	0.15	0.22	0.24	0.24	0.28	0.31	0.24	0.16	0.29	0.13	0.11	0.18	0.11	0.16	0.06
RSA	0.16	0.20	0.22	0.22	0.28	0.31	0.22	0.15	0.28	0.31	0.20	0.21	0.20	0.28	0.26
	r-value														
BMS	0.46	0.30	0.39	0.16	0.36	0.16	0.38	0.07	0.09	0.86	0.88	0.45	0.93	0.89	0.89
HMS	0.18	0.31	0.19	0.38	-0.02	0.19	0.24	0.46	0.37	0.94	0.97	0.90	1.00	0.95	1.00
OOM	0.08	0.29	0.38	0.22	0.36	0.16	0.34	0.42	0.46	0.96	0.77	0.72	0.59	0.75	0.92
AM	0.12	0.31	0.40	-0.01	0.40	0.19	0.42	0.50	0.37	0.95	0.99	0.91	0.98	0.94	1.00
PCA	0.18	-0.04	0.46	0.19	0.29	0.39	0.58	0.51	0.51	0.94	0.98	0.90	0.99	0.95	1.00
RSA	0.25	0.37	0.54	0.26	0.39	0.24	0.42	0.13	0.26	0.60	0.75	0.45	0.51	0.70	0.62

square deviation computes the sample standard deviation of the differences between predicted values by the estimator and actual values. RMSE is also an indicator of the predicted and observed values. From Table VIII, we infer that the best RMSE value in-case of ELM with linear kernel is for response time and latency QoS parameters. The minimum RMSE value obtained is 0.08 for HMS metrics and conformity parameter in-case of linear kernel. From Table VIII, we observe that in-case of polynomial kernel, the performance of PCA based feature extraction technique is better for some parameters in comparison to RSA based feature selection technique and similarly the performance of RSA is better than PCA for some parameters. We do not observe a dominate approach between PCA and RSA.

IX. COMPARING ALGORITHMS USING STATISTICAL SIGNIFICANCE TESTING

Our objective is to compare several learning algorithms and assess which algorithm is better. Dietterich et al. review 5 approximate statistical tests for determining whether one learning algorithm outperforms another on a particular learning task and dataset [18]. We apply the 10-fold cross-validated paired t-test as described in the paper by Dietterich et al. [18]. We have several combination of subsets of metrics and ELM kernel functions as learning algorithms. We consider 6 different sets

TABLE X
PERFORMANCE MATRIX FOR ELM WITH RBF KERNEL

	Response Time	Availability	Throughput	Successability	Reliability	Compliance	Best Practices	Latency	Documentation	Maintainability	Modularity	Reusability	Testability	Interoperability	Conformity
BMS	0.10	0.14	0.18	0.17	0.24	0.26	0.18	0.11	0.22	0.25	0.16	0.16	0.14	0.24	0.25
HMS	0.10	0.14	0.18	0.17	0.24	0.26	0.18	0.11	0.22	0.25	0.16	0.18	0.14	0.25	0.29
OOM	0.09	0.13	0.18	0.16	0.24	0.25	0.18	0.11	0.22	0.24	0.17	0.16	0.15	0.25	0.27
AM	0.09	0.13	0.19	0.17	0.24	0.26	0.20	0.11	0.21	0.24	0.17	0.17	0.14	0.24	0.31
PCA	0.09	0.13	0.19	0.16	0.24	0.25	0.20	0.11	0.21	0.24	0.17	0.17	0.14	0.24	0.30
RSA	0.10	0.13	0.18	0.16	0.24	0.25	0.18	0.11	0.22	0.26	0.16	0.16	0.14	0.26	0.26
	MMRE														
BMS	0.34	0.31	0.74	0.35	0.67	0.60	0.39	0.56	0.91	1.03	0.60	0.71	0.55	0.52	0.47
HMS	0.34	0.32	0.74	0.35	0.66	0.61	0.38	0.56	0.92	1.05	0.58	0.76	0.53	0.53	0.65
OOM	0.33	0.31	0.73	0.34	0.70	0.58	0.38	0.56	0.89	1.03	0.61	0.65	0.53	0.56	0.49
AM	0.32	0.32	0.74	0.35	0.68	0.60	0.48	0.56	0.90	0.97	0.59	0.72	0.50	0.54	0.69
PCA	0.31	0.30	0.75	0.34	0.68	0.59	0.49	0.58	0.88	0.96	0.60	0.73	0.49	0.54	0.68
RSA	0.34	0.30	0.73	0.33	0.68	0.57	0.37	0.55	0.90	1.12	0.61	0.70	0.55	0.55	0.47
	RMSE														
BMS	0.15	0.20	0.21	0.22	0.27	0.29	0.22	0.15	0.28	0.31	0.21	0.22	0.20	0.29	0.27
HMS	0.15	0.20	0.22	0.22	0.27	0.30	0.22	0.15	0.28	0.31	0.21	0.24	0.20	0.29	0.31
OOM	0.14	0.19	0.22	0.21	0.28	0.29	0.22	0.14	0.27	0.30	0.23	0.21	0.21	0.30	0.28
AM	0.14	0.20	0.22	0.22	0.28	0.29	0.23	0.14	0.27	0.30	0.23	0.23	0.20	0.29	0.32
PCA	0.14	0.19	0.22	0.21	0.28	0.29	0.24	0.16	0.27	0.30	0.23	0.23	0.20	0.29	0.32
RSA	0.15	0.19	0.21	0.21	0.27	0.29	0.21	0.14	0.27	0.31	0.22	0.22	0.20	0.30	0.28
	r-value														
BMS	0.31	0.50	0.19	0.02	0.08	0.24	0.50	0.46	0.15	0.87	0.81	0.47	0.81	0.78	0.93
HMS	0.34	0.18	0.43	0.38	0.29	0.33	0.33	0.38	0.34	0.93	0.96	0.48	0.91	0.78	0.97
OOM	0.57	0.34	0.35	0.44	0.55	0.37	0.64	0.29	0.26	0.82	0.58	0.71	0.53	0.70	0.83
AM	0.63	0.39	0.29	0.20	0.48	0.28	0.32	0.30	0.34	0.96	0.94	0.62	0.83	0.91	0.95
PCA	0.50	0.20	0.32	0.00	0.44	0.41	0.64	0.26	0.24	0.92	0.73	0.68	0.75	0.85	0.98
RSA	0.21	0.39	0.36	0.42	0.38	0.46	0.61	0.35	0.43	0.42	0.59	0.20	0.44	0.56	0.57

of metrics: All Metrics (AM), Only Object Oriented Metrics (OOM), Harry M. Sneed's Metrics (HMS), Baski and Misra Metrics (BMS), Metrics derived after executing PCA, and metrics derived after executing RSA. We consider 6 sets of metrics as input to develop a model to predict 15 different QoS parameters. We investigate the application of extreme learning machine with three different types of kernel functions: linear kernel, polynomial kernel, and radial basis function with three different performance parameters. Hence, for each subset of metrics, a total number of three set (one for each performance measure) are used, each with 45 data points (3 kernels multiplied by 15 QoS parameters). Table XI displays the result of the 10-fold cross-validated paired t-test analysis. For each of the 3 kernels (Linear, Polynomial and RBF), 6 different subset of metrics are considered as input with three different performance parameters. The three different performance parameters are: Mean Absolute Error (MAE), Mean Magnitude of Relative Error (MMRE) and Root Mean

TABLE XI
EXPERIMENTAL RESULTS ON T-TEST BETWEEN DIFFERENT SET OF METRICS

MAE												
	Mean Difference						**p-value**					
	BMS	HMS	OOM	AM	PCA	RSA	BMS	HMS	OOM	AM	PCA	RSA
BMS	0.000	0.014	0.000	0.009	0.011	-0.013	NaN	0.004	0.986	0.140	0.008	0.008
HMS	-0.014	0.000	-0.014	-0.004	-0.002	-0.026	0.004	NaN	0.017	0.164	0.175	0.003
OOM	0.000	0.014	0.000	0.009	0.011	-0.013	0.986	0.017	NaN	0.155	0.025	0.026
AM	-0.009	0.004	-0.009	0.000	0.002	-0.022	0.140	0.164	0.155	NaN	0.570	0.035
PCA	-0.011	0.002	-0.011	-0.002	0.000	-0.024	0.008	0.175	0.025	0.570	NaN	0.005
RSA	0.013	0.026	0.013	0.022	0.024	0.000	0.008	0.003	0.026	0.035	0.005	NaN
MMRE												
	Mean Difference						**p-value**					
	BMS	HMS	OOM	AM	PCA	RSA	BMS	HMS	OOM	AM	PCA	RSA
BMS	0.000	0.037	0.000	0.026	0.027	-0.037	NaN	0.011	0.990	0.182	0.051	0.009
HMS	-0.037	0.000	-0.037	-0.010	-0.010	-0.074	0.011	NaN	0.039	0.346	0.199	0.005
OOM	0.000	0.037	0.000	0.026	0.027	-0.037	0.990	0.039	NaN	0.199	0.081	0.078
AM	-0.026	0.010	-0.026	0.000	0.000	-0.063	0.182	0.346	0.199	NaN	0.967	0.048
PCA	-0.027	0.010	-0.027	0.000	0.000	-0.064	0.051	0.199	0.081	0.967	NaN	0.014
RSA	0.037	0.074	0.037	0.063	0.064	0.000	0.009	0.005	0.078	0.048	0.014	NaN
RMSE												
	Mean Difference						**p-value**					
	BMS	HMS	OOM	AM	PCA	RSA	BMS	HMS	OOM	AM	PCA	RSA
BMS	0.000	0.018	-0.001	0.010	0.013	-0.014	NaN	0.002	0.781	0.205	0.008	0.005
HMS	-0.018	0.000	-0.020	-0.008	-0.005	-0.033	0.002	NaN	0.010	0.066	0.038	0.002
OOM	0.001	0.020	0.000	0.012	0.014	-0.013	0.781	0.010	NaN	0.167	0.021	0.050
AM	-0.010	0.008	-0.012	0.000	0.003	-0.025	0.205	0.066	0.167	NaN	0.575	0.045
PCA	-0.013	0.005	-0.014	-0.003	0.000	-0.027	0.008	0.038	0.021	0.575	NaN	0.004
RSA	0.014	0.033	0.013	0.025	0.027	0.000	0.005	0.002	0.050	0.045	0.004	NaN

TABLE XII
EXPERIMENTAL RESULTS ON T-TEST BETWEEN THREE DIFFERENT KERNELS

Mean Difference											
MAE				**MRE**				**RMSE**			
	Linear	**Polynomial**	**RBF**		**Linear**	**Polynomial**	**RBF**		**Linear**	**Polynomial**	**RBF**
Linear	0.000	0.007	-0.021	**Linear**	0.000	0.005	-0.084	**Linear**	0.000	0.005	-0.018
Polynomial	-0.007	0.000	-0.028	**Polynomial**	-0.005	0.000	-0.090	**Polynomial**	-0.005	0.000	-0.023
RBF	0.021	0.028	0.000	**RBF**	0.084	0.090	0.000	**RBF**	0.018	0.023	0.000
p-value											
MAE				**MRE**				**RMSE**			
	Linear	**Polynomial**	**RBF**		**Linear**	**Polynomial**	**RBF**		**Linear**	**Polynomial**	**RBF**
Linear	NaN	0.13	0.001	**Linear**	NaN	0.467	0.000	**Linear**	NaN	0.121	0.007
Polynomial	0.13	NaN	0.000	**Polynomial**	0.467	NaN	0.000	**Polynomial**	0.121	NaN	0.005
RBF	0.001	0.000	NaN	**RBF**	0.000	0.000	NaN	**RBF**	0.007	0.005	NaN

Squared Error (RMSE). Hence for each kernel a total three set (one for each performance measure) are used, each with 90 data points (six subsets of metrics multiplied by 15 QoS). The experimental results of t-test analysis for different performance parameter (MAE, MMRE and RMSE) and three different ELM kernels are summarized in Table XII. Table XII contains two parts. The first part of the table XII shows the mean difference value and second part shows the p-value between different pairs. Table XII reveals that there is no significant difference between the kernel function, due to the fact that p-value is greater than 0.05. However, by closely examining the value of mean difference, polynomial kernel yields better result compared to other kernels function i.e., linear and RBF kernel functions.

X. CONCLUSION

We develop a predictive model to estimate QoS parameters of web services using source code (implementing the services) metrics. We experiment with six different sets of metrics as input to develop a prediction model. The performance of these sets of metrics are evaluated using Extreme Learning Machines (ELM) with various kernel functions such as linear, polynomial and RBF kernel function. From the correlation analysis between metrics, we observe that there exists a high correlation between Object-Oriented metrics and WSDL metrics. From t-test analysis, we infer that in most of the cases, there is the difference between the various sets of metrics in terms of the performance of the estimator is not substantial but moderate. We observe that the predictive model developed using Harry M. Sneed (HMS) metrics yields better result compared to other sets of metrics such as all metrics and Baski and Misra metrics. From t-test analysis, we can also interpret that difference between the three kernel functions in-terms of their influence on the predictive accuracy is moderate. We conclude that none of the feature selection technique dominate the other and one feature selection method is better than the other for some QoS parameters and vice-versa. By assessing the value of mean difference, we infer that the polynomial kernel for ELM yields better result compared to other kernels function i.e., linear and RBF kernel functions. From performance results, it is observed that the performance of the predictive model or estimator varies with the different sets of software metrics, feature selection technique and the kernel functions. Finally, we conclude that it is possible to estimate the QoS parameters using ELM and source code metrics.

REFERENCES

[1] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the web services web: an introduction to soap, wsdl, and uddi," *IEEE Internet computing*, vol. 6, no. 2, p. 86, 2002.

[2] E. Newcomer and G. Lomow, *Understanding SOA with Web services*. Addison-Wesley, 2005.

[3] J. L. O. Coscia, M. Crasso, C. Mateos, A. Zunino, and S. Misra, *Predicting Web Service Maintainability via Object-Oriented Metrics: A Statistics-Based Approach*. Springer Berlin Heidelberg, 2012, pp. 29–39.

[4] J. L. O. Coscia, M. Crasso, C. Mateos, and A. Zunino, "Estimating web service interface quality through conventional object-oriented metrics." *CLEI Electron. J.*, vol. 16, 2013.

[5] C. Mateos, M. Crasso, A. Zunino, and J. L. O. Coscia, "Detecting wsdl bad practices in code-first web services," *Int. J. Web Grid Serv.*, vol. 7, no. 4, pp. 357–387, Jan. 2011.

[6] L. Kumar, M. Kumar, and S. K. Rath, "Maintainability prediction of web service using support vector machine with various kernel methods," *International Journal of System Assurance Engineering and Management*, pp. 1–18, 2016.

[7] S. O. Olatunji, Z. Rasheed, K. Sattar, A. Al-Mana, M. Alshayeb, and E. El-Sebakhy, "Extreme learning machine as maintainability prediction model for object-oriented software systems," *Journal of Computing*, vol. 2, no. 8, pp. 49–56, 2010.

[8] E. Al-Masri and Q. H. Mahmoud, "Qos-based discovery and ranking of web services," in *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on*, 2007, pp. 529–534.

[9] E. Al Masri and Q. H. Mahmoud, "Investigating web services on the world wide web," *Proceedings of the 17th International Conference on World Wide Web*, pp. 795–804, 2008.

[10] D. Baski and S. Misra, "Metrics suite for maintainability of extensible markup language web services," *IET Software*, vol. 5, no. 3, pp. 320–341, 2011.

[11] H. M. Sneed, "Measuring web service interfaces," in *Web Systems Evolution (WSE), 2010 12th IEEE International Symposium on*, 2010, pp. 111–115.

[12] M. Jureczko and D. Spinellis, *Using Object-Oriented Design Metrics to Predict Software Defects*, ser. Monographs of System Dependability, 2010, vol. Models and Methodology of System Dependability, pp. 69–81.

[13] D. Spinellis, "Tool writing: a forgotten art? (software tools)," *IEEE Software*, vol. 22, no. 4, pp. 9–11, 2005.

[14] R. W. Swiniarski and A. Skowron, "Rough set methods in feature selection and recognition," *Pattern recognition letters*, vol. 24, no. 6, pp. 833–849, 2003.

[15] G.-B. Huang, D. H. Wang, and Y. Lan, "Extreme learning machines: a survey," *International Journal of Machine Learning and Cybernetics*, vol. 2, no. 2, pp. 107–122, 2011.

[16] S. Ding, Y. Zhang, X. Xu, and L. Bao, "A novel extreme learning machine based on hybrid kernel function," *Journal of Computers*, vol. 8, no. 8, pp. 2110–2117, 2013.

[17] B. A. Kitchenham, L. M. Pickard, S. G. MacDonell, and M. J. Shepperd, "What accuracy statistics really measure [software estimation]," *IEE Proceedings-Software*, vol. 148, no. 3, pp. 81–85, 2001.

[18] T. G. Dietterich, "Approximate statistical tests for comparing supervised classification learning algorithms," *Neural Comput.*, vol. 10, no. 7, pp. 1895–1923, Oct. 1998.

Local Variables with Compound Names and Comments as Signs of Fault-Prone Java Methods

Hirohisa Aman
Center for Information Technology
Ehime University
Matsuyama, Ehime 790-8577, Japan
Email: aman@ehime-u.ac.jp

Sousuke Amasaki
Tomoyuki Yokogawa
Faculty of Computer Science
and Systems Engineering
Okayama Prefectural University
Soja, Okayama 719–1197, Japan

Minoru Kawahara
Center for Information Technology
Ehime University
Matsuyama, Ehime 790-8577, Japan

***Abstract*—This paper focuses on two types of artifacts—local variables and comments in a method (function). Both of them are usually used at the programer's discretion. Thus, naming local variables and commenting code can vary among individuals, and such an individual difference may cause a dispersion in quality. This paper conducts an empirical analysis on the fault-proneness of Java methods which are collected from nine popular open source products. The results report the following three findings: (1) Methods having local variables with compound names are more likely to be faulty than the others; (2) Methods having local variables with simple and short names are unlikely to be faulty, but their positive effects tend to be decayed as their scopes get wider; (3) The presence of comments within a method body can also be useful sign of fault-prone method.**

I. Introduction

Software systems have been utilized in many aspects of our daily life, and management of software quality has been the most significant activity for ensuring the safety and security of the people. In fact, it is hard to always make a one-shot release of a perfect software product which has no need to be enhanced or modified in the future; software systems usually require upgrades after their releases in order to fix their faults and/or to enrich their functionality. Needless to say, it is better to reduce both the frequency of their upgrades and the size of their patches to be applied.

To minimize upgrades of software products, thorough review and testing before their releases are desirable activities. In general, software review and testing help to detect concealed faults or identify suspicious software modules which are fault-prone [1], [2]. Then, those problems can be resolved by fixing faults or refactoring problematic programs in order to reduce the risk of causing unwanted upgrades after their releases. While review and testing are useful activities, they are also costly ones, so there have been many studies using software metrics to predict fault-prone modules prior to software review and testing activities [3]. By predicting fault-prone parts of a software product, cost-effective review and testing would be performed, i.e., we would detect more faults at less cost.

Most studied methods and models for predicting fault-prone modules have been based on structural features of products such as their sizes and complexities, or on development histories stored in their code repositories such as the number of bug-fix commitments which have been made by a certain point in time [4], [5], [6]. However, the impact of human factors would also be significant since programming activities are usually done by human beings. Different programmers would probably develop different programs for the same specification. Such a difference among individuals must have a certain level of influence on the quality of products, i.e., it must cause a dispersion in quality. Therefore, we focus on the following two artifacts which may vary from person to person, (1) local variables declared in a method (function) and (2) comments written inside the method body. While these artifacts have no impact on the structure of a program, they seem to be related to the understandability and the readability of the program, so they can be expected to play important roles in predicting fault-prone methods. In this paper, we quantitatively analyze the relationships of these artifacts with the fault-proneness.

The key contribution of this paper is to provide the following findings derived from the results of our empirical analysis with nine popular open source software (OSS) products:

- Local variables with descriptive compound names (for example, "`countOfSatisfactoryRecords`") can be signs that the methods are fault-prone.
- Methods having local variables with simple and short names (for example, "`c`" or "`cnt`") are unlikely to be faulty, but their positive effects tend to be decayed as their scopes get wider.
- Comments within a method body also seem to be related to the fault-proneness of the method.

The remainder of this paper is organized as follows. Section II describes two types of artifacts which may vary among programmers—(1) names of local variables and (2) comments written inside a method body—and their relationships with the quality of source programs, and gives our research questions in regard to impacts of those artifacts. Section III reports on an empirical analysis on our research questions using popular OSS products, and discusses the results. Section IV briefly describes related work. Finally, Section V presents the conclusion of this paper and our future work.

II. Local Variables and Comments

This paper focuses on local variables and comments, since they may vary widely from person to person and cause a variation in quality. This section describes concerns of local variable names and comments in regard to source code quality, and set up our research questions.

A. Local Variable Name

Since local variables are valid only within a function or a method, names of local variables are usually not specified in their software specifications or design documents. Therefore, naming local variables can be at the programmer's discretion. In general, different programmers would prefer different names for local variables even if they implement the same algorithm in their function or method. For example, a programmer likes to use "`count`" as the name of local variable for storing the number of records which satisfy a certain condition, but another programmer prefers "`c`" as its name; there might even be a programmer who wants to give "`countOfSatisfactoryRecords`" to the variable.

Needless to say, local variables with fully-spelled names such as "`count`" or ones with descriptive compound names "`countOfSatisfactoryRecords`" make it easy to understand the roles of those variables in their function or method since those names provide more information about those variables than shorter and/or simpler names. Lawrie et al. [7] surveyed the understandability of identifiers (including not only local variables' names but also functions' names) used in programs by comparing three types of names, (1) fully-spelled names such as "`count`," (2) abbreviated names such as "`cnt`" and (3) names using only an initial letter such as "`c`." They reported that a longer name is easier to understand for programmers, but there is not a significant difference in comprehensibility between fully-spelled names and abbreviated ones in their survey results. That is to say, it is not always necessary to give a long and descriptive name to a local variable, and a short and simple name may be sufficient.

There are also programming heuristics on naming local variables. Both the GNU coding standards [8] and the Java coding convention [9] have said that names of local variables should be shorter. Moreover, Kernighan and Pike [10] also argued that shorter names are sufficient for local variables; for example, they considered that name "`n`" looks good for a local variable storing "the number of points" while name "`numberOfPoints`" seems to be overdone. Thus, long and descriptive names have not been recommended for the names of local variables. However, the impact of such a descriptive name on the code quality has not been clearly discussed in those heuristics.

Aman et al. [11] conducted an empirical work and showed that methods having local variables with long names are more likely to be fault-prone and change-prone than the other methods. That is to say, they showed a relationship between a long name of a local variable and a poor quality of the code in a statistical manner. However, their analysis missed taking into account the following two aspects: (1) the composition of variable's name and (2) the scope of local variable. Focusing on not only the length of local variable's name but also those two aspects would be more worthy in analyzing the impact of local variable's name and in enhancing the quality of code. This is a key motivation of this work.

B. Comments

Comments are documents embedded in a source file, which usually provide beneficial information in regard to the program [12]. While there are several types of comments, we focus on comments written inside a method (function) body in this paper. Those comments usually give explanations or programmer's memos for their implementation in the method. Of course, the other types of comments also provide important information regarding the program. However, such comments written outside a method body are often the copyright designation or the programmer's manual explaining how to use the method, i.e., those comments may not be decided at the discretion of the programmer. Thus, those comments outside a method body may be out of our research scope focusing on the individual difference among programmers. That is the reason why we will focus only on the comments written inside a method body.

While comments along with executable code can be a great help in understanding the code, there have also been criticisms on their effects: comments might be written to compensate for a lack of readability in complicated programs [13]. In this context, Fowler [14] pointed out that well-written comments may be "deodorant" for masking "code smells." Although comments themselves are good artifacts, they may be used for neutralizing a "bad-smelling" code. Kernighan and Pike [10] said that programmers should not add detailed comments to a bad code; in such a case, it is better to rewrite their code rather than adding comments. If a programmer wants to add detailed comments to their code during their programming activity, the programmer may consider that the program is hard to understand for others without those comments. That is to say, comments may be signs of complicated programs. Aman et al. [11], [15] reported supporting empirical results that commented programs tend to be more fault-prone than non-commented ones. In this paper, we conduct a further analysis examining combinations of (1) the composition of local variable's name, (2) local variable's scope and (3) comments, in terms of fault-proneness.

C. Research Questions

As mentioned above, both the local variables and the comments are not only artifacts which may vary among programmers, but also remarkable ones which are expected to have relationships with the quality of the code. However, the analyses in the previous work [11], [15] missed considerations for the composition of local variable's name and the scope of local variable. We will conduct a further analysis by focusing on those missed aspects as well. In order to clarify our points of view in our empirical analysis, we set up the following two research questions (RQs):

TABLE I
SURVEYED OSS PRODUCTS.

Product	Size (KLOC)	#Methods Having a Local Variable	Data Collection Period	Domain
IP-Scanner	16	433	2006-07-19 — 2016-04-04	Networking
Checkstyle	21	738	2003-05-05 — 2016-03-28	Code analysis
eXo	21	675	2007-03-17 — 2016-04-06	Social collaboration software
FreeMind	71	2,353	2011-02-06 — 2016-03-30	Mind-mapping tool
ARM	282	1,300	2013-09-11 — 2016-03-14	Development support
Hibernate	387	6,372	2007-06-29 — 2016-03-31	Object/Relational mapping
ProjectLibre	224	1,466	2012-08-22 — 2016-04-06	MS Project clone
PMD	75	738	2002-06-21 — 2016-04-05	Source code analyzer
SQuirreL	405	6,060	2001-06-01 — 2016-04-05	Database client
Total	1,502	20,135		

RQ1 Can local variables with compound names be signs of fault-prone methods?

RQ2 How does a local variable's scope relate to the effect of local variable's name on the fault-proneness in a method?

We will check the above two questions while considering the impact of comments as well. □

As mentioned in Section II-A, there have been concerns in giving descriptive names to local variables. Compound names such as "`numberOfPoints`" are typical descriptive names. RQ1 asks whether a local variable with such a compound name can be a sign to find fault-prone method or not.

If a local variable is declared with a narrow scope, it does not seem to need a descriptive name since its influence is limited within a narrow range. RQ2 focuses on the relationship of local variables' names with their scopes.

In examining these RQs, this paper expects to find yet another useful clue of fault-prone methods by focusing on their local variable names.

III. EMPIRICAL ANALYSIS

This section conducts an empirical analysis in which we collect quantitative data from popular OSS products and analyzes that data in order to discuss the above research questions.

A. Aim and Dataset

The aim of this analysis is to quantitatively examine the fault-proneness of Java methods by focusing on the names of local variables, the scopes of them and the presence of comments. The results of this analysis are expected to present useful points to be checked during code review activities.

We collected data from nine popular OSS products of different size and domain, shown in Table I—(1) Angry IP Scanner (IP-Scanner)[1], (2) Eclipse Checkstyle Plug-in (Checkstyle)[2], (3) eXo Platform (eXo)[3], (4) FreeMind[4], (5) GNU ARM Eclipse Plug-ins (ARM)[5], (6) Hibernate ORM (Hibernate)[6], (7) PMD[7], (8) ProjectLibre[8] and (9) SQuirreL SQL Client (SQuirreL)[9]. All of them are ranked in the top 50 popular Java products at SourceForge.net[10], and their source files have been maintained with the Git. The restrictions of the development language and the version control system are from our data collection tools[11].

B. Procedure of Data Collection

We collected data from each OSS project in the following procedure.

(1) Make a clone of the repository, and make the list of all methods included in the current version.

(2) Get the change history of each method:
We check the source lines which had been changed through each commitment on the repository, and decide which methods were modified at that time (see Fig.1). The decision is made by the following three steps.

(2a) Get both the older version and the newer version of the source file which had upgraded through the commitment.

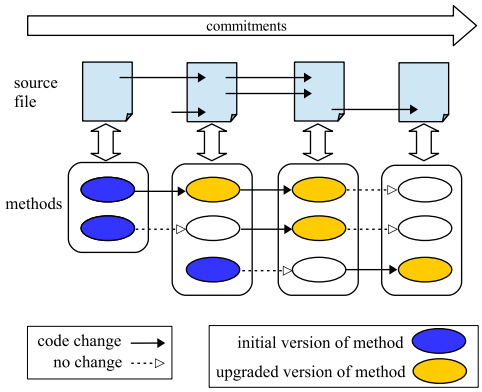


Fig. 1. Change histories of methods included in a source file.

[1]http://angryip.org/
[2]http://eclipse-cs.sourceforge.net/
[3]http://exoplatform.com/
[4]http://freemind.sourceforge.net/wiki/index.php/Main_Page
[5]http://gnuarmeclipse.livius.net/blog/
[6]http://hibernate.org/
[7]https://pmd.github.io/
[8]http://www.projectlibre.org/
[9]http://www.squirrelsql.org/
[10]http://sourceforge.net/
[11]http://se.cite.ehime-u.ac.jp/tool/

(2b) Compare those two consecutive versions, and find different parts between them. Then, obtain corresponding line numbers in the newer version.

(2c) Decide which method(s) had been upgraded, by checking the line numbers of upgraded lines against each method's position (range) in the newer version.

By iterating these steps for all commitments, we get the change history of each method.

(3) Collect the data on representative local variables' names and scopes, and comments for each method:
We survey names of local variables declared in the initial version of a method (see Fig.1) and the scopes of those variables. We define the length of a local variable's scope to be the number of lines where the variable is valid except for the line of its declaration. For example, the length of scope of variable "`len`" shown in Fig.2 is 5 and that of variable "`str`" is 2, respectively.
When there are two or more variables in a method, we focus on the variable whose scope is widest in the method as the "representative local variable" in order to connect the features of the local variable to the method. In the example shown in Fig.2, the "representative local variable" of method "`foo`" is variable "`len`." If there are two or more local variables with the widest scope in a method, we will adopt the variable with the longer name (having more characters) as the representative variable. Needless to say, if there is only one local variable in a method, the variable is the representative local variable of the method. On the other hand, any methods having no local variable are excluded from the data of interest in this work.
We collect the lines of comments written inside a method body as well.

(4) Check if a bug fix has occurred for each method:
We examine the change history of each method obtained above and check if a bug fix has occurred or not at the method's upgrade. We decide whether a code change was intended to a bug fixing or not, by checking their commitment message [16]. For example, Fig. 3 shows a part of commitment message (obtained by using `git log` command) on the repository of SquirreL SQL Client, which seems to be a bug fixing commitment. Since method "`_init`" in "`AliasEditController.java`" was modified through the commitment, we consider that a bug fixing was performed at the method.

C. Procedure of Data Analysis

We conducted our data analysis in the following procedure.

```
String foo (String arg) {
   int len = arg.length();
   if (len < 5) {
     return new String(arg);
   }
   String str = arg.substring(0, 5);
   return str + "...";
}
```

Fig. 2. An example of method having local variables.

```
commit 0d005dc6573dcc12df03917ee974a0736b4d5cfd
............
Bug #1236 Shortcut for comment/uncomment current line
(ctrl + "/") does not
Fixed according to the suggestion in the bug #1236
Please note: The orginal comment/uncomment hot key of
SQuirreL is ctrl+Num
```

Fig. 3. An example of actual commitment message.

(1) Perform a random sampling of methods, which have a local variable, from all projects:
In order to avoid an impact of project's size bias on our empirical results, we randomly sample the same number of methods from each project.

(2) Divide the set of methods into subsets according to the representative local variable's name.
We consider "a local variable with a short name" to be one such that the length of its name is less than or equal to the 25 percentile in the distribution of length of name. We also take into account if the name is compound one or not for RQ1. Thus, we consider the following three categories.

- V_1: the set of methods such that the name of representative local variable is *short* and *not compound*.
- V_2: the set of methods such that the name of representative local variable is *not short* and *not compound*.
- V_3: the set of methods such that the name of representative local variable is a *compound* one.

We decide that a variable has a compound name if it is composed in camel case such as "`numberOfItems`." That is to say, we consider a name to be compound one if it has a lower case letter followed by an upper case letter. We regard such a pair of lower case letter and upper case letter as a splitting position of the name. For example, there are two splitting positions in "`numberOfItems`," i.e., the pair of "`r`" and "`O`," and the pair of "`f`" and "`I`," so the name can be split into three portions (words) "`number`," "`Of`" and "`Items`." We consider that such compounded names cannot be short ones composed by at most a few characters. Thus, we do not divide the set of methods having representative local variables with compound names, and define V_3 only (not V_3 and V_4).

(3) Divide the subsets of methods obtained at Step (2) into two, according to the presence of comments:
In order to analyze the impact of comments as well, we divide the set of methods into two subsets by checking if there are comments[12] inside method bodies or not.

- C_0: the set of methods having *no comment*.
- C_1: the set of methods having *comments*.

Then, we define $M_{ij} = C_i \cap V_j$ for $i = 0, 1$ and $j = 1, 2, 3$. For example, M_{01} is the set of non-commented methods in which the representative local variable has a short and non-compound name. Table II summarizes these categories (the method sets) M_{ij}.

[12]We excluded the comment out cases from our data by using a checking algorithm [17].

TABLE II
SYMBOLS REPRESENTING CATEGORIES.

Symbol	Name of representative local variable		
	Non-compound		Compound
	short	not short	
Non-commented methods	M_{01}	M_{02}	M_{03}
Commented methods	M_{11}	M_{12}	M_{13}

(4) Examine the fault-proneness of methods by the above categories M_{ij}:
We statistically compare the bug fix rates among categories M_{ij} (for $i = 0, 1$ and $j = 1, 2, 3$) and discuss the results.
(5) Examine the trends of the bug fix rates over scope:
In order to analyze the impact of variable's scope as well, we analyze the changes in bug fix rate by varying the range (the length of scope) which we focus on. In the concrete, we compare the moving averages of the bug fix rates among categories M_{ij} (for $i = 0, 1$ and $j = 1, 2, 3$), by varying the range of focusing scope.

D. Results and Discussion: Collected Data

We first show the results of our data collection. Since the minimum number of methods included in a project was 433 as shown in Table I (project "IP-Scanner"), we randomly sampled 400 methods from each project, so our dataset consists of $3,600$ methods in total.

Table III shows the distributions of length of representative local variables' names in character count and in word count, respectively. Here, "word count" means the number of words composing a variable's name which is split according to the notion of the camel case. The longest names in character count were "`containsSuppressWarningsHolderModule`" and "`organizationInitializersHomePathNode`" which consist of 36 characters, and the longest name in word count was "`thereWereNodesToBeFolded`" which consists of 6 words. Although such some long and descriptive names appear in some methods, most local variables have names that consist of at most a few characters and they are non-compound names whose word count is one. Since the 25 percentile (Q_1) of the character count is four as shown in Table III, we will consider a name whose length is less than or equal to four letters to be short in the following analysis.

Table IV presents the distribution of length of a representative local variable's scope. Since there were some methods as shown in Fig.4, where the minimum length of the scope is zero. As all local variables are valid only within a (part of the) method, the majority of them are around a few to ten lines of code. In order to filter out extreme data which may be noise in our analysis, we will use only the data whose scopes are in between 25 percentile ($Q_1 = 4$) and 75 percentile ($Q_3 = 19$) of their distribution. By this data filtering, the number of our samples are reduced to $1,872$. Table V gives the number of methods belong to each category M_{ij} (for $i = 0, 1$; $j = 1, 2, 3$) after this filtering.

Table VI shows the distribution of the number of bug fixes which had occurred in methods over their upgrades. About 18% of methods seemed to have had a hidden fault and have fixed through their code changes. Since we already filtered out the methods such that the scope of the representative local variable was wide, most of the methods in our dataset were small-sized and thus possibly more simple in structure. Hence, conventional size metrics and structural complexity metrics would be ineffective for analyzing the fault-proneness of methods in detail. It would be worth it to focus on a feature of methods other than the size and complexity. A local variable name might be yet another useful feature to be focused on.

E. Results and Discussion: Comparison of Bug Fix Rates by Category

Table VII presents the bug fix rate in each category M_{ij} (for $i = 0, 1$; $j = 1, 2, 3$). There seem to be differences in the

TABLE III
DISTRIBUTION OF LENGTH OF REPRESENTATIVE LOCAL VARIABLE NAMES.

Unit	Min.	Q_1	Median	Q_3	Max.
Character	1	4	6	10	36
Word	1	1	1	2	6

(Q_1: 25 percentile; Q_3: 75 percentile)

TABLE IV
DISTRIBUTION OF SCOPE OF REPRESENTATIVE LOCAL VARIABLES.

Min.	Q_1	Median	Q_3	Max.
0	4	9	19	793

(Q_1: 25 percentile; Q_3: 75 percentile)

```
private void doConnectToRunningChanged() {
 if (doStartGdbServer.getSelection()) {
   boolean enabled = doConnectToRunning.getSelection();
 }
}
```

Fig. 4. An instance of local variable whose scope is zero ("enabled").

TABLE V
NUMBER OF METHODS BELONG TO EACH CATEGORY.

Category	Non-Compound		Compound Name	Total
	≤ 4	> 4		
Non-Commented	401 (M_{01})	527 (M_{02})	427 (M_{03})	$1,355$ (C_0)
Commented	139 (M_{11})	164 (M_{12})	214 (M_{13})	517 (C_1)
Total	540 (V_1)	691 (V_2)	641 (V_3)	$1,872$

TABLE VI
DISTRIBUTION OF NUMBER OF BUG FIXES OBSERVED IN METHODS AND BUG FIX RATE.

Min.	Q_1	Median	Q_3	Max.	Rate
0	0	0	0	5	18.1%

(Q_1: 25 percentile; Q_3: 75 percentile)

bug fix rates among categories. The minimum bug fix rate is 0.135 in M_{01} and the maximum bug fix rate is 0.252 in M_{13}, so the latter rate is about twice larger than the former one.

We did a χ^2 test for the differences of bug fix rates in the results. The test confirmed that there are statistically significant differences among the bug fix rates in the categories, at $p = 0.0053 < 1\%$ level of significance ($\chi^2 = 16.6$; degree of freedom $= 5$). That is to say, the above categorization of methods by focusing on the name of local variables and comments is meaningful for discussing the differences of fault-proneness in the methods.

In the categories of non-commented methods M_{0j} (for $j = 1, 2, 3$), we can observe an increasing trend in the bug fix rate (BFR): $\mathrm{BFR}(M_{01}) = 0.135 < \mathrm{BFR}(M_{02}) = 0.165 < \mathrm{BFR}(M_{03}) = 0.211$ (see Table VII and Fig.5(a)). We also identified that the increasing tendency is statistically significant through the Cochran-Armitage test [18] at $p = 0.0035 < 1\%$ level of significance ($\chi^2 = 8.52$; degree of freedom $= 1$). From this trend, we can say that methods having representative local variables with shorter names are likely to be better in terms of fault-proneness, and the ones with compound names are worse than others.

On the other hand, in the categories of commented methods M_{1j} (for $j = 1, 2, 3$), we cannot identify an increasing trend in the bug fix rate; they seems that $\mathrm{BFR}(M_{11}) = 0.180 \simeq \mathrm{BFR}(M_{12}) = 0.177 < \mathrm{BFR}(M_{13}) = 0.252$ (see Table VII and Fig.5(b)).

For all three categories, their bug fix rates were higher than ones of non-commented methods, i.e., $\mathrm{BFR}(M_{0j}) < \mathrm{BFR}(M_{1j})$ (for $j = 1, 2, 3$):

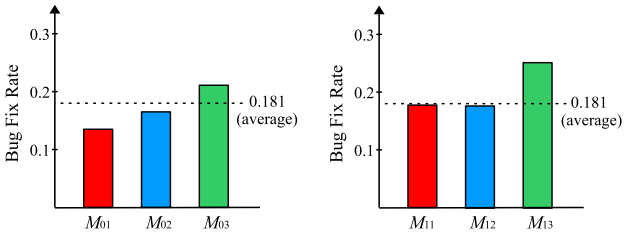
- $\mathrm{BFR}(M_{01}) = 0.135 < \mathrm{BFR}(M_{11}) = 0.180$,
- $\mathrm{BFR}(M_{02}) = 0.165 < \mathrm{BFR}(M_{12}) = 0.177$, and
- $\mathrm{BFR}(M_{03}) = 0.211 < \mathrm{BFR}(M_{13}) = 0.252$.

Thus, the commented methods seem to be riskier in fault-proneness than the non-commented methods. Similar trends in regard to comments have been reported in the previous work [11], [15] as well. Since programmers might want to add comments when they considered that their code is difficult to understand without an explanation, the presence of comments would be a sign indicating that the code is complicated.

Notice that the bug fix rates in the categories of compound names, M_{03} and M_{13}, are the highest ones among categories; Only those two categories show bug fix rates which are higher than the average of all (18.1%) (see Fig.5). Thus, the methods having representative local variables with compound names are likely to be fault-prone regardless of the presence of comments.

TABLE VII
BUG FIX RATES BY CATEGORY.

Category	Non-Compound name ≤ 4		Non-Compound name > 4		Compound name	
Non commented	M_{01}	0.135 ($\frac{54}{401}$)	M_{02}	0.165 ($\frac{87}{527}$)	M_{03}	0.211 ($\frac{90}{427}$)
Commented	M_{11}	0.180 ($\frac{25}{139}$)	M_{12}	0.177 ($\frac{29}{164}$)	M_{13}	0.252 ($\frac{54}{214}$)



(a) Non-commented (b) Commented

Fig. 5. Comparison of bug fix rates by category.

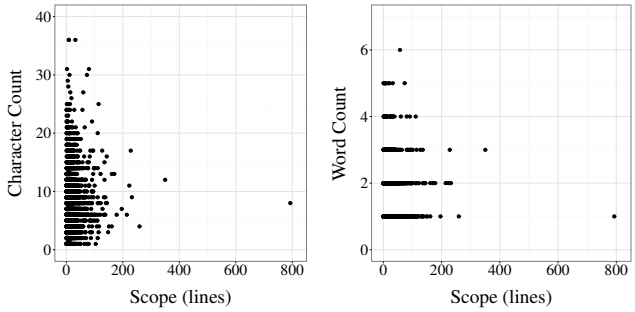
F. Results and Discussion: Comparison of Bug Fix Rates over Scope

This subsection compares the bug fix rates among the categories from another in-depth perspective of local variable's property, "scope."

We first checked correlations of the length of a local variable name with its scope. There do not seem to be specific correlation between the length of local variable's name and the length of its scope (see Fig.6): Spearman rank-correlation coefficients in character count and in word count were 0.083 and 0.0003, respectively. Hence, the length of a local variable's name is statistically independent of the length of its scope, and the scope is not a confounding factor for discussing the fault-proneness of methods by using their local variable's name.

To observe the changes in fault-proneness over variable's scope, we computed the moving averages of bug fix rates by varying the focusing interval of scope $[s - 5, s + 5]$ for $s = 9, 10, \ldots, 14$; in simplified terms, we obtained the bug fix rates of methods whose representative local variable's scope is "around s" ($s \pm 5$), where the lower and the upper limit of s are decided so as to keep the interval $[s - 5, s + 5]$ within the scope range of all data: between 4 and 19. For example, if $s = 9$ then $[4, 14]$ is the focusing interval, we focus only on the methods whose representative local variable's scope is "around 9" (9 ± 5). Figure 7 shows those results.

In Fig.7(a), we observed the relationships of bug fix rates regardless of scope: $\mathrm{BFR}(M_{01}) < \mathrm{BFR}(M_{02}) < \mathrm{BFR}(M_{03})$,



(a) Character count (b) Word count

Fig. 6. Scatter diagrams: the length of variable's name vs. the length of variable's scope.

which are similar to the results shown in Fig.5(a). Thus, we can say with emphasis: while the fault-proneness of methods having representative local variables with shorter names are low, the methods having representative local variables with compound names are high. Since the gap in the bug fix rate between M_{01} and M_{02} becomes smaller as the scope gets wider, the superiority of a shorter name may be limited to a narrower scope. If a local variable with a short and simple name is used in a wider scope, it might cause an abuse of the variable or a poor understandability of the program's behavior. While Kernighan and Pike [10] said to give a short and simple name to a local variable, they did not recommend such a naming in any case, and their argument supposed the case that a local variables was used in just "locally" within a part of a program. The results observed in Fig.7(a) seem to support such a programming heuristic.

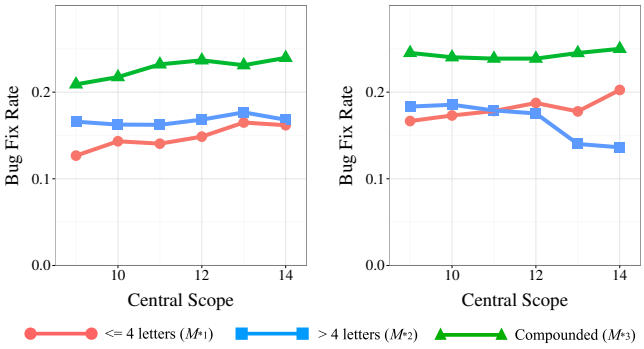
In Fig.7(b), while M_{11} ($\leq$ 4 letters) are better than M_{12} ($>$ 4 letters) with narrower scopes around 9 or 10, their magnitude relationship inverts as their scope gets wider. That would be the reason why $\mathrm{BFR}(M_{11}) \simeq \mathrm{BFR}(M_{12})$ in Fig.5(b). Therefore, we can say that a shorter name is better with a narrower scope, but cannot claim a shorter name is *always* better. If programmers wanted to add comments, there would be a lack of clarity in their code. In such a case, a shorter name with a wider scope might spur the program's poor comprehensibility. On the other hand, compound names always show the worst (highest) bug fix rates regardless of scope, similar to the results in Fig.7(a). Although compound names are usually descriptive, they seem to be signs of fault-prone methods. If a programmer wanted to give a compound name to a local variable, the role of the variable would be somewhat complicated, so methods having such local variables might be riskier than the others in terms of fault-proneness.

G. Answers to RQs

From the results of Sections III-E and III-F, we summarize our findings for RQ1 and RQ2 in the following.

For RQ1, we conclude that methods having local variables with compound names are likely to be faulty regardless of scope. Although we do not imply that compound names cause faults in methods, the presence of a local variable with a compound name may be a clue finding a risky part from the perspective of the fault-proneness in a method. Such a local variable might have an important role or a more complex role in the program, so they have to be reviewed more carefully.

For RQ2, we can say that shorter names are better for local variables with narrower scope. As a scope gets wider, the positive effect of shorter names seems to be decayed. While a short and simple name would be preferable as mentioned in some coding conventions and programmers' heuristics [8], [9], [10], our empirical results quantitatively showed that the variable's scope is also a feature worthy of consideration. Moreover, the presence of comments may degrade the superiority of shorter names as their scopes get wider. Therefore, we should take into account not only the composition of local variable's name but also its scope and comments in the code review.



(a) Non-commented (b) Commented

Fig. 7. Moving averages of bug fix rates over scope.

H. Threats to Validity

This empirical analysis has been conducted for Java products. While another programming language might produce different results, there would not be essential differences in the concept of local variables and comments, among Java and many other modern programming languages. Thus, the difference in programming language would not be a serious threat to validity.

In order to avoid the data selection bias, we adopted a random sampling in our data collection. Moreover, we used popular different sized OSS products from different domains. Therefore, our construction of dataset would not be a threat to validity.

Since our data is collected from the initial version of the methods, some methods might be no longer used today. However, all methods in our dataset are included in the latest version of the product because we made our method list by checking the latest version of their source files as described in Section III-B. Moreover, we did a random sampling from them. Thus, we consider it will not be a serious threat to validity in our empirical work.

Our definition of compound name is based on the notion of camel case. If there are local variables whose names are composed by another rule such as the snake case, e.g., "`number_of_items`," they are wrongly categorized into non-compound names. Thus, we rechecked all representative local variables' names included in our data set, then we found only two variables having snake case names, "`s_descriptors`" and "`size_h`." Due to the small number of error cases, our name splitting method was not a serious threat to validity.

IV. Related Work

Lawrie et al. conducted a survey on names of identifiers in terms of their comprehensibility for over 100 programmers [7]. In their survey research, they classified names of identifiers into three categories (1) fully-spelled name, (2) abbreviated name and (3) initial letter—for example, (1) "`count`," (2) "`cnt`" and (3) "`c`"—, then compared their understandability.

Their results showed that fully-spelled names were the easiest to understand but that there did not seem to be significant differences with abbreviated names in their comprehensibility level. While their work provides a useful motivation to study whether a shorter name is better or not, they did not discuss the fault-proneness of program.

Kawamoto and Mizuno [19] conducted an empirical study with two OSS products and reported that a class including a long identifier tends to be faulty. While their work is one of our most significant previous studies, our work focuses on a finer-grained artifact—local variable—and conducts a statistical analysis with taking into account of the scopes and the comments.

Binkley et al. [20] focused on the relationship between the length of identifier (including a variable's name, a method's name and a class's name) and the human short-term memory. They identified that identifiers with long names are related to a difficulty in program comprehension. They were concerned that a long chain, e.g., "`class.firstAssignment().name.trim()`," would cause a loss of the readability of the code. While the research viewpoint differs from our work, the fundamental concern about the length of name is common, and it seems to be well accorded with our results showing the compound names are not recommended for local variables.

Aman et al. [11] reported an empirical analysis showing that Java methods having local variables with long names are more likely to be fault-prone and change-prone than the other methods. That report is our significant previous work, and this paper focuses more detailed features of local variables, i.e., the composition of name and their scopes. While another work by Aman et al. [15], reporting that commented programs tend to be more fault-prone, is also our important previous work, we conduct a further analysis examining combinations of the local variable's name, the scope and the comments in this paper.

V. Conclusion

We have focused on programming artifacts which may vary among individuals: local variables' names and comments. Popular code conventions say that names of local variables should be shorter and simple, and it seems to have been a heuristic of programmers. We empirically evaluated the heuristic in terms of fault-proneness by checking the names of local variables, their scopes and the presence of comments. The empirical analysis for the data from nine popular OSS products showed the following three findings.

(1) Local variables with compound names can be signs of fault-prone methods.
(2) Methods having the representative local variables with non-compound and shorter names (≤ 4 letters) are less fault-prone, but their positive effects are decayed as their scopes get wider (around 10 or more lines).
(3) Methods having comments in their bodies are also more likely to be faulty.

These findings are expected to be useful guidelines for more efficient code reviews.

One of our significant future works is to conduct further analyses of local variables' names, which include an application of the natural language processing technologies to evaluate the meaning of local variables' names. A further analysis with products written in a programming language other than Java is also our future project in order to ensure the generality of the above findings.

Acknowledgment

This work was supported by JSPS KAKENHI #16K00099. The authors would like to thank anonymous reviewers for their helpful comments.

References

[1] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. N.J.: John Wiley & Sons, 2004.
[2] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proc. 9th Joint Meeting on Foundations of Softw. Eng.*, Aug. 2013, pp. 202–212.
[3] P. L. Li, J. Herbsleb, M. Shaw, and B. Robinson, "Experiences and results from initiating field defect prediction and product test prioritization efforts at ABB Inc." in *Proc. 28th Int'l Conf. Softw. Eng.*, May 2006, pp. 413–422.
[4] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, July 2008.
[5] Y. Liu, T. Khoshgoftaar, and N. Seliya, "Evolutionary optimization of software quality modeling with multiple repositories," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 852–864, Nov 2010.
[6] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proc. 2013 Int'l Conf. Softw. Eng.*, May 2013, pp. 432–441.
[7] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *Proc. 14th Int'l Conf. Program Comprehension*, June 2006, pp. 3–12.
[8] Free Software Foundation, "Gnu coding standards," https://www.gnu.org/prep/standards/.
[9] Sun Microsystems, "Code conventions for the java programming language," http://www.oracle.com/technetwork/java/codeconvtoc-136057.html.
[10] B. W. Kernighan and R. Pike, *The practice of programming*. Boston, MA: Addison-Wesley Longman, 1999.
[11] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Empirical analysis of change-proneness in methods having local variables with long names and comments," in *Proc. 2015 ACM/IEEE Int'l Symp. Empirical Softw. Eng. and Measurement*, Oct. 2015, pp. 50–53.
[12] M. J. Sousa and H. Moreira, "A survey on the software maintenance process," in *Proc. Int'l Conf. Softw. Maintenance*, Nov. 1998, pp. 265–274.
[13] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proc. 2008 Int'l Symp. Softw. Testing and Analysis*, 2008, pp. 121–130.
[14] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley Longman, 1999.
[15] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Lines of comments as a noteworthy metric for analyzing fault-proneness in methods," *IEICE Trans. Inf. & Syst.*, vol. E98-D, no. 12, pp. 2218–2228, Dec. 2015.
[16] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. Int'l Workshop on Mining Softw, Repositories*, May 2005, pp. 1–5.
[17] H. Aman, "An empirical analysis of the impact of comment statements on fault-proneness of small-size module," in *Proc. 19th Asia-Pacific Softw. Eng. Conf.*, Dec. 2012, pp. 362–367.
[18] A. Agresti, *Categorical Data Analysis*, 2nd ed. N.J.: Wiley, 2002.
[19] K. Kawamoto and O. Mizuno, "Predicting fault-prone modules using the length of identifiers," in *Proc. 4th Int'l Workshop on Empirical Softw. Eng. in Practice*, Oct. 2012, pp. 30–34, Japan.
[20] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, "Identifier length and limited programmer memory," *Science of Computer Programming*, vol. 74, no. 7, pp. 430–445, May 2009.

Towards improved Adoption: Effectiveness of Research Tools in the Real World

Richa Awasthy
Australian National University
Canberra, Australia
Email: richa.awasthy@anu.edu.au

Shayne Flint
Research School of Computer Science
Australian National University
Canberra, Australia
shayne.flint@anu.edu.au

Ramesh Sankaranarayana
Research School of Computer Science
Australian National University
Canberra, Australia
ramesh@cs.anu.edu.au

***Abstract*—One of the challenges in the area of software engineering research has been the low rate of adoption by industry of the tools and methods produced by university researchers. We present a model to improve the situation by providing tangible evidence that demonstrates the real-world effectiveness of such tools and methods. A survey of practising software engineers indicates that the approach in the model is valid and applicable. We apply and test the model for providing such evidence and demonstrate its effectiveness in the context of static analysis using FindBugs. This model can be used to analyse the effectiveness of academic research contributions to industry and contribute towards improving their adoption.**

I. Introduction

The success of software engineering research in universities can be measured in terms of the industrial adoption of methods and tools developed by researchers. Current adoption rates are low [1] and this contributes to a widening gap between software engineering research and practice. Consider, for example, code inspections, which according to Fagan's law, are effective in reducing errors in software, increasing productivity and achieving project stability [2]. Static analysis tools developed by university researchers help automate the code inspection process. However, the use of such tools has not obtained widespread adoption in industry. One reason for this limited adoption is that researchers often fail to provide real-world evidence that the methods and tools they develop are of potential value to industry practitioners [1], [3].

One approach to providing such evidence is to conduct experiments that demonstrate the effectiveness of research tools in a real-world context. We apply this approach to analyse the effectiveness of a static analysis tool. In doing so, we demonstrate that such experimentation can contribute to closing the gap between research and practice.

The structure of this paper is as follows: Section II provides the background of our work leading to the proposed model; Section III presents a survey which shows that real world evidence can positively influence the decision of software engineers to use research tools; Section IV explains our experimental method which uses FindBugs [4] to analyse real world bugs in Eclipse [5] code; Section V discusses how simple experiments like ours can encourage more developers to use tools developed by researchers and thus contribute to closing the gap between research and practice. Section VI provides an overview of related research; Section VII presents conclusions and discusses future research.

II. Background

Since the 1970's there have been ongoing efforts to increase the adoption of research outcomes outside of universities [6]. As a result of the United States Bayh Dole Act in 1980 [7], universities began to establish *Technology Transfer Offices* (TTOs) to facilitate the transfer of knowledge from universities to industry [8]. However, the effectiveness of TTOs has been questioned in recent years [9], [10] and there is a need to look beyond TTOs to improve adoption of academic research in industry.

Researchers in universities are working towards addressing significant problems. The outcome of their work can be a tool or method which may or may not achieve wide-spread industry adoption. A key factor limiting the readiness of these research outcomes for adoption in industry is a lack of tangible evidence that they would be effective in practice [1]. This suggests that demonstrating the effectiveness of a tool in practice can contribute to improved adoption.

Figure 1 depicts our model for demonstrating the real-world effectiveness of research tools and methods. The model involves 4 steps with intermediate activities. First step is to identify a problem to address. Step 2 is to develop a tool or a method to address the problem. The intermediate iterative activity involved between these 2 steps is the process of solution formulation, which involves adding new ideas to the available state of the art. These steps are followed by iterative testing for validation in Step 3. Step 3 confirms the readiness of the research outcome for adoption. An idea should be validated in a practical setting [11] to improve its adoption. Our model respects this viewpoint and emphasises the importance of tangible evidence from a practical setting in Step 4. Researchers should test their research outcomes in a scenario that involves real world users who are an important stakeholder for industry to increase the relevance of the evidence for industry. Demonstrating the effectiveness of research outcomes in real-world scenario will lead to change in industry perception and improved adoption of the research outcomes.

We test the applicability of this model in the static analysis context by identifying a static analysis tool created by university researchers and analysing its effectiveness in a real-world scenario.

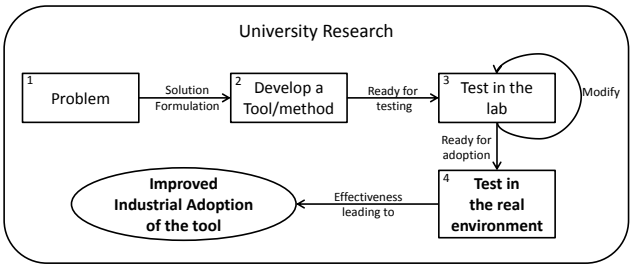


Fig. 1. Proposed model for improving adoption of university research by industry

III. The impact of evidence from real-world scenario

In order to understand the impact of evidence from a user-scenario on real-world decisions to use a research tool, we conducted an on-line survey of software developers.

The survey uses static analysis as an example and was prepared and delivered using our university's online polling system [12]. Participants were invited by email which included a link to the on-line survey and a participant information statement. On completion of the survey, we manually analysed the results.

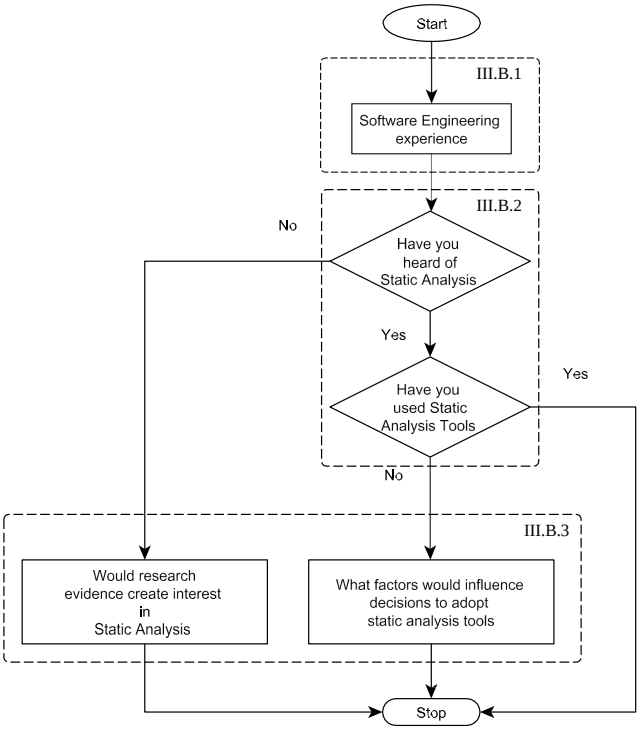


Fig. 2. Flowchart for the survey questionnaire design

A. Participants

We invited 20 software industry practitioners and around 10 computer science researchers with industry experience in software development.

B. Survey Questions

The survey data consisted of responses to the sequence of questions depicted in Figure 2 and described below.

1) Software engineering experience: We gathered information about the level of software development expertise of each participant so that we can understand any relationship between experience and use of static analysis tools.

2) Static analysis knowledge: We asked the following questions to determine each participant's level of understanding and use of static analysis tools.

a) '*Static analysis is the analysis of computer software to find potential bugs without actually executing the software. Have you heard of static analysis before?*'
b) '*Have you used any automated static analysis tools during software development (e.g. FindBugs, Coverity)?*'

Answers to these questions were used to determine the final question we asked, as indicated in Figure 2.

3) The impact of the tangible evidence: At the end of the survey each participant who has not used static analysis tools was asked a question to determine the impact that tangible evidence (that the tool can identify real bugs early in the software development life-cycle) might have on their approach to static analysis. The exact question asked depended on their answers to questions described in Section III-B2. Specifically:

1) Participants who had no knowledge of static analysis were asked '*Would our research results interest you in gaining knowledge of static analysis and adoption of automated static analysis tools?*'.
2) Participants who knew about static analysis but had not used any static analysis tools (our primary group of interest) were asked to rate the impact that the following factors would have on their decision to adopt static analysis tools. A Likert scale was used with 5 options (*No influence, May be, Likely, Highly Likely, and Definitely*).
   a) Effectiveness of tool in finding bugs
   b) Ease of use
   c) Integration of tool to development environment.
   d) License type.
   e) The availability of tangible evidence that the tool can identify real bugs early in the software development life-cycle - before they are reported by users.

C. Analysis of Survey Results

The response rate for our survey was high with 27 responses out of 30 invitations. Responses to the survey indicate that tangible evidence of real-world effectiveness of a tool has positive impact on decisions to adopt static analysis tools.

Analysis of the survey results provides the following specific findings:

1) *Software Engineering Experience* - As expected, participants had varied level of experience in software development. However, we do not find any direct relation between the experience and usage of tools.
2) *Static Analysis Knowledge* - Survey results show that 9 participants (33%) had no prior knowledge of static analysis. It is noteworthy that while the remaining 18 participants knew about static analysis, only 4 of them had used static analysis tools.
3) *Impact of the tangible evidence* - Our survey results show that tangible evidence has a positive impact on decisions to adopt static analysis tools. Out of the 9 respondents who had no prior knowledge of static analysis, 8 said that tangible evidence would interest them in gaining knowledge of static analysis and adopting automated static analysis tools. This is a valuable information indicating that providing evidence of effectiveness could contribute to improved adoption of research tools in industry.
   Of the 14 participants who had knowledge of static analysis but who had not used any tools, 7 participants (50%) indicated that tangible evidence would be *Highly Likely* or would *Definitely* influence their decision to adopt static analysis tools (Figure 3). Another four participants indicated that such evidence would be *Likely* to influence their decision. Considering the response of three participants as *May be*, it is possible that they respond positively, which will add to the percentage of participants agreeing that tangible evidence will influence the decision.

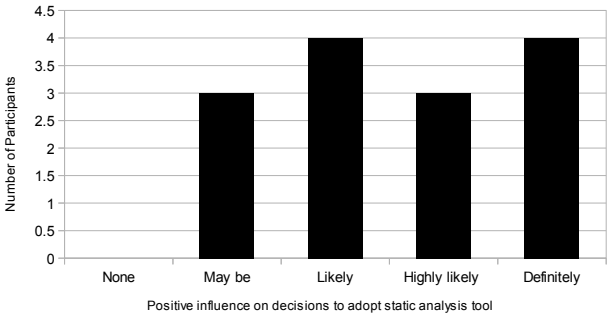


Fig. 3. The impact of tangible evidence on decisions to adopt static analysis tools

As shown in Figure 4, our results also indicate that other factors such as *Ease of use*, *IDE integration* and *License type* have a positive impact on decisions to adopt static analysis tools. It is interesting to note that under the *May be* and *Definitely* category, the top two influencing factors are *License* and *Tangible evidence*.

Our results clearly show that tangible evidence is an important factor in influencing decisions to adopt research tools in industry.

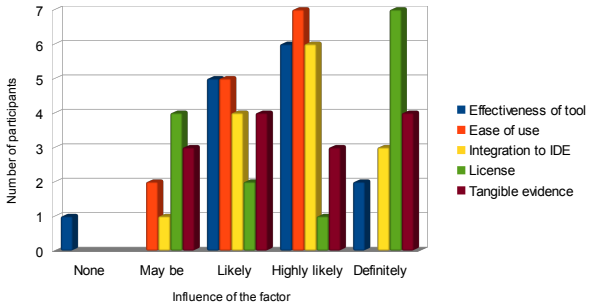


Fig. 4. Other factors influencing decisions to adopt static analysis tools

IV. Applicability of the Model

In order to test the applicability of our proposed model, we first had to identify an appropriate tool developed by researchers and a scenario to test its effectiveness. FindBugs version 3.0 was chosen for our research as according to the tool's website, there are few organizations using FindBugs [4]. This indicates low adoption of the tool in software industry. Also, it is an open-source static analysis tool with a university's trademark. We conducted an experiment with the FindBugs static analysis tool to analyse its effectiveness in the real world. To analyse the effectiveness in real-world, we wanted to determine if FindBugs is capable of finding bugs reported by real users of Eclipse. To do so, we adopted an approach to establishing a connection between warnings generated by the FindBugs static analysis tool and field bugs reported by Eclipse users on Bugzilla [13] that includes the below steps:

1) Use FindBugs to identify potential bugs in Eclipse class files.
2) Search the Eclipse bug-tracking system Bugzilla to identify bug reports that include stack traces.
3) Match the code pointed in Java classes associated with FindBugs warnings identified in 1) with code pointed by stack trace associated with the bugs identified in 2).

A. FindBugs

FindBugs analyses Java class files to report warnings and potential errors in the associated source code. The tool performs analysis on Java class files without needing access to the source code. It is stable, easy to use, and as mentioned in [14] has higher precision compared to other tools such as CodePro Analytix, PMD and UCDetector. It has a low rate of false positives [15], and has more than 400 types of bug classification along with categorization based on severity level. The analysis is based on bug patterns which are classified into nine categories: Bad Practice, Correctness, Experimental, Internationalization, Malicious Code Vulnerability, Multithreaded Correctness, Performance, Security, and Dodgy Code. The warnings reported by FindBugs can be further categorised within the tool as: Scariest (Ranks 1-4), Scary (Ranks 1-9) and Troubling (Ranks 1-14). The category includes all the bugs

with the ranking mentioned, for example, the 'Scary' category will list the bugs included under the 'Scariest' category, as well.

B. Eclipse

We identified Eclipse as the object of analysis as it is a large widely used open source project with good records of user reported bugs over many years and versions of the software. For our experimentation we focused on the analysis of Eclipse version 4.4 (Luna) because it was the current version at the time of our experimentation.

C. Identification of potential bugs

Java Jars associated with Eclipse versions 4.4 were analysed using FindBugs version 3.0. Findbugs generated a list of warnings pertaining to code considered as faulty.

D. Search for user reported bugs

The Eclipse project uses Bugzilla to track bugs reported by users. In order to identify bugs that could be associated with FindBugs warnings, we needed to identify bug reports that included a documented stack-trace. This was achieved by performing an advanced Bugzilla search for bugs that satisfied the following criteria:

- *Version:* 4.4,
- *Classification:* Eclipse,
- *Status:* NEW, ASSIGNED, VERIFIED [1].

We then inspected the query results to identify those bug reports that included a documented stack-trace.

E. Match FindBugs warnings with user reported Bugs

Our last step was to match warnings generated by FindBugs (Section IV-C) with user-reported bugs (Section IV-D). This was achieved using the following steps:

1) For each of the bugs identified using the procedure described in Section IV-D, we identified the Java class that was the likely source of the reported bug. This class was usually the one appearing at the top of the stack-trace. In some cases, we had to traverse through lower levels in the stack-trace to find matching classes.
2) We then searched for the above classes in the warnings generated by FindBugs (Section IV-C) and analysed the code associated with the warning. We did this by using the FindBugs class name filter feature to show warnings related to the class of interest.
3) Finding a matching line of code in the FindBugs warnings establishes a connection between the warnings generated by FindBugs and the bugs reported by users.

[1]Because our experiment looks at the ability of FindBugs to find bugs that have not been fixed, we ignore the bugs with CLOSED status. In addition, we do not consider the possibility of bugs that have been closed incorrectly.

F. Results of the Experiment

1) Analysis of FindBugs warnings for Eclipse version 4.4: Our analysis of FindBugs warnings for Eclipse version 4.4 showed that static analysis of Eclipse version 4.4 generated warnings in the categories of *Correctness* (652), *Bad Practice* (547), *Experimental* (1), *Multithreaded correctness* (390), *Security* (3), and *Dodgy code* (55) under the rank range of *Troubling* (Rank 1-14), as depicted in Figure 5. We focused on the *Scariest* warnings (Rank 1-4), considering them as real problems requiring attention. There were 82 *Scariest* warnings in total. These comprised 81 warnings in the *Correctness* category and one in the *Multi-threaded correctness* category.

Additional investigation found that warnings in the *Correctness* category included a range of coding errors such as comparisons of incompatible types, null pointer dereferences and reading of uninitialized variables.

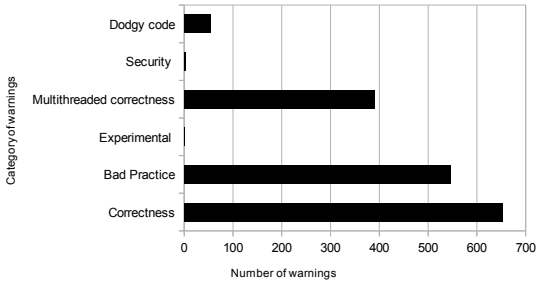


Fig. 5. Number of warnings in each category in Eclipse 4.4

2) Connection between FindBugs warnings and user reported bugs: Execution of the query described in Section IV-D resulted in a dataset of 2575 bugs, which included 347 enhancements. We excluded the enhancements from our analysis. Out of the remaining bugs, we have analysed 1185 bug reports so far, 90 of which included a documented stack-trace.

We used the method described in Section IV-E to compare the stack-trace in these 90 bug reports with the warnings generated by FindBugs (Section IV-F1). We found that six of the user reported bugs could be associated with FindBugs warnings as presented in Table I. The data presented in the table includes the Bugzilla Bug ID, a description of the bug, and a description of the warning generated by FindBugs.

V. Discussion

The model proposed in this paper considers that in order to improve the adoption of university research outcomes, researchers need to demonstrate its effectiveness in a real-world scenario and think about the value of their research outcomes beyond the lab boundaries. Our main purpose in conducting the experiment described was to test the applicability of our proposed model by analysing the value of a research tool in industrial practice. Specifically, we evaluated the performance of the FindBugs static analysis tool by analysing its capability

TABLE I
USER-FILED BUGS IN ECLIPSE WITH ASSOCIATED WARNING IN FINDBUGS

Bug Id	Problem Description	FindBugs Warning
436138	NPE when select SWT.MOZILLA style in control example	Method call passes null for nonnull parameterThis method call passes a null value for a nonnull method parameter. Either the parameter is annotated as a parameter that should always be nonnull, or analysis has shown that it will always be dereferenced. Bug kind and pattern: NP - NP_NULL_PARAM_DEREF
414508	NPE trying to launch Eclipse App	Load of known null value. The variable referenced at this point is known to be null due to an earlier check against null. Although this is valid, it might be a mistake (perhaps you intended to refer to a different variable, or perhaps the earlier check to see if the variable is null should have been a check to see if it was nonnull). Bug kind and pattern: NP - NP_LOAD_OF_KNOWN_NULL_VALUE
433526	browser.getText() is throwing exception after Internet Explorer 11 install	Possible null pointer dereference There is a branch of statement that, if executed, guarantees that a null value will be dereferenced, which would generate a NullPointerException when the code is executed. Of course, the problem might be that the branch or statement is infeasible and that the null pointer exception can't ever be executed; deciding that is beyond the ability of FindBugs. Bug kind and pattern: NP - NP_NULL_ON_SOME_PATH
459025	Can't right-click items in manifest editor's extensions tab on OSX	Non-virtual method call passes null for nonnull parameter A possibly-null value is passed to a nonnull method parameter. Either the parameter is annotated as a parameter that should always be nonnull, or analysis has shown that it will always be dereferenced. Bug kind and pattern: NP - NP_NULL_PARAM_DEREF_NONVIRTUAL
427421	NumberFormatException in periodic Workspace Save Job	Boxing/unboxing to parse a primitive A boxed primitive is created from a String, just to extract the unboxed primitive value. It is more efficient to just call the static parseXXX method. Bug kind and pattern: Bx - DM_BOXED_PRIMITIVE_FOR_PARSING
428890	Search view only shows default page (NPE in PageBookView.showPageRec)	Possible null pointer dereference There is a branch of statement that, if executed, guarantees that a null value will be dereferenced, which would generate a NullPointerException when the code is executed. Of course, the problem might be that the branch or statement is infeasible and that the null pointer exception can't ever be executed; deciding that is beyond the ability of FindBugs. Bug kind and pattern: NP - NP_NULL_ON_SOME_PATH
426485	[EditorMgmt][Split editor] Each split causes editors to be leaked	Possible null pointer dereference There is a branch of statement that, if executed, guarantees that a null value will be dereferenced, which would generate a NullPointerException when the code is executed. Of course, the problem might be that the branch or statement is infeasible and that the null pointer exception can't ever be executed; deciding that is beyond the ability of FindBugs. Bug kind and pattern: NP - NP_NULL_ON_SOME_PATH

in generating warnings relating to real-world bugs reported by users of the Eclipse IDE.

Our results indicate that FindBugs is capable of identifying bugs that will manifest themselves as bugs reported by users. Since real-world evidence would influence the decision to adopt a tool as indicated in the survey results, FindBugs needs to improve the percentage of such warnings to make the evidence convincing. Currently, FindBugs does not have the intelligence to track the bug among the list of false positives that can manifest. Our research provides improvement directions to FindBugs in identifying the important bugs from the warning base by analysing the historical data and user requirement. FindBugs results can be improved by introducing a new 'user-impact' category to classify the warnings which will potentially have an impact in the client environment and hence, need immediate attention. For this, sufficient information from industrial practice needs to be applied into testing the research tool.

The model appears simple but it is challenging for researchers to identify a scenario and approach users and/or data to demonstrate the effectiveness of their tool or methods. It might be difficult for them to find the user-filed data always. Also, once they have the data to demonstrate the effectiveness, they need a mechanism to propagate it to industry. Also, the concern about the relevance of research suggests that researchers need to think about the relevance of the problem they are trying to address. These factors indicate that universities and industry need to start collaborating at an early stage and consider co-developing whenever possible and feasible. This can pave a good start towards improving the relevance and adoption of research outcomes in general, and bridging the gap between industry and academia.

A. Limitations and Threats to Validity

Our experiments were limited by the small number of Eclipse bugs reported with a documented stack-trace. It is important to note that the ability to analyse only 90 of the 1185 bugs considered reflects a limitation of our approach and does not reflect a limitation of FindBugs. There are some limitations to the validity of our experiments:

1) FindBugs does not always point to the exact line number referred to in the stack trace. It might be possible that the source of error could be different from the warnings provided by FindBugs.
2) While it is likely that the FindBugs warnings listed in Table I are the actual cause of the listed real-world bugs reported by Eclipse users, we cannot be certain of this.
3) As there is lack of literature detailing the use of static analysis tools like FindBugs by the Eclipse development team, a comparison study was not feasible.

4) This test highlights the relevance of static analysis tools, though the effectiveness in real-world projects, particularly large scale projects, cannot be confirmed as the sample size of our survey is small. However, by considering the sample sizes of 20 and 18 used in related studies [16], [17], we decided to proceed with our sample size of 27 participants.
5) The conclusion might not be generalised as the results are specific to the static analysis context. The proposed model needs to be validated regarding the tools in other phases of software development.

VI. Related Work

Adoption of software engineering research outcomes in industry practice has been a concern [1], [18]. There have been ongoing efforts to improve the adoption of research outcomes since the 1970's. However, the efforts mainly focused on approaches to increase university-industry collaboration for improving adoption through technology transfer. These efforts include policy changes leading to establishment of TTOs in universities and proposing models for effective technology transfer [8], [19]. However, TTOs generally focus on building collaborative relationships between researchers and industry [20] rather than the readiness of research outcomes for adoption in industry. In this paper we demonstrated how some simple experiments can analyse the effectiveness of software engineering research tools in practice. Specifically we analysed the effectiveness of a static analysis tool.

Various experiments have been conducted to demonstrate the effectiveness of the FindBugs static analysis tool by showing that it is able to detect the specific problems it has been designed to detect [15], [21], [22]. However, our experiment has been conducted in unconstrained environment that involves real-world scenario that has impacted clients. Ruthruff et al. [23] involved developers in determining which reports were useful and which ones were not. This information was used to filter FindBugs warnings so that only those that developers found useful were reported. We retrace the user-filed bug to the warning generated by the static analysis tool. This can pave way to create an intelligent mechanism to prioritise bugs based on user-impact.

Al Bessey et al. [24] identify several factors that impacted the industry adoption of their static analysis tool Coverity [25]. They include trust between the researchers and industry users, the number of false positives, and the capability of the tool to provide warnings relating to problems which have had a significant impact on its users. Our work also confirms that tool's capability is important. It also identifies that licensing, IDE integration and ease of use are significant factors.

Johnson et al. [16] investigated the reasons behind the low adoption rate of static-analysis tools despite their proven benefits in finding bugs. Their investigations confirmed that large numbers of false positives is a major factor in low adoption rates. We note that their findings were based on the survey of developers who had all used static analysis tools. This meant that authors were not able to comment on whether low-adoption rates were due to a lack of awareness of static analysis tools among developers.

None of the work described above analyses the effectiveness of FindBugs in identifying problems that manifest themselves as real-world bugs reported by users. The experiments described in this paper analyse the connection between warnings generated by the FindBugs static analysis tool and field defects reported by Eclipse users on Bugzilla bringing in client into the perspective. The experiments test the applicability of our proposed model in the static analysis context.

VII. Conclusion and Future Work

We have proposed a model to contribute towards improving the adoption of research tools by industry by demonstrating the effectiveness of the tool in real world scenario. We have presented a mechanism which involves a research tool as a medium of building tangible evidence. A survey of software developers supports our hypothesis that such tangible evidence of effectiveness of a tool can have a positive influence on real-world decisions to adopt static analysis tools.

Further experiment for testing the applicability of the model in the static analysis context was conducted. In this experiment, by establishing a connection between user-reported bugs and warnings generated by the FindBugs static analysis tool, we have demonstrated the ability of static analysis tools to eliminate some defects before software is deployed. However, the evidence needs to be stronger regarding the number of such connections in order to be more convincing and improving the industrial adoption of the tool.

Future research would present more detailed analysis of the complete list of the bugs found in Section IV-F2, which will provide us precise data about the effectiveness of the tool according to our approach. Our approach also presents a scenario where industry and university researchers can work together to create more useful tools. We plan to discuss these results with the FindBugs development team to explore the possibility of strengthening the evidence and devising a new classification *user-impact* to indicate the warnings that would manifest in client-environment.

Finally, we would like to adapt this approach to explore the effectiveness of research tools involved in other phases of the software development life-cycle.

References

[1] D. Rombach and F. Seelisch, "Balancing agility and formalism in software engineering," B. Meyer, J. R. Nawrocki, and B. Walter, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Formalisms in Software Engineering: Myths Versus Empirical Facts, pp. 13–25.
[2] A. Endres and H. D. Rombach, *A handbook of software and systems engineering: empirical observations, laws and theories*. Pearson Education, 2003.
[3] M. Ivarsson and T. Gorschek, "A method for evaluating rigor and industrial relevance of technology evaluations," *Empirical Software Engineering*, vol. 16, no. 3, pp. 365–395, 2011.
[4] University of Maryland, "Findbugs," viewed May 2015, http://findbugs.sourceforge.net, 2015.
[5] The Eclipse Foundation, "Eclipse," viewed May 2015, http://www.eclipse.org, 2015.

[6] R. Grimaldi, M. Kenney, D. S. Siegel, and M. Wright, "30 years after bayh–dole: Reassessing academic entrepreneurship," *Research Policy*, vol. 40, no. 8, pp. 1045–1057, 2011.

[7] W. H. Schacht, "Patent ownership and federal research and development (R&D): A discussion on the Bayh-Dole act and the Stevenson-Wydler act." Congressional Research Service, Library of Congress, 2000.

[8] D. S. Siegel, D. A. Waldman, L. E. Atwater, and A. N. Link, "Commercial knowledge transfers from universities to firms: improving the effectiveness of university–industry collaboration," *The Journal of High Technology Management Research*, vol. 14, no. 1, pp. 111–133, 2003.

[9] J. G. Thursby, R. Jensen, and M. C. Thursby, "Objectives, characteristics and outcomes of university licensing: A survey of major us universities," *The Journal of Technology Transfer*, vol. 26, no. 1-2, pp. 59–72, 2001.

[10] D. S. Siegel, D. A. Waldman, L. E. Atwater, and A. N. Link, "Toward a model of the effective transfer of scientific knowledge from academicians to practitioners: qualitative evidence from the commercialization of university technologies," *Journal of Engineering and Technology Management*, vol. 21, no. 1, pp. 115–142, 2004.

[11] R. L. Glass, "The relationship between theory and practice in software engineering," *Communications of the ACM*, vol. 39, no. 11, pp. 11–13, 1996.

[12] The Australian National University, "Anu polling online," viewed July 2015, ¡https://anubis.anu.edu.au/apollo/¿, 2015.

[13] Creative Commons License, "bugzilla," viewed May 2015, https://www.bugzilla.org, 2015.

[14] A. K. Tripathi and A. Gupta, "A controlled experiment to evaluate the effectiveness and the efficiency of four static program analysis tools for java programs," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2014, p. 23.

[15] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM Sigplan Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[16] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 672–681.

[17] L. Layman, L. Williams, and R. S. Amant, "Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE, 2007, pp. 176–185.

[18] S. Beecham, P. OLeary, I. Richardson, S. Baker, and J. Noll, "Who are we doing global software engineering research for?" in *2013 IEEE 8th International Conference on Global Software Engineering*. IEEE, 2013, pp. 41–50.

[19] S. L. Pfleeger, "Understanding and improving technology transfer in software engineering," *Journal of Systems and Software*, vol. 47, no. 2, pp. 111–124, 1999.

[20] D. S. Siegel, R. Veugelers, and M. Wright, "Technology transfer offices and commercialization of university intellectual property: performance and policy implications," *Oxford Review of Economic Policy*, vol. 23, no. 4, pp. 640–660, 2007.

[21] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Using findbugs on production software," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 2007, pp. 805–806.

[22] ——, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2007, pp. 1–8.

[23] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: an experimental approach," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 341–350.

[24] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[25] Synposys Inc., "Coverity," viewed May 2015, http://www.coverity.com, 2015.

Code Coverage Analysis of Combinatorial Testing

Eun-Hye Choi*, Osamu Mizuno†, Yifan Hu†
* National Institute of Advanced Industrial Science and Technology (AIST), Ikeda, Japan
Email: e.choi@aist.go.jp
† Kyoto Institute of Technology, Kyoto, Japan
Email: o-mizuno@kit.ac.jp, y-hu@se.is.kit.ac.jp

***Abstract*—Combinatorial *t*-way testing with small *t* is known as an efficient black-box testing technique to detect parameter interaction failures. So far, several empirical studies have reported the effectiveness of *t*-way testing on fault detection abilities. However, few studies have investigated the effectiveness of *t*-way testing on *code coverage*, which is one of the most important coverage criteria widely used for software testing. This paper presents a quantitative analysis to evaluate the code-coverage effectiveness of *t*-way testing. Using three open source utility programs, we compare *t*-way testing with exhaustive (all combination) testing w. r. t. code coverage and test suite sizes.**

***Keywords*-Combinatorial testing; *t*-way testing; Exhaustive testing; Code coverage; Line coverage; Branch coverage.**

I. Introduction

Combinatorial testing [15], [20] is a common black-box testing to detect failures caused by parameter interactions. Modern software systems have a lot of parameters, and thus their interactions are too numerous to be exhaustively tested. Combinatorial *t-way testing* [15], [20], where *t* is called an *interaction strength*, addresses this problem by testing all value combinations of *t* parameters with small *t*, instead of testing all parameter-value combinations exhaustively. *t*-way testing has been applied to e. g. conformance testing for DOM (Document Object Model) Events standard [19], rich web applications [18], commercial MP3 players [25], and a ticket gate system for transportation companies [14].

Kuhn et al. [16] investigated the fault detection effectiveness of *t*-way testing; their result showed that *t*-way testing with small interaction strength t (≤ 4) can efficiently detect most interaction failures while significantly reducing the number of test cases compared to *exhaustive testing*, which tests all parameter-value combinations. Other studies [2], [8], [25] also supported the result by Kuhn et al. [16].

On the other hand, as far as we know, the only work by Giannakopoulou et al. [10] reported the effectiveness of *t*-way testing on *code coverage*. They compared code coverage between their model-checker based exhaustive testing and 3-way testing with two program modules for a NASA air transportation system.

Code coverage, which measures what percentage of source code is executed by a test suite, has been considered as one of the most important coverage metrics for software testing and is required by many industrial software development standards (e. g. [1]). Therefore, the code coverage effectiveness of *t*-way testing would be of big interest to practitioners who consider applying *t*-way testing to their software testing.

Note that *t*-way testing is a black-box testing and thus is difficult to achieve 100% code coverage and its code coverage depends on the *system under test (SUT)* model, e. g. parameters and their values, designed for *t*-way testing. Therefore, in order to evaluate the code coverage effectiveness of *t*-way testing, we compare code coverage obtained by *t*-way testing with that by exhaustive testing, similarly to [10].

In order to quantitatively analyze the code coverage effectiveness of *t*-way testing compared to exhaustive testing, we set up the following two research questions:

- RQ1: How high code coverage can *t*-way testing achieve compared to exhaustive testing? Can *t*-way testing obtain higher code coverage earlier compared to exhaustive testing? How large interaction strength *t* is necessary for *t*-way testing to achieve the code coverage close to that by exhaustive testing?
- RQ2: With the same number of test cases, how different are *t*-way testing and exhaustive testing on code coverage?

For evaluating the code coverage effectiveness of *t*-way testing, RQ1 compares *t*-way testing and exhaustive testing in their original sizes, while RQ2 compares *t*-way testing and exhaustive testing in the same sizes.

To answer the above research questions, we perform a case study that analyzes *t*-way test suites with $1 \leq t \leq 4$ on two kinds of widely used code coverage; *line coverage* (i. e., statement coverage) and *branch coverage* (i. e., decision coverage). For an empirical case study, we use seventeen versions of three C program projects, flex, grep, and make, from the Software-artifact Infrastructure Repository (SIR) [7]. To prepare *t*-way test suites, we first construct SUT models with constraints from test plans in Test Specification Language (TSL) [21] of the repository. We next generate *t*-way test suites for the SUT models using two state-of-the-art *t*-way test generation tools, ACTS [3], [26] and PICT [6], [31]. We evaluate the code coverage effectiveness of *t*-way testing by comparing the *t*-way test suites and exhaustive test suites on the examining line coverage and branch coverage with test suite sizes.

Paper Organization: Section II-A explains combinatorial *t*-way testing and Section II-B describes related work on the effectiveness evaluation of *t*-way testing. Section III describes our experimental setting, and Section IV explains experimental results which answer the research questions. Section V concludes this paper.

TABLE I
AN EXAMPLE SUT MODEL.

Parameter	*Values*
Debug mode (= p_1)	on, off
Bypass use (= p_2)	on, off
Fast scanner (= p_3)	FastScan (= 1), FullScan (= 2), off
Constraint: (Fast scanner = FullScan) → (Bypass use ≠ on)	

TABLE II
AN EXAMPLE OF ALL POSSIBLE PAIRS OF PARAMETER-VALUES.

Param. pairs	*Parameter-value pairs*
(p_1, p_2)	(on, on), (on, off), (off, on), (off, off)
(p_1, p_3)	(on, 1), (on, 2), (on, off), (off, 1), (off, 2), (off, off)
(p_2, p_3)	(on, 1), (on, off), (off, 1), (off, 2), (off, off)

TABLE III
A 2-WAY TEST SUITE $\mathcal{T}_1$.

	p_1	p_2	p_3
1	on	on	1
2	on	off	off
3	off	off	1
4	off	on	off
5	off	off	2
6	on	off	1

TABLE IV
AN EXHAUSTIVE TEST SUITE $\mathcal{T}_2$.

	p_1	p_2	p_3
1	on	on	1
2	on	on	off
3	on	off	1
4	on	off	2
5	on	off	off
6	off	on	1
7	off	on	off
8	off	off	1
9	off	off	2
10	off	off	off

TABLE V
RELATED WORK.

	Metrics studied	
	Code coverage	Fault detection
Kuhn et al. (2004) [16]		✓
Zhang et al. (2012) [25]		✓
Petke et al. (2015) [22]		✓
Henard et al. (2015) [11]		✓
Choi et al. (2016) [4]		✓
Giannakopoulou et al. (2011) [10]	✓	
This paper	✓	

II. BACKGROUND AND RELATED WORK

A. Combinatorial t-way Testing

The *System Under Test (SUT)* for combinatorial testing is modeled from parameters, their associated values from finite sets, and constraints between parameter-values. For instance, the SUT model shown in Table I, has three parameters (p_1, p_2, p_3); the first two parameters have two possible values and the other has three possibilities. Constraints among parameter-values exist when some parameter-value combinations cannot occur. The example SUT has a constraint such that $p_2 \neq on$ if $p_3 = 1$, i. e., the combination of $p_2 = on$ and $p_3 = 1$ is not allowed.

More formally, a model of an SUT is defined as follows:

Definition 1 (SUT model)**.** *An* SUT model *is a triple* $\langle P, V, \phi \rangle$*, where*

- P *is a finite set of parameters* $p_1, \ldots, p_{|P|}$*,*
- V *is a family that assigns a finite value domain* V_i *for each parameter* p_i $(1 \leq i \leq |P|)$*, and*
- ϕ *is a constraint on parameter-value combinations.*

A *test case* is a value assignment for the parameters that satisfies the SUT constraint. For example, a 3-tuple (p_1 *=on,* p_2 *=on,* p_3 *=1*) is a test case for our example SUT model. We call a sequence of test cases a *test suite*.

An *exhaustive test suite* (i. e. *all combination test suite*) is a sequence of all possible test cases, i. e., a test suite that covers all parameter-value combinations satisfying the SUT constraint. In general, exhaustive testing is impractical since it stipulates to test all possible test cases and thus its size (the number of test cases) increases exponentially with the number of parameters.

Combinatorial *t-way testing* (e. g., *pairwise*, when $t = 2$) alternatively stipulates to test all *t-way* parameter-value combinations satisfying the SUT constraint at least once. We call t an *interaction strength*. An exhaustive test suite corresponds to a t-way test suite with $t = |P|$.

Definition 2 (t-way test suite)**.** *Let* $\langle P, V, \phi \rangle$ *be an SUT model. We say that a tuple of* t $(1 \leq t \leq |P|)$ *parameter-values is* possible *iff it does not contradict the SUT constraint* ϕ*. A* t-way test suite *for the SUT model is a test suite that covers all possible* t*-tuples of parameter-values in the SUT model.*

Example 1. *Consider the SUT model in Table I and* $t = 2$*. There exist* 15 *possible t-tuples (pairs) of parameter-values, as shown in Table II. The test suites* $\mathcal{T}_1$ *in Table III is a 2-way (pairwise) test suite since it covers all the possible parameter-value pairs in Table II.* $\mathcal{T}_2$ *in Table IV is a 3-way test suite and corresponds to the exhaustive test suite since the number of parameters in the example model is three .*

Many algorithms to efficiently construct t-way test suites have been proposed so far. Approaches to generate t-way test suites for SUT models with constraints include greedy algorithms (e. g., AETG [5], PICT [6], [31], and ACTS [3], [26]), heuristic search (e. g., CASA [9], HHSA [12], and TCA [17]), and SAT-based approaches (e. g., Calot [23], [24]).

B. Related Work: Effectiveness evaluation of t-way testing

The effectiveness of t-way testing with small interaction strength t on fault detection have been reported by several empirical studies so far [13], [15], but the code coverage of t-way testing has not been studied well. Table V summarizes the effectiveness metrics studied in related work.

Kuhn et al. [16] investigated parameter interactions inducing actual failures of four systems; a software for medical devices, a browser, a server, and a database system. As a result, 29–68% of the faults involved a single parameter; 70–97% (89–99%) of the faults involved up to two (three) parameter interactions; 96–100% of the faults involved up to four and five parameter

interactions; no fault involved more than six parameters. From the result, the authors concluded that most failures are triggered by parameter interactions with small t (at most four to six) and thus t-way testing with $4 \leq t \leq 6$ could provide the fault detection ability of "pseudo-exhaustive" testing.

Zhang et al. [25] also explored that failures of actual commercial MP3 players are triggered by t-way parameter interactions with at most $t = 4$.

Petke et al. [22] more thoroughly studied the efficiency of early fault detection by t-way testing with $2 \leq t \leq 6$. They used six projects, flex, make, grep, gzip, nanoxml, and siena, from the Software artifact Infrastructure Repository (SIR) and showed the number of faults detected after 25%, 50%, 70%, and 100% of test cases are executed.

Henard et al. [11] used five projects, grep, sed, flex, make, and gzip, also from SIR and compared the number of faults detected by test suite prioritization with t-way coverage ($2 \leq t \leq 4$) and other black-box and white-box prioritization.

Choi et al. [4] used three projects, flex, grep, and make, from SIR and investigated a correlation of the fault detection effectiveness with two evaluation metrics, called weight coverage and KL divergence, for prioritized t (= 2)-way testing.

To our knowledge, the only work by Giannakopoulou et al. [10] reported code coverage of t-way testing. Their target system is a component of the Tactical Separation Assisted Flight Environment (TSAFE) of the Next Generation Air Transportation System (NextGen) by the NASA Ames Research Center. In their work, t (= 3)-way testing and their model-checker (JPF [30]) based exhaustive testing are compared w. r. t. code coverage; line coverage, branch coverage, loop coverage, and strict condition coverage, which are computed using CodeCover [28].

Giannakopoulou et al. reported that for two program modules, the differences of code coverage by 3-way testing and exhaustive testing are 0–2% for the four coverage metrics they used, while the numbers of test cases are 6,047 for 3-way testing but 9.9×10^6 for exhaustive testing. In this paper, we more thoroughly analyze the code coverage effectiveness of t-way testing with $1 \leq t \leq 4$ using three open source utility programs.

III. Experiments

A. Subject Programs

To investigate code coverage of t-way testing, we use three open source projects of C programs, flex, grep, and make, from the Software artifact Infrastructure Repository (SIR) [32]. flex is a lexical analysis generator. grep is a program to search for text matching regular expressions. make is a program to control the compile and build process. The programs have been widely used to evaluate testing techniques by researchers in studies including [4], [11], [22]. Table VI shows for each version of programs we use, the version identifier, the year released, and the lines of code (LoC) calculated using cloc [27].

TABLE VI
Subject programs.

Proj.	Ver.	Identifier	Year of release	LoC
flex	v0	2.4.3	1993	10,163
	v1	2.4.7	1994	10,546
	v2	2.5.1	1995	13,000
	v3	2.5.2	1996	13,048
	v4	2.5.3	1996	13,142
	v5	2.5.4	1997	13,144
grep	v0	2.0	1996	8,163
	v1	2.2	1998	11,945
	v2	2.3	1999	12,681
	v3	2.4	1999	12,780
	v4	2.4.1	2000	13,280
	v5	2.4.2	2000	13,275
make	v0	3.75	1996	17,424
	v1	3.76.1	1997	18,525
	v2	3.77	1998	19,610
	v3	3.78.1	1999	20,401
	v4	3.79.1	2000	23,188

```
Parameters:
...
 Debug mode:    # -d
   Debug_on.
   Debug_off.

 Bypass use:    # -Cr
   Bypass_on.   [property Bypass]
   Bypass_off.

 Fast scanner:  # -f, -Cf
   FastScan.    [property FastScan]
   FullScan.    [if !Bypass][property FullScan]
   off.         [property f&Cfoff]
...
```

Fig. 1. A part of the test plan for flex in TSL.

B. Subject Test Suites

1) SUT Models: For each project, flex, grep, and make, we construct an SUT model for t-way testing whose parameters, values, and constraints are fully extracted from the test plan in TSL (Test Specification Language), which is included in SIR. For example, Figure 1 shows a part of the test plan in TSL for project flex. From the TSL specification, we construct the SUT model for flex whose parameters include Debug mode($= p_1$), Bypass use($= p_2$) and Fast scanner($= p_3$), p_2 has two values including Bypass_on($= on$), p_3 has three values including FullScan($= 2$), and constraints include $(p_3 = 2) \rightarrow (p_2 \neq on)$. Table I corresponds to a part of the SUT model for flex, which is constructed from the part of the test plan in Figure 1.

Table VII shows the size of the SUT model constructed for each project. In the table, the size of parameter-values is expressed as $k; g_1^{k_1} g_2^{k_2} \ldots g_n^{k_n}$, which indicates that the number of parameters is k and for each i there are k_i parameters that have g_i values. The size of constraints is expressed as $l; l_1^{h_1} l_2^{h_2} \ldots l_m^{h_m}$, which indicates that the constraint is described in conjunctive normal form (CNF) with l variables whose Boolean value

TABLE VII
CONSTRUCTED SUT MODELS.

Proj.		Model size
flex	Parameter-values	29; $3^{23}4^{4}6^{2}$
	Constraints	97; $2^{712}22^{1}24^{2}25^{17}26^{9}$
grep	Parameter-values	14; $2^{4}3^{1}4^{3}5^{1}6^{1}9^{1}11^{1}13^{1}20^{1}$
	Constraints	87; $2^{433}3^{27}4^{8}7^{5}16^{1}24^{1}27^{1}28^{1}31^{10}$
make	Parameter-values	22; $2^{2}3^{12}4^{4}5^{2}6^{1}7^{1}$
	Constraints	79; $2^{526}21^{1}22^{1}23^{1}24^{3}25^{7}26^{9}$

TABLE VIII
SIZES AND CODE COVERAGE OF EXHAUSTIVE TEST SUITES.

Proj.	Size		Line coverage	Branch coverage
flex	525	Avg.	0.7968	0.8544
		Min.	0.7789	0.8151
		Max.	0.8312	0.9316
grep	470	Avg.	0.4961	0.4948
		Min.	0.4726	0.4746
		Max.	0.5900	0.5826
make	793	Avg.	0.4543	0.5373
		Min.	0.4234	0.5126
		Max.	0.4726	0.5494

represents an assignment of a value to a parameter and for each j there are h_j clauses that have l_j literals. For the example SUT model in Figure 1, the size of parameter-values is $3; 2^2 3^1$ and the size of constraints is $2; 2^1$.

2) Test Suites: We use t-way test suites with $1 \leq t \leq 4$ that are generated by ACTS [3], [26] and PICT [6], [31] for our constructed SUT models with constraints. The tools ACTS and PICT are state-of-the-art open source t-way test generation tools developed by NIST (National Institute of Standards and Technology) and Microsoft, respectively. For comparison, we also use exhaustive test suites each of which obtains all possible test cases. The exhaustive test suite for the test plan of each project is included in SIR.

3) Evaluation Metrics: To evaluate code coverage of each test suite, we analyze the following two kinds of code coverage, which are computed using gcov [29]:

- Line coverage: the percentage of program lines executed.
- Branch coverage: the percentage of branches of conditional statements executed.

gcov is a source code analysis tool, which is a standard utility delivered with the GNU C/C++ Compiler and reports how many lines and branches are executed.

IV. RESULTS

Table VIII shows the size, i. e. the number of test cases, and the code coverage (line coverage and branch coverage) of the exhaustive test suite for each project, while Table IX and Table X show those of the subject t-way test suites ($1 \leq t \leq 4$) generated by ACTS and PICT. Table IX shows the sizes of the subject t-way test suites with the ratio of them over the sizes of exhaustive test suites. Table X summarizes line coverage and branch coverage of the subject t-way test suites for each project.

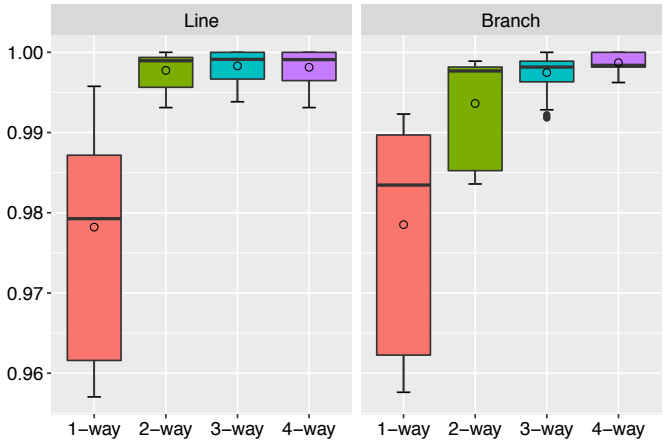


Fig. 2. Ratio of code coverage of t-way testing ($1 \leq t \leq 4$) over that of exhaustive testing for all versions of projects.

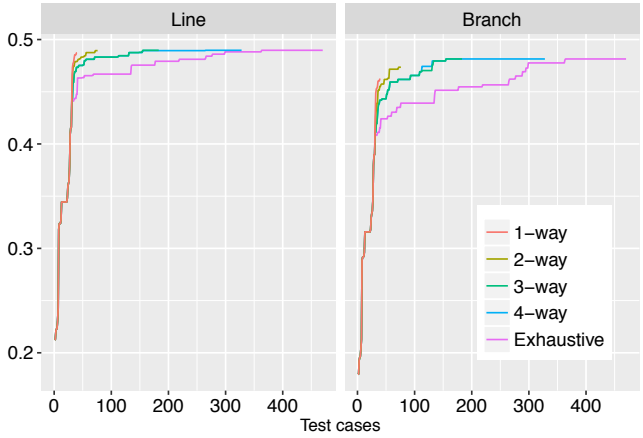


Fig. 3. Example code coverage growths of t-way testing ($1 \leq t \leq 4$) and exhaustive testing for one version (v1) of grep.

In Table VIII and Table X, we show the average, minimum, and maximum values of code coverage for versions of each project.

For example, for project flex, the sizes of 2-way test suites (52 by ACTS and 51 by PICT) are less than 10% of the size of the exhaustive test suite (525) from Table VIII and Table IX. On the other hand, for flex, line coverage and branch coverage of 2-way test suites (the exhaustive test suite) are on average 0.7927 (0.7968) and 0.8522 (0.8544) from Table VIII and Table X.

A. RQ1: t-way testing vs. exhaustive testing

To compare the code coverage between t-way testing and exhaustive testing, we investigate the following metric

$$R_{Cov}(T_t, EX) = Cov(T_t) \ / \ Cov(EX),$$

which denotes the ratio of code coverage of t-way test suite T_t over that of exhaustive test suite EX.

Table XI summarizes the values of $R_{Cov}(T_t, EX)$ with $1 \leq t \leq 4$ for line coverage and branch coverage for each project and

TABLE IX
SIZES OF t-WAY TEST SUITES ($1 \leq t \leq 4$).

Proj.		# of test cases (ratio over # of the exhaustive test cases)							
		1-way		2-way		3-way		4-way	
flex	ACTS	30	(5.71 %)	52	(9.90 %)	91	(17.33 %)	155	(29.52 %)
	PICT	30	(5.71 %)	51	(9.71 %)	90	(17.14 %)	154	(29.33 %)
grep	ACTS	40	(8.51 %)	76	(16.17 %)	183	(38.94 %)	328	(69.79 %)
	PICT	40	(8.51 %)	77	(16.38 %)	180	(38.30 %)	326	(69.36 %)
make	ACTS	27	(3.40 %)	34	(4.29 %)	44	(5.55 %)	68	(8.58 %)
	PICT	27	(3.40 %)	34	(4.29 %)	45	(5.67 %)	69	(8.70 %)

TABLE X
CODE COVERAGE OF t-WAY TEST SUITES ($1 \leq t \leq 4$).

Proj.		Line coverage				Branch coverage			
		1-way	2-way	3-way	4-way	1-way	2-way	3-way	4-way
flex	Avg.	0.7683	0.7927	0.7934	0.7931	0.8407	0.8522	0.8522	0.8522
	Min.	0.7481	0.7755	0.7763	0.7755	0.8018	0.8136	0.8136	0.8136
	Max.	0.8145	0.8264	0.8267	0.8267	0.9225	0.9281	0.9281	0.9281
grep	Avg.	0.4917	0.4959	0.4961	0.4961	0.4769	0.4880	0.4933	0.4948
	Min.	0.4668	0.4723	0.4726	0.4726	0.4549	0.4676	0.4712	0.4746
	Max.	0.5875	0.5897	0.5900	0.5900	0.5726	0.5786	0.5845	0.5826
make	Avg.	0.4451	0.4539	0.4540	0.4540	0.5321	0.5364	0.5366	0.5366
	Min.	0.4168	0.4230	0.4230	0.4230	0.5053	0.5117	0.5117	0.5117
	Max.	0.4628	0.4724	0.4724	0.4726	0.5442	0.5484	0.5484	0.5484

TABLE XI
COMPARISON OF CODE COVERAGE BETWEEN t-WAY TESTING ($1 \leq t \leq 4$) AND EXHAUSTIVE TESTING.

	Proj.	Line coverage				Branch coverage			
		1-way	2-way	3-way	4-way	1-way	2-way	3-way	4-way
Avg. of $R_{Cov}(T_t, EX)$ ($R = Cov(T_t)/Cov(EX)$)	flex	96.41 %	99.49 %	99.58 %	99.54 %	98.40 %	99.74 %	99.74 %	99.74 %
	grep	99.11 %	99.95 %	100.00 %	100.00 %	96.32 %	98.58 %	99.67 %	100.00 %
	make	97.97 %	99.91 %	99.93 %	99.92 %	99.03 %	99.84 %	99.88 %	99.87 %
	Avg.	97.82 %	99.77 %	99.83 %	99.81 %	97.85 %	99.36 %	99.76 %	99.87 %
# ($R \geq 99.5$ %) / # all cases	flex	0 / 12	8 / 12	8 / 12	8 / 12	0 / 12	12 / 12	12 / 12	12 / 12
	grep	4 / 12	12 / 12	12 / 12	12 / 12	0 / 12	0 / 12	7 / 12	12 / 12
	make	0 / 10	10 / 10	10 / 10	10 / 10	0 / 10	10 / 10	10 / 10	10 / 10
	Total	4 / 34	30 / 34	30 / 34	30 / 34	0 / 34	22 / 34	29 / 34	34 / 34

all projects. In the table, we also show the numbers of cases where $R_{Cov}(T_t, EX) \geq 99.5\%$, i. e. t-way testing achieves more than 99.5% of the coverage obtained by exhaustive testing, over the numbers of all cases (versions) for projects.

Figure 2 presents the box plots for the results of $R_{Cov}(T_t, EX)$ for all projects. Each box plot shows the mean (circle in the box), median (thick horizontal line), the first/third quartiles (hinges), and highest/lowest values within 1.5 × the inter-quartile range of the hinge (whiskers).

- How high code coverage can t-way testing achieve compared to exhaustive testing?

From Table XI and Figure 2, we can see that t-way testing with even small t can achieve high values of $R_{Cov}(T_t, EX)$, i. e. high ratios of code coverage over the code coverage of exhaustive testing.

In the result of our case study, 1-way (2-way) testing covers avg. 97.82% (99.77%) of line coverage of exhaustive testing and avg. 97.85% (99.36%) of branch coverage of exhaustive testing. With 3-way (4-way) testing, line coverage is avg. 99.83% (99.81%) and branch coverage is avg. 99.76% (99.87%) of the coverage of exhaustive testing.

- Can t-way testing obtain higher code coverage earlier compared to exhaustive testing?

Figure 3 shows example line coverage growths and branch coverage growths of t-way test suites ($1 \leq t \leq 4$) and the exhaustive test suites for one version (v1) of project grep. (The coverage growths represent the typical cases of our experiment results.) We can see that t-way testing with smaller t obtains higher code coverage earlier compared to exhaustive testing and t-way testing with larger t.

For the example case in Figure 3, to obtain 48% line coverage (46% branch coverage), 1-way, 2-way, 3-way, and 4-way testing respectively require 35, 42, 56, and 56 (36, 47, 71, and 71) test cases, while exhaustive testing requires 219 (265) test cases.

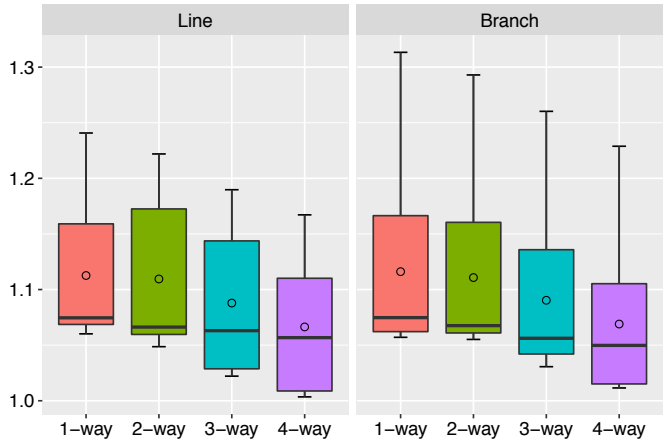


Fig. 4. Ratios of code coverage of t-way testing ($1 \leq t \leq 4$) over that of exhaustive testing with the same sizes for all versions of projects.

- How large t is necessary for t-way testing to achieve the code coverage close to that by exhaustive testing?

Surprisingly, as the result of our case study, all t-way test suites with $1 \leq t \leq 4$ obtain more than 95% of code coverage of exhaustive test suites. Especially, for project grep, 4-way test suites obtain the same line coverage and branch coverage with the exhaustive test suite. As described in Table XI, in 30 cases of all 34 cases, 2-way, 3-way, and 4-way test suites achieve more than 99.5% of line coverage of exhaustive test suites. For branch coverage, in all cases, 4-way test suites achieve more than 99.5% of coverage of exhaustive test suites.

> *From the results, t-way testing with small t ($1 \leq t \leq 4$) can efficiently obtain code coverage close to that by exhaustive testing while requiring smaller test cases.*

B. RQ2: t-way testing vs. exhaustive testing in the same sizes

To compare the code coverage between t-way testing and exhaustive testing with the same sizes, we investigate the following metric

$$R_{Cov}(T_t, EX*) = Cov(T_t) \ / \ Cov(EX*),$$

which denotes the ratio of code coverage of t-way test suite T_t over that of a subset, hereafter denoted by $EX*$, of exhaustive test suite EX whose size is same with T_t. In our experiments, we constructed $EX*$ 100 times by randomly selecting $|T_t|$ test cases from exhaustive test suite EX and use the average value of the code coverage for the 100 $EX*$.

Table XII summarizes the values of $R_{Cov}(T_t, EX*)$ with $1 \leq t \leq 4$ for line coverage and branch coverage for each project and all projects. In the table, we also show the numbers of cases where $R_{Cov}(T_t, EX*) \geq 105\%$, i. e. t-way testing achieves more than 105% of the coverage obtained by exhaustive testing with the same size, over the numbers of all cases for projects. Figure 4 presents the box plots for the results of $R_{Cov}(T_t, EX*)$ for all projects.

- With the same number of test cases, how different are t-way testing and exhaustive testing on code coverage?

From Table XII and Figure 4, we can see that t-way test suites with $1 \leq t \leq 4$ achieve higher line coverage and branch coverage compared to exhaustive test suites in the same sizes. Especially, t-way testing with smaller t obtains higher values of $R_{Cov}(T_t, EX*)$, i. e. higher ratios of code coverage over that of exhaustive testing in the same size.

As described in Table XII, for all cases, 1-way and 2-way testing achieve more than 105% of code coverage of exhaustive testing with the same size. For 3-way and 4-way testing, the numbers of cases that achieve more than 105% of line (branch) coverage of exhaustive testing with the same sizes are 24 and 20 (24 and 16) cases among all 34 cases.

> *From the results, t-way testing with smaller t can obtain higher code coverage compared to exhaustive testing with the same number of test cases.*

V. Conclusion

This paper analyzes the code coverage effectiveness of combinatorial t-way testing with small t. As a result of our empirical evaluation using a collection of open source utility programs, t-way testing with small t ($1 \leq t \leq 4$) efficiently covers more than 95% of code coverage achieved by exhaustive testing, while requiring much smaller test cases. In addition, comparing in the same test suite sizes, t-way testing with smaller t obtains higher ratio of code coverage over that by exhaustive testing.

In this paper, we evaluate two kinds of widely used code coverage metrics, line coverage and branch coverage. Further work includes evaluating other metrics such as loop coverage, condition coverage, etc. Another further work is to investigate both the code coverage effectiveness and the fault detection effectiveness of t-way testing and analyze the relation between them on real software projects.

Acknowledgments

The authors would like to thank anonymous referees for their helpful comments and suggestions to improve this paper. This work was partly supported by JSPS KAKENHI Grant Number 16K12415.

References

[1] International Standardization Organization, ISO26262: Road vehicles - Functional safety, November 2011.

[2] K. Z. Bell and M. A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. In *Proc. of International Conference on Information and Communication Technology*, pages 221–235. IEEE, 2005.

[3] M. N. Borazjany, L. Yu, Y. Lei, R. Kacker, and R. Kuhn. Combinatorial testing of ACTS: A case study. In *Proc. of the 5th International Conference on Software Testing, Verification and Validation (ICST)*, pages 591–600. IEEE, 2012.

[4] E. Choi, S. Kawabata, O. Mizuno, C. Artho, and T. Kitamura. Test effectiveness evaluation of prioritized combinatorial testing: a case study. In *Proc. of the International Conference on Software Quality, Reliability & Security (QRS)*, pages 61–68. IEEE, 2016.

TABLE XII
COMPARISON OF CODE COVERAGE BETWEEN t-WAY TESTING ($1 \leq t \leq 4$) AND EXHAUSTIVE TESTING WITH THE SAME SIZES.

	Proj.	Line coverage				Branch coverage			
		1-way	2-way	3-way	4-way	1-way	2-way	3-way	4-way
Avg. of $R_{Cov}(T_t, EX*)$ ($R* = Cov(T_t)/Cov(EX*)$)	flex	116.45 %	117.26 %	114.45 %	111.04 %	116.65 %	116.12 %	113.65 %	110.57 %
	grep	109.93 %	108.39 %	105.36 %	103.31 %	111.26 %	110.44 %	107.36 %	104.94 %
	make	106.63 %	106.45 %	106.11 %	105.36 %	105.99 %	105.80 %	105.51 %	104.85 %
	Avg.	111.26 %	110.95 %	108.79 %	106.64 %	111.61 %	111.08 %	109.04 %	106.90 %
# ($R* \geq 105$ %) / # all cases	flex	12 / 12	12 / 12	12 / 12	12 / 12	12 / 12	12 / 12	12 / 12	12 / 12
	grep	12 / 12	12 / 12	2 / 12	2 / 12	12 / 12	12 / 12	2 / 12	2 / 12
	make	10 / 10	10 / 10	10 / 10	6 / 10	10 / 10	10 / 10	10 / 10	2 / 10
	Total	34 / 34	34 / 34	24 / 34	20 / 34	34 / 34	34 / 34	24 / 34	16 / 34

[5] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Trans. Software Eng.*, 34(5):633–650, 2008.

[6] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test-case senarios. In *Proc. of the 24th Pacific Northwest Software Quality Conference*, pages 419–430. Citeseer, 2006.

[7] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[8] G. B. Finelli. NASA software failure characterization experiments. *Reliability Engineering & System Safety*, 32(1):155–169, 1991.

[9] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, 2011.

[10] D. Giannakopoulou, D. Bushnell, J. Schumann, H. Erzberger, and K. Heere. Formal testing for separation assurance. *Annals of Mathematics and Artificial Intelligence*, 63(1):5–30, 2011.

[11] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. Comparing white-box and black-box test prioritization. In *Proc. of the 38th International Conference on Software Engineering (ICSE)*, pages 523–534. ACM, 2016.

[12] Y. Jia, M. B. Cohen, M. Harman, and J. Petke. Learning combinatorial interaction testing strategies using hyperheuristic search. In *Proc. of the 37th International Conference on Software Engineering (ICSE)*, pages 540–550. IEEE/ACM, 2015.

[13] R. N. Kacker, D. R. Kuhn, Y. Lei, and J. F. Lawrence. Combinatorial testing for software: An adaptation of design of experiments. *Measurement*, 46(9):3745–3752, 2013.

[14] T. Kitamura, A. Yamada, G. Hatayama, C. Artho, E. Choi, N. T. B. Do, Y. Oiwa, and S. Sakuragi. Combinatorial testing for tree-structured test models with constraints. In *Proc. of the International Conference on Software Quality, Reliability & Security (QRS)*, pages 141–150. IEEE, 2015.

[15] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Introduction to combinatorial testing*. CRC Press, 2013.

[16] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng.*, 30(6):418–421, 2004.

[17] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang. TCA: An efficient two-mode meta-heuristic algorithm for combinatorial test generation. In *Proc. of the 30th International Conference on Automated Software Engineering (ASE)*, pages 494–505. ACM/IEEE, 2015.

[18] C. Maughan. Test case generation using combinatorial based coverage for rich web applications. *PhD Thesis. Utah State University*, 2012.

[19] C. Montanez, D. R. Kuhn, M. Brady, R. M. Rivello, J. Reyes, and M. K. Powers. An application of combinatorial methods to conformance testing for document object model events. *NISTIR-7773*, 2010.

[20] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11, 2011.

[21] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Commun. ACM*, 31(6):676–686, 1988.

[22] J. Petke, M. Cohen, M. Harman, and S. Yoo. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Trans. Software Eng.*, 41(9):901–924, 2015.

[23] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E. Choi. Greedy combinatorial test case generation using unsatisfiable cores. In *Proc. of the 31st International Conference on Automated Software Engineering (ASE)*, pages 614–624. IEEE/ACM, 2016.

[24] A. Yamada, T. Kitamura, C. Artho, E. Choi, Y. Oiwa, and A. Biere. Optimization of combinatorial testing by incremental SAT solving. In *Proc. of the 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[25] Z. Zhang, X. Liu, and J. Zhang. Combinatorial testing on id3v2 tags of mp3 files. In *Proc. of the 5th International Conference on Software Testing, Verification and Validation (ICST)*, pages 587–590. IEEE, 2012.

[26] ACTS, Available: http://csrc.nist.gov/groups/SNS/acts/.

[27] cloc – Count Lines of Code, Available: http://cloc.sourceforge.net.

[28] CodeCover – an open-source glass-box testing tool, Available: http://codecover.org.

[29] gcov – a test coverage program, Available: http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[30] JavaPathfinder, Available: http://babelfish.arc.nasa.gov/trac/jpf.

[31] Pairwise Independent Combinatorial Tool, Available: http://github.com/Microsoft/pict.

[32] Software-artifact Infrastructure Repository, Available: http://sir.unl.edu/.

Sustainability Profiling of Long-living Software Systems

Ahmed D. Alharthi, Maria Spichkova and Margaret Hamilton
RMIT University, Melbourne, Australia
Email: {ahmed.alharthi, maria.spichkova,margaret.hamilton}@rmit.edu.au

***Abstract*—This paper introduces a framework for software sustainability profiling. The goal of the framework is to analyse sustainability requirements for long-living software systems, focusing on usability and readability of the sustainability profiles. To achieve this goal, we apply a quantitate approach such as fuzzy rating scale-based questionnaires to rank the sustainability requirements, and the Technique for Order Preference by Similarity to Ideal Solution (TOPSIS) to analyse the results of questionnaires and to provide a basis for system profiling.**

The core profiling elements provided by our framework are (1) a sustainability five-star rating, (2) visualisation of the five sustainability dimensions as a pentagon graph detailing combination for individual, social, technical, economic and environmental dimensions, and (3) a bar graph of overall sustainability level for each requirement. To ensure sustainability, the proposed profiling framework covers the five dimensions of sustainability to quantify the sustainability of any software system not only during the requirement gathering phase but also during maintenance phase of software system lifecycle.

I. Introduction

Addressing the impacts of software systems on sustainability is a first-class quality concern beside usability, safety and security [1]. A number of studies showed that if a software system is developed without taking sustainability requirements into account, the system could have negative impacts on individual, social, technology, economic, and environment sustainability, cf. [2]–[5]. Environmental awareness is crucial for software engineering, especially in the case of large-scale systems having many thousands of users.

The analysis of system sustainability has to be initiated on the requirements engineering (RE) phase [6], [7]. Based on this idea, Becker et al. [8] emphasised that the importance of identifying stakeholders whose outside interests are affected, and the use of long-life scenarios techniques during requirements elicitation could forecast potential impacts. Duffy [9] highlighted that sustainability could be achieved especially in the social dimension through usability, which is a non-functional requirement, and its traditional methodologies.

This question is especially important for long-living systems, where the stakeholders requirements and preferences might change over the time the system is in use. For example, a system that can be seen as sustainable today, might be rated as environmentally unsustainable in few years, while new techniques to increase environmental sustainability are developed. To solve this problem, we require an easy-to-use profiling framework based on quantitate approaches that would allow to analyse the up-to-date system sustainability profiles, based on system characteristics and the up-to-date ratings (quotations) of the corresponding requirements. Usability and readability of the approach is crucial to make it applicable for real software development processes, as the quotation process and the generated profiles have to be easy-to-use by all stakeholders.

Contributions: To ensure the sustainability of long-living software systems over their eitire live-cycle, we propose a framework for sustainability profiling. The framework allows to analyse sustainability requirements for long-living software systems. The up-to-date profiles could be generated both during the RE and the maintenance phase of the software system lifecycle. The framework workflow is presented in Figure 1. First of all, stakeholders are assigned to a group to rate requirements from the different perspective of sustainability dimensions (individual, social, technical, economic and environmental). Then, a fuzzy rating scale is used to avoid imprecision for answering quantitative questionnaires [10]. As the next step, the Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS, cf. [11]) is utilised to find alternatives that are the nearest distance from the positive ideal solution and farthest distance from the negative ideal solution. The software sustainability profiling includes an overall picture of how sustainable a software system really is. The profile is presented as three core elements: (1) a five-star rating, (2) five dimensions of sustainability in a pentagon graph, and (4) an overall measure of sustainability for each requirement in a bar graph.

Outline: The rest of the paper is organised as follows. In Section II we discuss the background and related work. Section III introduces our framework for software sustainability profiling. Section IV introduces an example scenario to show how the framework can be used to profile software systems. Section V summarises the core contributions of our work.

II. Background and Related Work

In this section we discuss the research directions and approaches that provide a background for our framework: RE for sustainable systems, the idea of the sustainability profiling, quantitative approaches, approaches using the fuzzy rating scale, and the TOPSIS framework for requirements analysis. We selected TOPSIS for our sustainability profiling framework, as this technique has been successfully used for prioritising requirements and solving conflict among non-functional requirements, cf. [11]–[13]. Previously, TOPSIS was used

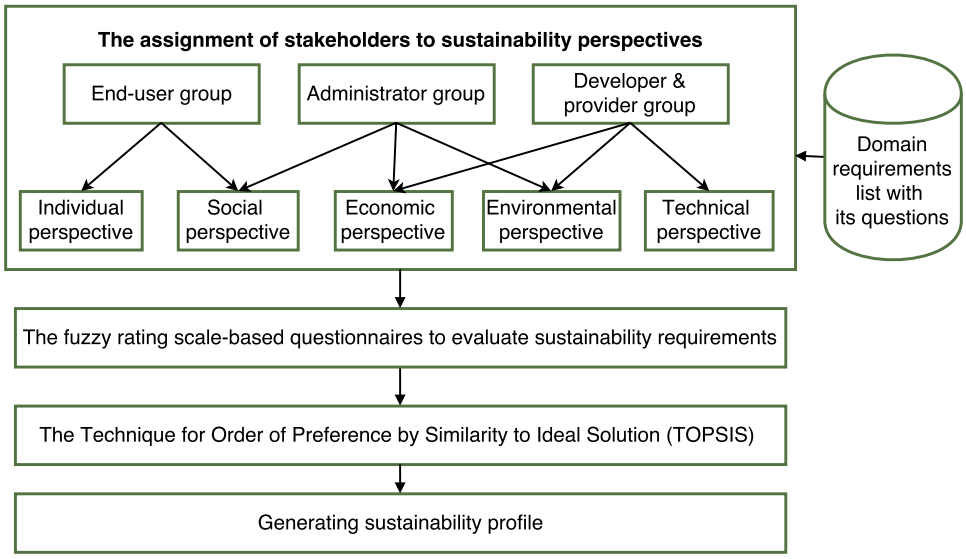


Fig. 1. Software Sustainability Profiling Framework

without taking into account the sustainability aspects, but the extension to evaluate sustainability requirements is possible and easy to implements. In the sustainability dimensions we have the same kind of relations among requirements: (1) each requirement has impacts on other requirements, and (2) each requirement has positive or negative impacts on sustainability dimensions that could be maximised or minimised during the TOPSIS procedure.

A. *Requirements engineering for sustainable systems*

The RE phase of software development focuses on discovering, developing, tracing, analysing, qualifying, communicating and managing system requirements, cf. e.g., [14]. Lami et al. [15] proposed to define a sustainable software process as one which meets realistic sustainability objectives, taking into account not only direct but also indirect impacts of the software on economy, society, human beings, and environment.

Penzenstadler [16] defined RE for sustainability as *"the concept of using requirements engineering and sustainable development techniques to improve the environmental, social and economic sustainability of software systems and their direct and indirect effects on the surrounding business and operational context"*.

Sustainability in software has various dimensions. Goodland [17] suggested to distinguish the following four dimensions: *human (individual), social, economic and environmental sustainability*. Penzenstadler and Femmer [5] as well as Razavian et al. [18] added to the new dimension of *technical sustainability*.

In our framework, we analyse the system sustainability using the five dimensions:

- **Individual sustainability**: Individual needs should be protected and supported with dignity and in a way that developments should improve the quality of human life and not threaten human beings;
- **Social sustainability**: Relationships of people within society should be equitable, diverse, connected and democratic;
- **Technical sustainability**: Technology must cope with changes and evolution in a fair manner, respecting natural resources;
- **Environmental sustainability**: Natural resources have to be protected from human needs and wastes; and
- **Economic sustainability**: A positive economic value and capital should be ensured and preserved.

B. *Sustainability Profiling*

Sustainability profiling has been used mostly for software energy and data centre consumption, as well as in cities and urban settlements. James [19] highlighted that a holistic and integrated understanding of urban life is essential. He presented an urban profile framework for cities sustainability including four main domains ecology, economics, politics and culture as well as seven sub-domains for each main domain. The framework was also applied to the sustainability of eLearning by Stewart and Khare [20]. This framework was providing a nine-point scale rating that is imprecise and has to be extended to fit software development process and to cover the corresponding sustainability dimensions.

Gmach et al. [21] proposed a profiling approach for the sustainability of data centres, to quantify energy during design and operation of data centres. Similarly, Jagroep et al. [22] demonstrated a software energy profiling to analyse software changes in energy consumption between releases of a software product. Although both studies focused on energy consumption that could impact environmental and economic dimensions of sustainability, individual and social dimensions were ignored in the measurement. Our approach covers the five dimensions of sustainability to quantify the sustainability

of any software system, starting from the requirements phase and continuing over the phase of maintenance.

C. Quantitative Approach

Quantitative approaches are used to analyse data and to measure qualities in software engineering [23], [24]. For instance, goal-oriented requirements and user experience are analysed and measured via quantitative techniques having a rating scale of probability between satisfaction and denial of satisfaction. The rating scales and data analysis techniques vary from one quantitative approach to another. Some approaches use a five-level Likert scale while others employ a nine-point scale to present people's attitudes by scaling their responses. Notably, the Likert rating scales and the nine scales that are giving a number of options are closed format. For example, if a questionnaire has a closed five Likert scale, participants can only express their opinion through one of the five choices. These closed format options are imprecise, difficult to choose between and limited. A solution to overcome drawbacks of closed formatted scales are the fuzzy rating scale [10], cf. Section II-D for more details.

The quantitate approaches can be applied to several types of data, and the type of data to analyse might influence the choice of the approach. Tullis and Albert [23] suggest to distinguish the following four types of data:

- **Nominal data** is categorised or classification data, which it is not in any particular order, e.g., gender or hair colour;
- **Ordinal data** is ordered classified data, but the differences between them are not meaningful, e.g., product and movie ratings;
- **Interval data** is classified data where the difference between two data items is meaningful, but without natural zero points, e.g., temperature units;
- **Ratio data** is interval data with absolute zero, e.g., weight and height.

To analyse sustainability requirements, we will create from the provided by stakeholders ranking the corresponding *ratio data*. This transformation will be done using TOPSIS, cf. Section II-E. The ratio data will be then further explored to build the system profile.

D. The Fuzzy Rating Scale

A fuzzy rating scale (FRS) allows the capturing of the diversity of individual responses in questionnaires, also avoiding imprecision while rating a questionnaire [10]. For our sustainability profiling, stakeholders will be required to rate the corresponding sustainability dimensions. For example, as an alternative of stakeholders choice from a five classified rating scale, they can select their range and extend it between a range of two extreme poles.

To implement an FRS, we adopt the fuzzy rating scale method proposed by Lubiano et al. [25]:

Step 1 Considering a representative rating on the bounded interval;

Step 2 Determining a core response to be considered as *fully compatible*;

Step 3 Determining a support response to be considered as *compatible to some extent*; and

Step 4 Creating a trapezoidal fuzzy number from the two intervals, which are *linearly interpolated*, as $Tra(a, b, c, d)$, where $0 \leq a \leq b \leq c \leq d \leq 1$.

Figure 2 presents an example on application the above method to within our framework: The scale goes from 0 to 100%, where 0 corresponds to the worst case (critical value), and 100 corresponds to the best case (green value). For simplicity, it is also possible to use a scale from 0 to 1, where 1 corresponds to 100%.

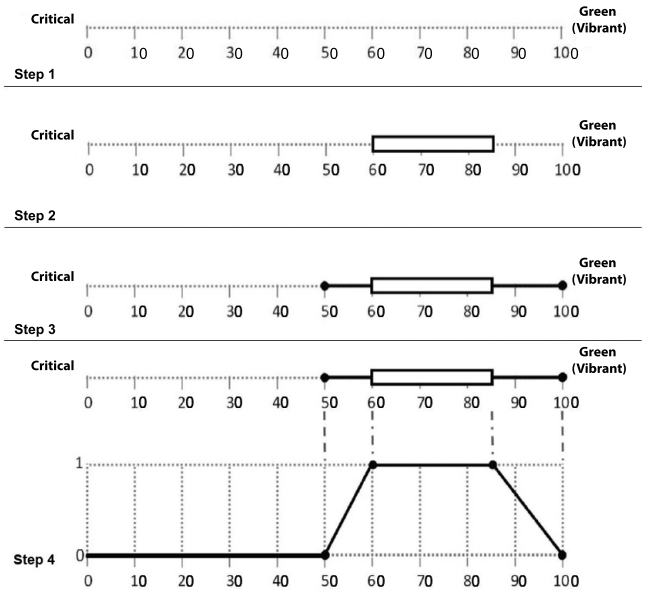


Fig. 2. Fuzzy Rating Scale for Sustainability Profiling

E. TOPSIS

Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS) is an effective technique to evaluate sustainability requirements which change over time is utilising. TOPSIS is one of the multiple criteria decision analysis approaches to identify the best alternative that is nearest to an ideal solution and farthest from negative ideal solution [12]. The principles of TOPSIS are simple, and positive ideal solutions and negative ideal solutions formed [26]. The benefit criteria in the positive ideal solution are maximised, and the cost criteria are minimised, while the cost criteria in the negative ideal solution are maximised, and the benefit criteria are minimised [11].

The following is the stepwise procedure of TOPSIS according to Behzadian [11]:

Step 1 Construct normalised decision matrix r_{ij}

$$r_{ij} = \frac{x_{ij}}{\sqrt{\sum_{i=1}^{m} x_{ij}^2}}, \quad for\ i = 1, \cdots, m, \quad j = 1, \cdots, n \tag{1}$$

Step 2 Construct the weighted normalised decision matrix v_{ij}

$$v_{ij} = w_i r_{ij} \tag{2}$$

where w_i is the weight for j criterion.

Step 3 Determine the positive ideal (A^*) and the negative ideal solutions ($A^{'}$):
Positive ideal solutions

$$A^* = \{max(v_{ij})|j \in J; min(v_{ij})|j \in J^{'}\} = \{v_1^*, \cdots, v_n^*\} \tag{3}$$

Negative ideal solutions

$$A^{'} = \{min(v_{ij})|j \in J; max(v_{ij})|j \in J^{'}\} = \{v_1^{'}, \cdots, v_n^{'}\} \tag{4}$$

Step 4 Calculate the separation measures:
The separation from positive ideal is

$$S^* = \sqrt{\sum_{j=1}^{n}(v_{ij} - v_i^*)^2}, \quad i = \{1, \cdots, m\} \tag{5}$$

Similarly, the separation from negative ideal is

$$S^{'} = \sqrt{\sum_{j=1}^{n}(v_{ij} - v_i^{'})^2}, \quad i = \{1, \cdots, m\} \tag{6}$$

Step 5 Calculate the relative closeness to the ideal solution C_i^*

$$C_i^* = \frac{S^{'}}{(S^* + S^{'})}, \quad 0 < C_i^* < 1, \quad i = \{1, \cdots, m\} \tag{7}$$

$C_i^* = 1$ if A_i solution has the best condition,
$C_i^* = 0$ if A_i solution has the worst condition.

III. Framework for Sustainability Profiling

The general idea of the framework workflow is presented in Figure 1. To measure the sustainability aspects of the requirements, we adopted the FRS approach. Requirements are rated against sustainability dimensions, which gives an input to the TOPSIS procedure. The provided by TOPSIS results will create a basis for sustainability profiling: using these results, our framework determines (1) the sustainable of each system requirement, (2) sustainability of the software system as whole. This will be presented in a five-star rating within each level of sustainability dimensions and the overall sustainability of each requirement. The analytical approach consists of the following five steps, cf. also Figure 1.

A. Assigning Stakeholders

Requirements engineers should assign stakeholders to one of the three stakeholder groups having end-users, administrators, and developers and providers groups. For instance, in eLearning systems the learner and instructor are in the end-users group while ITs support could be assigned to the administrator group.

B. Defining Questions

The framework will generate a questionnaire including related questions (instructions) for each requirement with regard to the sustainability dimensions and stakeholders groups. Thus, for each requirement k questions will be created, where $1 \leq k \leq 5$. Each question should present a single sustainability dimension perspective, which is covered by the requirement, and have a form
Rate the influence of the requirement on the X sustainability, where X is belongs to the set
$\{individual, social, technical, environmental, economic\}$.
The generated questionnaire can be further revised and adapted by both requirements engineers and sustainability experts, before continuing with the next step.

For example, requirement R1 has to have five questions, covering each dimension of the sustainability.

C. Rating Requirements

Each stakeholder has to answer allotted question from vary views of certain sustainability dimension by using the FRS. For example, stakeholders, who are in the learners and instructors group, will answer two questions for each requirement: from the individual and from the social sustainability point of view. and another time for the social sustainability. Each answer, also, will be in a form of trapezoidal fuzzy number from the two intervals as $Tra(a, b, c, d)$, where $0 \leq a \leq b \leq c \leq d \leq 1$.

D. Analysing Sustainability Using TOPSIS

After all stakeholders answered the questionnaire, the results of the FRS approach become inputs for TOPSIS. The data will be normalised and weighted according to Equations 1 and 2, and after that the steps 3, 4 and 5 of TOPSIS need to be applied twice:

- **First round**: Applying requirements as criteria to determine overall sustainability within the separation of requirements' impacts for each requirement; and
- **Second round**: Applying sustainability dimensions as criteria to analysis each dimension within all requirements and overall sustainability rating for the software.

E. Generating Software Sustainability Profiling

The result of TOPSIS analysis including two rounds helps to generate software sustainability profiling which is visualised representing the result. The profiling includes:

- **Sustainability five-star rating** Presenting the average of $\sum C^*$ in the both rounds of sustainability dimensions and requirements;
- **Five sustainability dimensions** Illustrating each dimension level combined in pentagon or bar graph (optional) for the software having all rated requirements; and
- **Bar graph** Showing an overall sustainability for each requirement.

An example of a sustainability profile for a software system, which is created using the proposed framework, is presented in the next section (cf. Figure 5).

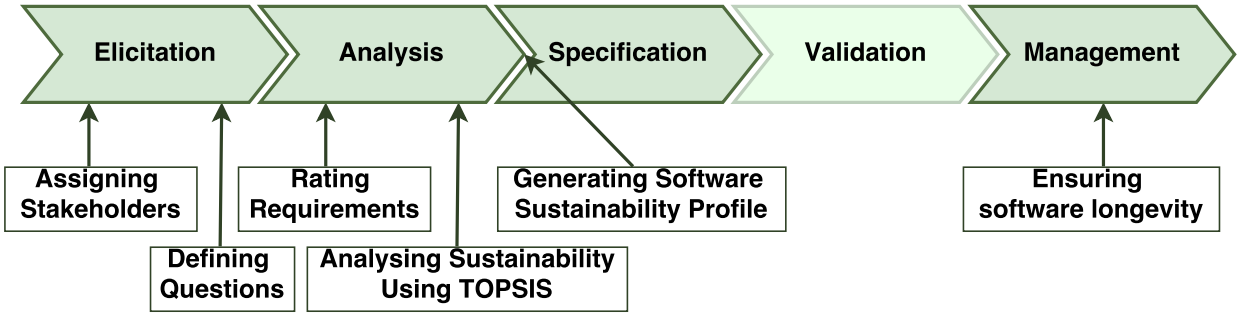


Fig. 3. Sustainability Profiling as a part of RE Activities

TABLE I
THE KEY CHART IN SOFTWARE SUSTAINABILITY PROFILING

Percentage %	Colour Code	Description
80-100	Dark green	Green (Vibrant)
60-79	Light green	Satisfactory
40-59	Yellow	Basic
20-39	Orange	Unsatisfactory
0-19	Red	Critical

Considering a different information in the profiling, we simplify and visualise the result by creating a key chart with five categories as shown in Table I. This key chart includes numeric variables in percentage, colour codes for visualisation, and linguistic variables as a description.

Figure 3 demonstrates how the proposed framework can be uses during the RE activities (we follow the definition of the RE activities introduced by [27]–[30]):

- **Requirements elicitation** is the practice of understanding and determining stakeholders' needs and constraints.
  To rate the sustainability requirements using the proposed framework, at this phase two actions are necessary:
  (A) the stakeholders have to be assigned, (B) the questionnaires have to be generated.
  However, taking into account the long-living nature of the system, re-iteration of these steps might be necessary on the management phase, to ensure the sustainability over the software system lifecycle: (A') new stakeholders can be assigned, (B') the questionnaires can be updated.
- **Requirements analysis** is the practice of refining stakeholders' needs and constraints by defining the process, data and object of the required system.
  On this phase, we conduct the following steps of our framework:
  (C) the stakeholders rate the requirements, (D) the sustainability of the system is analyses using TOPSIS, (E) the sustainability profile is generated.
  To ensure longevity of the system, these steps also can be repeated during the management phase.
- **Requirements specification** is the practice of writing down stakeholders' needs and constraints, and this documentation should be unambiguous, complete, correct, understandable, consistent, concise, and feasible.
  The sustainability profile could be seen as one of the input to the specification phase.
- **Requirements validation** is the practice of checking that the specification captures users' needs and constraints. The proposed framework does not cover the validation activities, which might be one of the future work directions.
- **Requirements management** is the practice of scheduling, controlling changes and tracking requirements over time. In the case of long-living systems, the management activities are crucial to keep the software system sustainable. The steps $(A) - (E)$ have to be repeated to provide an up-to-date sustainability profile of the system.

IV. APPLICATION OF THE PROPOSED FRAMEWORK

Let us discuss an example scenario with five requirements R1, . . . , R5. The aim of this scenario is to illustrate application of the proposed framework, without going into the technical details like generating of questions within real questionnaires. In this scenario, we will go through all framework steps and present the created sustainability profile as the final result.

A. Assigning Stakeholders

Let us assume that the requirements will be rated by ten assigned stakeholders: four in the end-users group, three in administrators group, and three in developers and providers group.

B. Defining Questions

This step is omitted in the example, as the rating activities will be simulated.

C. Rating Requirements

To simulate the rating activities where each stakeholder rates requirements against sustainability dimensions by answering defined questions, we generate random numbers between [0:1] (0 corresponds to a critical value, 1 corresponds to a green value) for the fuzzy rating scales. Figure 4 shows the results of application of the FRS approach to the requirement R1, from the prospective of ten assigned stakeholders.

As follows from Figure 4, Stakeholder S2, who is assigned to individual and social sustainability dimensions, rates R1 for

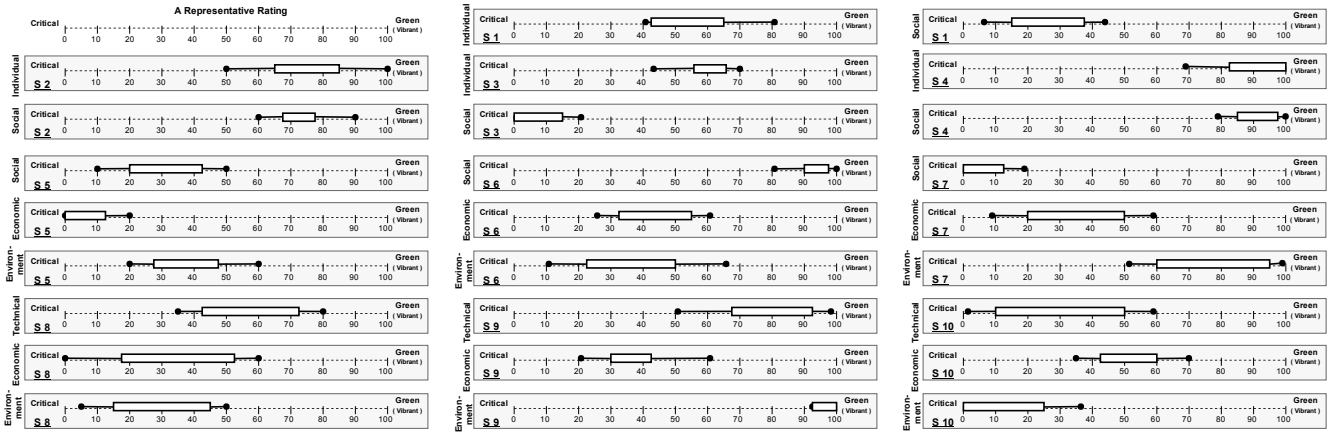


Fig. 4. Example of Fuzzy Rating Scale for Requirement (R1)

TABLE II
OUTPUT EXAMPLES OF FUZZY RATING SCALE FOR REQUIREMENTS ANALYSIS

	Dimension	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
R1	Individual	0.573	0.754	0.625	0.914						
	Social	0.276	0.727	0.087	0.917	0.377	0.942	0.066			
	Technical								0.579	0.808	0.324
	Economic					0.158	0.446	0.340	0.345	0.362	0.529
	Environment				0.382	0.351	0.799	0.291	0.986	0.130	
R2	Individual	0.281	0.472	0.232	0.289						
	Social	0.096	0.587	0.605	0.301	0.660	0.455	0.407			
	Technical								0.925	0.677	0.309
	Economic					0.093	0.506	0.738	0.567	0.459	0.395
	Environment				0.224	0.794	0.781	0.362	0.642	0.018	
R3	Individual	0.966	0.379	0.974	0.509						
	Social	0.030	0.331	0.170	0.717	0.835	0.128	0.909			
	Technical								0.173	0.157	0.728
	Economic					0.182	0.001	0.473	0.050	0.366	0.504
	Environment				0.257	0.282	0.187	0.814	0.711	0.688	
R4	Individual	0.287	0.802	0.347	0.361						
	Social	0.012	0.376	0.318	0.976	0.785	0.381	0.808			
	Technical								0.583	0.667	0.320
	Economic					0.163	0.417	0.547	0.599	0.360	0.821
	Environment				0.244	0.871	0.953	0.013	0.222	0.249	
R5	Individual	0.619	0.546	0.957	0.614						
	Social	0.600	0.005	0.460	0.003	0.977	0.535	0.518			
	Technical								0.215	0.995	0.943
	Economic					0.244	0.072	0.328	0.251	0.349	0.610
	Environment				0.214	0.704	0.662	0.949	0.714	0.583	

individual perspective as $Tra(0.51, 0.66, 0.856, 1.00)$ while social perspective as $Tra(0.60, 0.66, 0.75, 0.9)$. We calculate fuzzy values from each fuzzy rating by mean measurement, so individual and social means of R1 for S2 are 0.754 and 0.727, respectively.

D. Analysing Sustainability

In the next step, all the FRS outputs become inputs for TOPSIS, cf. Table II. These data are normalised according to Equation 1 for the five system requirements R1, . . . , R5 within the individual, social, technical, economic and environmental dimensions of sustainability. The result of normalisation step presented in Table III.

The weighted normalisation that was constructed according to Equation 2 is showed in Table IV. Following the TOPSIS procedure, we calculate for both rounds the separation measures from positive ideal S^* and negative ideal solutions $S^{'}$, as well as the relative closeness C^*. The results are summarised in Tables V and VI. Noteworthy, we could calculate the negative impact of economic and environmental sustainability dimensions via the negative ideal solution that maximises the cost criteria and minimises the benefit criteria.

E. Sustainability Profiling

The generating software sustainability profiling is presented in Figure 5 within an overall of 49% sustainability which is the mean of $\sum C^*$ in the two rounds (in Table V and VI). Among

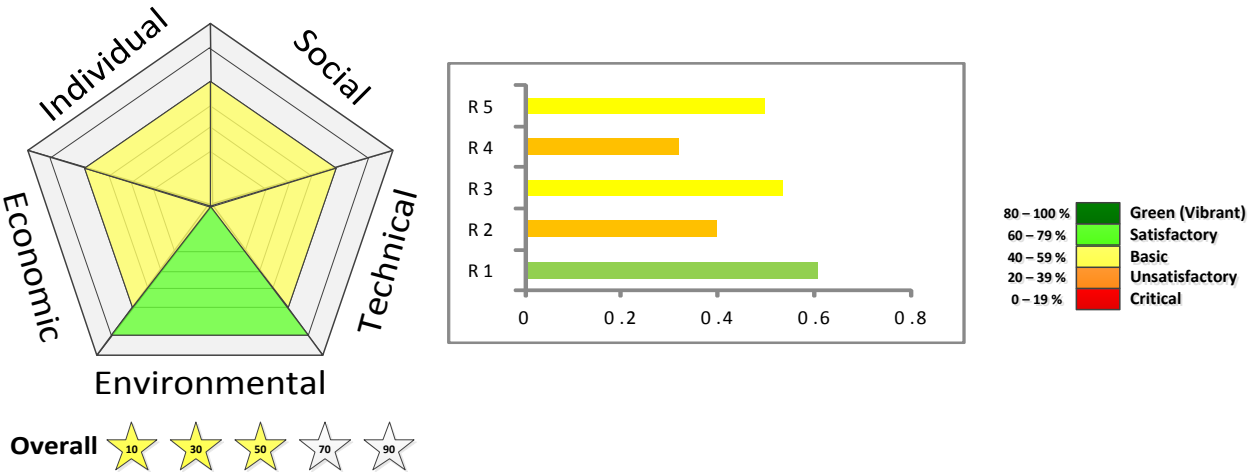


Fig. 5. Sustainability Profile of a Software System using the default colour schema. To increase accessibility of our approach, we also provide another colouring option for colour-challenged people. In this option the red colour is replaced by blue.

TABLE III
THE NORMALISATION DECISION (STEP 1) USING EQUATION 1

Dimensions	R1	R2	R3	R4	R5
Individual	0.536	0.238	0.529	0.336	0.512
Social	0.462	0.423	0.425	0.498	0.422
Technical	0.444	0.496	0.275	0.408	0.559
Economic	0.421	0.533	0.304	0.562	0.358
Environment	0.431	0.414	0.431	0.374	0.561

TABLE IV
THE WEIGHTED NORMALISATION STEPS FROM EQUATION 2

Dimensions	R1	R2	R3	R4	R5
Individual	0.146	0.033	0.166	0.063	0.126
Social	0.085	0.081	0.084	0.108	0.067
Technical	0.097	0.136	0.043	0.089	0.144
Economic	0.058	0.105	0.035	0.113	0.040
Environment	0.081	0.084	0.094	0.066	0.128

TABLE V
RESULTS OF THE STEPS 4 AND 5 IN THE FIRST ROUND

Dimensions	S*	S'	C*
Individual	0.0917	0.130	0.586
Social	0.143	0.118	0.452
Technical	0.134	0.137	0.505
Economic	0.132	0.104	0.440
Environment	0.093	0.151	0.617

TABLE VI
RESULTS OF THE STEPS 4 AND 5 IN THE SECOND ROUND

	R1	R2	R3	R4	R5
S*	0.116	0.139	0.135	0.191	0.121
S'	0.179	0.091	0.154	0.088	0.119
C*	0.607	0.394	0.533	0.317	0.497

the five requirements, R1 meets the highest level as satisfactory as well as environmental dimensions. Also, individual, social, technical and economic dimensions become basic as the lowest level of software sustainability including the five requirements in this example.

V. DISCUSSION AND CONCLUSIONS

In this paper, we introduced a framework for software sustainability profiling. We also presented and example scenario to provide a numerical illustration on how the framework can be applied. The framework allows to create the following profiling elements:

1) *Sustainability five-star rating* for overall sustainability ranking of entire software requirements;
2) *Visualisation of the five sustainability dimensions* as a pentagon graph (and, optionally, also a bar graph) for all dimension levels of the entire software requirements; and
3) *Bar graph* for overall sustainability of each requirement.

In our framework we apply a quantitative approach to measure sustainability of the software systems. The fuzzy rating scale is utilised to overcome inexplicit choices in questionnaires and increase the usability of the framework. The TOPSIS approach for requirements analysis is used to analyse ranking within the best ideal solution and the worst ideal solution among requirements that could assist to recognise the positive and negative impacts on sustainability via maximising or minimising the benefit or cost.

In the case of long-living systems, it is crucial to keep the software system sustainable over the whole lifecycle of the system. The stakeholders requirements and preferences might change over the time the system is in use, and proposed framework allows to analyse the up-to-date system sustainability profiles, based on system characteristics and the up-to-date ratings (quotations) of the corresponding requirements.

One of the core features of the framework is readability of the sustainability profiles, which also implies the usability of the proposed framework. For example, we apply the five-star rating to present sustainability ratings, as this rating is perceived as a common one in other areas: the five-star rating has become a standard for electricity consumption labelling in electronic appliances such as air conditioners and computer

monitors, allowing an energy efficient choice by reducing energy use and emissions (i.e., to increase environmental sustainability).

We follow the traffic lights colouring schema, where critical values are marked red and green (vibrant) are marked green to increase readability and graphic visualisation. These colours and their descriptions have been used in *Green IT* and *Sustainability Developments*. To increase accessibility of our approach, we also provide another colouring option for colour-challenged people, where the red colour is replaced by blue. Finally, there are two options to present the five sustainability dimensions as a pentagon or bar graph because it might be argued that the pentagon graph could be harder to read and need more effort to analyse represented data than the bar graph, so we provide the bar graph option for representing the five sustainability dimensions.

Future work: The main direction of our future work on the proposed framework is to develop a tool for software sustainability profiling, allowing to perform the framework steps within a single platform. We also would like to apply the proposed framework to our earlier work on the analysis of the RE aspects of ELearning systems [31] as well as of geographically distributed systems and within global product development [32]–[34].

Acknowledgement

The first author is supported by a scholarship from Umm Al-Qura University, Saudi Arabia.

References

[1] B. Penzenstadler, A. Raturi, D. Richardson, and B. Tomlinson, "Safety, security, now sustainability: The nonfunctional requirement for the 21st century," *Software, IEEE*, vol. 31, no. 3, pp. 40–47, 2014.

[2] F. Berkhout and J. Hertin, "Impacts of information and communication technologies on environmental sustainability: Speculations and evidence," *Report to the OECD, Brighton*, vol. 21, 2001.

[3] P. Lago and T. Jansen, "Creating environmental awareness in service oriented software engineering," in *Service-Oriented Computing*. Springer, 2011, pp. 181–186.

[4] S. Naumann, M. Dick, E. Kern, and T. Johann, "The greensoft model: A reference model for green and sustainable software and its engineering," *Sustainable Computing: Informatics and Systems*, vol. 1, no. 4, pp. 294–304, 2011.

[5] B. Penzenstadler and H. Femmer, "A generic model for sustainability with process- and product-specific instances," in *Proceedings of the 2013 Workshop on Green in/by Software Engineering*, ser. GIBSE '13. New York, NY, USA: ACM, 2013, pp. 3–8.

[6] G. G. Calienes, "Requirements prioritization framework for developing green and sustainable software using anp-based decision making," 2013.

[7] B. Penzenstadler, "Infusing green: Requirements engineering for green in and through software systems," *Christopher Arciniega, Birgit Penzenstadler TechReport UCI-ISR-14-2 June*, 2014.

[8] C. Becker, S. Betz, R. Chitchyan, L. Duboc, S. M. Easterbrook, B. Penzenstadler, N. Seyff, and C. C. Venters, "Requirements: The key to sustainability," *IEEE Software*, vol. 33, no. 1, pp. 56–65, Jan. 2016.

[9] V. G. Duffy, *Improving Sustainability through Usability*. Cham: Springer International Publishing, 2014, pp. 507–519.

[10] S. de la Rosa de Sáa, M. A. Gil, G. González-Rodríguez, M. T. López, and M. A. Lubiano, "Fuzzy rating scale-based questionnaires and their statistical analysis," *IEEE Transactions on Fuzzy Systems*, vol. 23, no. 1, pp. 111–126, Feb 2015.

[11] M. Behzadian, S. K. Otaghsara, M. Yazdani, and J. Ignatius, "A state-of the-art survey of TOPSIS applications," *Expert Systems with Applications*, vol. 39, no. 17, pp. 13 051 – 13 069, 2012.

[12] D. Mairiza, D. Zowghi, and V. Gervasi, "Utilizing TOPSIS: A Multi Criteria Decision Analysis Technique for Non-Functional Requirements Conflicts," in *RE*, ser. Communications in Computer and Information Science, D. Zowghi and Z. Jin, Eds. Springer, 2014, vol. 432, pp. 31–44.

[13] P. Achimugu, A. Selamat, R. Ibrahim, and M. N. Mahrin, "A systematic literature review of software requirements prioritization research," *Information and Software Technology*, vol. 56, no. 6, pp. 568 – 585, 2014.

[14] E. Hull, K. Jackson, and J. Dick, *Requirements engineering*. Springer Science & Business Media, 2010.

[15] G. Lami, F. Fabbrini, and M. Fusani, *Software Sustainability from a Process-Centric Perspective*, ser. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2012, vol. 301, ch. Systems, Software and Services Process Improvement, pp. 97–108.

[16] B. Penzenstadler, *Green in Software Engineering*. Cham: Springer International Publishing, 2015, ch. From Requirements Engineering to Green Requirements Engineering, pp. 157–186.

[17] R. Goodland, "Sustainability: human, social, economic and environmental," *Encyclopedia of Global Environmental Change. John Wiley & Sons*, 2002.

[18] M. Razavian, G. Procaccianti, D. A. Tamburri *et al.*, "Four-dimensional sustainable e-services," *EnviroInfo, Sep*, 2014.

[19] P. James, *Urban sustainability in theory and practice: circles of sustainability*. Routledge, 2014.

[20] B. Stewart and A. Khare, "elearning and the sustainable campus," in *Transformative Approaches to Sustainable Development at Universities*. Springer, 2015, pp. 291–305.

[21] D. Gmach, Y. Chen, A. Shah, J. Rolia, C. Bash, T. Christian, and R. Sharma, "Profiling sustainability of data centers," in *Proceedings of the 2010 IEEE International Symposium on Sustainable Systems and Technology*, May 2010, pp. 1–6.

[22] E. A. Jagroep, J. M. van der Werf, S. Brinkkemper, G. Procaccianti, P. Lago, L. Blom, and R. van Vliet, "Software energy profiling: Comparing releases of a software product," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 523–532.

[23] T. Tullis and B. Albert, *Measuring the User Experience*, 2nd ed., ser. Interactive Technologies. Boston: Morgan Kaufmann, 2013.

[24] J. Horkoff and E. Yu, "Analyzing goal models: Different approaches and how to choose among them," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011, pp. 675–682.

[25] M. A. Lubiano, S. de la Rosa de Sa, M. Montenegro, B. Sinova, and M. ngeles Gil, "Descriptive analysis of responses to items in questionnaires. why not using a fuzzy rating scale?" *Information Sciences*, vol. 360, pp. 131 – 148, 2016.

[26] J. R. S. C. Mateo, *TOPSIS*. London: Springer London, 2012, pp. 43–48.

[27] R. H. Thayer and M. Dorfman, *Software Requirements Engineering*. Wiley-IEEE Press, 2000, ch. Introductions, Issues, and Terminology, pp. 3–40.

[28] B. Nuseibeh and S. Easterbrook, "Requirements engineering: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 35–46.

[29] P. Sawyer, I. Sommerville, and S. Viller, "Requirements process improvement through the phased introduction of good practice," *Software Process: Improvement and Practice*, vol. 3, no. 1, pp. 19–34, 1997.

[30] I. Sommerville, *Software Engineering*, 9th ed. USA: Addison-Wesley Publishing Company, 2010.

[31] A. D. Alharthi, M. Spichkova, and M. Hamilton, "Requirements engineering aspects of elearning systems," in *Proceedings of the ASWEC 2015 24th Australasian Software Engineering Conference*. ACM, 2015, pp. 132–133.

[32] M. Spichkova and H. Schmidt, "Requirements engineering aspects of a geographically distributed architecture," in *10th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2015)*, 2015.

[33] M. Spichkova, H. W. Schmidt, M. R. I. Nekvi, and N. H. Madhavji, "Structuring diverse regulatory requirements for global product development," in *Requirements Engineering and Law*. IEEE, 2015, pp. 57–60.

[34] M. Spichkova, H. Liu, and H. Schmidt, "Towards quality-oriented architecture: Integration in a global context," in *European Conference on Software Architecture Workshops*. ACM, 2015, p. 64.

Improving Recall in Code Search by Indexing Similar Codes under Proper Terms

Abdus Satter* and Kazi Sakib†
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh
Email: *bit0401@iit.du.ac.bd, †sakib@iit.du.ac.bd

***Abstract*—The recall of a code search engine is reduced, if feature-wise similar code fragments are not indexed under common terms. In this paper, a technique named Similarity Based Method Finder (SBMF) is proposed to alleviate this problem. The technique extracts all the methods from a source code corpus and converts these into reusable methods (i.e., program slice) through resolving data dependency. Later, it finds similar methods by checking signature (i.e., input and output types) and executing methods for a randomly generated set of input values. Methods are considered as feature-wise similar if these produce the same output set. In order to index these methods against common and proper terms, SBMF selects the terms that are found in most of the methods. Finally, query expansion is performed before searching the index to solve the vocabulary mismatch problem. In order to evaluate SBMF, fifty open source projects implementing nine different functionalities or features were used. The results were compared with two types of techniques - Keyword Based Code Search (KBCS) and Interface Driven Code Search (IDCS). On an average, SBMF retrieves 38% and 58% more relevant methods than KBCS and IDCS, respectively. Moreover, it is successful for all the features by retrieving at least one relevant method representing each feature whereas IDCS and KBCS are successful for 3 and 7 features out of 9 respectively.**

***Index Terms*—code search, code reuse, method search**

I. Introduction

The recall of a code search engine, indicated by the number of relevant codes that is retrieved from the code repository, usually depends on the indexing mechanism and query formulation techniques. Proper indexing and query understanding help to retrieve relevant code snippets that satisfy user needs [1]. Most of the code search engines employ Information Retrieval (IR) centric approaches for indexing source code [2]. The working principle behind these approaches is to construct a term-based index, by extracting keywords from source codes. A common problem of these approaches is that a pair of codes - having same functionality, but written using different keywords are indexed against different terms. A traditional code search engine misses some important code fragments, because of this keyword matching policy. It results in a low recall code search engine with poor performance on benchmark datasets [3].

To improve recall of a code search engine, similar code fragments should be indexed under the same terms. However, it is challenging to automatically and efficiently determine that two code fragments are identical or similar [4]. Although identical code fragments can be detected through keywords matching [5], detecting feature wise similar code blocks is difficult. The reason is that automatically perceiving the intent of a code block is still a research challenge [6]. Another challenge is to select proper terms that best represent similar code fragments. For example, assume that there are two methods that perform bubble sort - "x" and "sort". Here, between two terms, "sort" is semantically more relevant name than "x". It is a challenging task to automatically determine that "sort" is the better keyword to represent these methods. Again, a code fragment may contain terms, which are not useful to express its intent (i.e., implemented feature) properly. Indexing based on these keywords reduces matching probability between user query and these keywords. It happens because, user query defines functionality but the extracted keywords do not express the feature properly. So, instead of using these keywords, more meaningful terms need to be selected that best match the query.

Researchers have proposed various techniques to improve the performance of code search engines where recall is considered as one of the performance indicators. These techniques can be broadly classified into four types like Keyword Based Code Search (KBCS), Interface Driven Code Search (IDCS), Test Driven Code Search (TDCS), and Semantic Based Code Search (SBCS). In KBCS [2], [7], [8], [9], [10], source codes are indexed based on the terms generated from the code and searching is performed on the index. As this approach does not consider similarity between source codes having different keywords, it cannot retrieve more relevant codes. In order to define required component interface as query, and find relevant components, IDCS [11], [12], [13] was proposed. It is possible to have two or more code fragments that contain different interfaces but perform the same task. IDCS considers that these code fragments are different due to having different interfaces. Thus, it does not retrieve these all together. To automatically find and adapt reusable components, TDCS [14], [15] and SBCS [16], [17] were proposed. These are effective in terms of precision as test cases are employed on the retrieved codes. In these approaches, most of the test cases fail not only for functional requirements mismatch but also for syntactic mismatch of the interface definition [15]. For this reason, semantically relevant code fragments cannot be retrieved and the recall is decreased.

In this paper a technique named Similarity Based Method Finder (SBMF) is proposed to retrieve more relevant methods from code base. The technique first parses all the methods from the source code to construct a repository of methods. It generates data dependency graph for each method and

converts the method into reusable method (i.e., program slice) through resolving data dependency, and redefining parameters and return type. Later, all the methods are clustered into a number of clusters where methods in the same cluster perform the same task. To detect feature-wise similarity among a set of methods' signatures (i.e., parameters and return types) of these methods are checked. Methods having the same signature are then executed against a set of randomly generated input values. Among these methods, those which produce the same output are considered as feature-wise similar and a cluster is constructed to store these methods. To identify proper terms for a cluster, keywords are obtained from the methods in the cluster and method frequency is calculated for each term. Such terms are considered as representative terms if these are found in most of the methods of the cluster. All the methods of the cluster are then indexed against the terms so that these are retrieved all together if a query term matches one of these methods. At last, user query is expanded by adding synonyms of each query term to increase the matching probability between the query terms and index terms [7].

In order to evaluate the proposed technique, a tool was developed. Two types of code search techniques, KBCS and IDCS, were compared with SBMF to show its efficiency. An existing system named Sourcerer [8] was used for the implementation of KBCS and IDCS. However, SBCS and TDCS were not considered for comparison, because these were proposed to improve precision rather than recall. For comparative result analysis, three metrics were used which are *recall*, *number of methods retrieved* and *feature successfulness*. Here, *feature successfulness* determines whether at least one relevant method is retrieved or not against user queries provided for a feature. In the context of this paper, *A feature can be considered as a requirement given to a developer to implement*. 50 open source projects were selected to carry out the experiment. The result analysis shows that on an average SBMF increases recall by 38% and 58% more than KBCS and IDCS, respectively against 170 queries. Besides, SBMF is successful for all the features whereas KBCS and IDCS are successful for 7 and 3 features out of 9 respectively.

II. Related Works

Reusing existing code fragments reduces development time and effort [18]. For this reason, searching for reusable code snippets has become a common task among the developers during software development [19]. Various techniques have been proposed in the literature to improve the performance of code search engine in terms of recall, precision, query successfulness, etc. These techniques can be broadly classified into four categories which are Keyword Based Code Search (KBCS), Interface Driven Code Search (IDCS), Semantic Based Code Search (SBCS), and Test Driven Code Search (TDCS). Significant works related to each category are discussed in the following subsections.

A. Keyword based Code Search (KBCS)

In KBCS, source code is considered as plain text document where traditional IR centric approaches are employed to index the code and query over the index [20]. Besides, other metadata such as comments, file name, commit message, etc. are used to retrieve relevant code fragments from a repository of source codes. One of the techniques related to KBCS is JSearch which indexes source code against the keywords extracted from the code [2]. However, it cannot retrieve all the code snippets that implement the same feature but contain different keywords. This is because, it does not check feature-wise similarity to detect common terms for these fragments.

Several techniques like Sourcerer [8], Codifier [9], krugle [10], etc. were proposed to provide infrastructure for large scale code search. These techniques use both structural and semantic information of source code to construct index. Structural information comprises language, source file, related documents, classes, methods, dependencies, and so on. Semantic information of a program is gathered by generating terms from method name, class name, field name, comments, etc. Although these techniques adopt both types of information to fetch more relevant code fragments, these cannot retrieve feature-wise similar code blocks simultaneously. The reason is that all these information are stored following IR based indexing mechanism, and no checking is performed to index similar code snippets under common proper terms.

B. Semantic Based Code Search (SBCS)

As open source codes are increasing day by day, it is thought that a significant amount of code that is written today, has already been available in the internet. However, reusing these existing codes often does not directly meet user needs or requires modifications. In order to find existing codes that support user requirements, a technique in form of SBCS was proposed by Steven [16]. It takes keywords that represent user requirements, and retrieves relevant code fragments containing these keywords. Later, it runs user provided test cases on the fetched code snippets and passed codes are delivered as final search result. It performs well in terms of precision but recall is reduced since proper terms are not determined while indexing feature-wise similar codes. So, some semantically similar code fragments cannot be fetched due to indexing these under inappropriate terms.

Sometimes, developers need to convert one type of object to another. To get example code implementing such conversion, Niyana proposed a technique named XSnippet [17]. It creates graph from source code by adopting code mining algorithm. The graph represents data flow within the corresponding source code. Moreover, user query is defined by providing input type and output type. For a user query, all the generated graphs are searched to find those code fragments that convert the input type into the output type. In this technique. developers need to provide exact input type and output type for getting example code blocks. Otherwise, it cannot retrieve code fragments that may satisfy user needs. However, according to the searching behavior, developers are

more interested in using keywords rather than concrete data types to define their query [21].

C. Test Driven Code Search (TDCS)

TDCS is a special type of SBCS where test cases are used to obtain program semantics. Lemos et al. proposed a TDCS technique named CodeGenie to support method level searching [14]. The technique takes method signature as query from the test cases written by developers. It uses Sourcerer infrastructure to retrieve relevant functions against the query. Next, all the test cases are executed for each retrieved method. Resultant methods are ranked based on the number of test cases successfully passed. Although the technique increases precision, it produces low recall. The reason is that it performs keyword matching to fetch methods from index without justifying the appropriateness of the keywords.

Usually retrieved methods may not pass corresponding test cases due to different order of the parameters, return type or parameter type. To resolve these issues, Janjic et al. proposed a technique that refactors the code to adapt with the program context [15]. It applies every possible adaptations like reordering parameters, using super type or sub class type of a given return or parameter type, converting primitive type to reference type, etc. Thus, it improves TDCS by finding more relevant methods. However, it produces low recall because it does not index similar methods under common terms.

D. Inreface Driven Code Search (IDCS)

IDCS helps the developers to define their queries in a more structured form rather than just a set of keywords joined by boolean expression. Signature matching was the first proposed IDCS technique to find relevant functions within a software library [13]. The approach crawls all the methods in the library, and uses signature of each method for indexing. Other code search techniques such as Sourcecer, ParseWeb, and Strathcona also support IDCS to improve the performance in code search [7]. Although IDCS assists to formulate user query, it does not select appropriate terms during indexing similar codes that perform the same functionality. Thus, functionally related code fragments will not be retrieved all together since these are indexed against inappropriate terms.

In order to find reusable code fragments, four types of techniques have been proposed in the literature which are KBCS, IDCS, TDCS, and SBCS. All these techniques extract keywords from source code to generate terms, and index corresponding code against the terms. However, none of the techniques checks the appropriateness of the terms with respect to implemented feature. As a result, the number of relevant codes retrieved is reduced due to indexing against improper term. Moreover, if two or more code snippets implement similar feature but contain different terms, existing techniques cannot retrieve all these code fragments simultaneously. The reason is that these are indexed against different terms. So, to improve recall in code search, feature-wise similar codes should be indexed under common appropriate terms.

III. Proposed Technique

In this paper, a technique named Similarity based Method Finder (SBMF) has been proposed to improve recall in code search. The technique comprises several steps such as *Reusable Method Generation*, *Clustering Similar Methods*, *Proper Term Selection*, *Handling Methods Having API/Library Function call*, *Index Construction*, and *Query Expansion*. Each of the steps is discussed as follows.

A. Reusable Method Generation

In this step, the proposed technique first parses the source code to identify all the methods in the code. For each method, it checks whether the body of that method contains any API/-function call statement or not. If no such statement is found, the technique takes the method to convert it into reusable method (i.e., program slice that can execute independently without having any dependency on other methods). Later, a data dependency graph is constructed for the corresponding method to determine its input and output types. Although the signature of the method expresses the input and output types, this is not sufficient enough to convert into reusable function for several scenarios. For example, a method may have return type void but it may manipulate one or more variables that are declared outside the body of the method. A method may not have any parameter (i.e., void) but use variables that are defined outside the body of the method. Again, the signature of a method may explicitly state the input and output types but some variables may be used or manipulated by it and these are declared outside the method body. Considering all of these scenarios, the technique generates data dependency graph to redefine the signature and convert into reusable method. Each node in the graph denotes the variable and an edge from a to b ($a \rightarrow b$) denotes variable a depends on variable b. After constructing the graph, nodes that have in degree zero and variables denoted by these nodes are declared outside the method body, are considered as input parameters. Besides, nodes that have out degree zero are considered as output variables of the method. If multiple output nodes are found, a complex data type is created where each field of the type denotes each node. The reason is that a method return type can be a single data type - either primitive or complex data type. The technique uses the variables found in the nodes containing in degree zero to generate parameters of the method. If a single node is found which out degree is zero, the type of the variable denoted by the node is used as return type of the method. Otherwise, generated composite data type as discussed earlier is used. The signature of the method is redefined by combining the return type, method name, and parameters. It is possible to have one or more variables that are declared outside the method body. In the data dependency graph, nodes representing these variables may have at least one in degree and one out degree. In this case, the technique parses the source code and checks the declaration statements of the variables to determine the types of the variables. Using this information, it adds declaration statement for each of the variables at the beginning of the function body. Thus, the

technique makes the method self-executable without having any external data dependency.

B. Clustering Similar Methods

To improve code search, it is required to check the similarity among methods found in the code base. Two or more methods may perform the same task in different ways. So, feature-wise similar methods needs to be detected to retrieve the similar methods all together. In Algorithm 1, the procedure named $ClusterSimilarMethods$ takes a list of reusable methods (M) as input which is constructed following the previous step. A variable C is declared to store different clusters of similar methods where each cluster contains the methods that perform the same functionality (Algorithm 1 Line 2). A *for* loop is declared that iterates on M to construct cluster of similar methods. The procedure $IsInAnyCluster$ is invoked to check whether each method m (belongs to M) is added to any cluster or not previously (Algorithm 1 Lines 4-5). If m does not belong to any cluster, a variable cl is declared to contain all the methods similar to m. A set of input data is generated based on the type of parameters found in the signature of m and corresponding output is generated by executing m (Algorithm 1 Lines 9-10). Here $inputset$ and $outputset$ determine the intent of m. Another *for* loop is declared to identify other methods that are similar to m. In each iteration, the signature of each method m' (in M) is matched with the signature of m to check whether the input data set can be fed into the method and return type is identical to m (Algorithm 1 Line 15). If the signatures of both methods are identical, the method m' is executed for $inputset$ and generated output is stored to $outputset'$. If $outputset$ and $outputset'$ are found the same, m' is considered similar to m as both methods produce same output for the same input data set (Algorithm 1 Lines 17-19). m' is then added to cl to store all the methods similar to m. At last, cl is inserted to the list of all identified clusters (C).

Algorithm 1 Cluster Similar Methods

Require: A list of methods (M) for which search index will be constructed

1: **procedure** CLUSTERSIMILARMETHODS(M)
2: $C = \emptyset$;
3: **for** each $m \in M$ **do**
4: **if** $IsInAnyCluster(m, C) == true$ **then**
5: continue
6: **end if**
7: $cl = \emptyset$
8: $cl.add(m)$
9: $inputset$ = generate a set of input data randomly for m
10: $outputset$ = execute m and generate corresponding output for $inputset$
11: **for** each $m' \in M$ **do**
12: **if** $IsInAnyCluster(m', C) == true$ **then**
13: continue
14: **end if**
15: **if** $m'.paramtersTypes == m.parametersTypes$ & $m'.returnType = m.returnType$ **then**
16: $outputset'$ = execute m' and generate corresponding output for $inputset$
17: **if** $outputset == outputset'$ **then**
18: $cl.add(m')$
19: **end if**
20: **end if**
21: **end for**
22: $C.add(cl)$
23: **end for**
24: **end procedure**
25: **procedure** ISINANYCLUSTER(m, C)
26: $found = false$
27: **for** $c \in C$ **do**
28: **if** $m \in c$ **then**
29: $found = true$
30: break;
31: **end if**
32: **end for**
33: **return** $found$
34: **end procedure**

C. Proper Term Selection

In order to retrieve more relevant methods, it is required to identify proper terms for each method before indexing. When two or more methods have different names or signatures, but implement the same functionality, these methods should be indexed under common appropriate terms. As a result, all these methods will be obtained against user query. So, after getting all the clusters from the previous step, representative terms are selected for each cluster. For a cluster, terms are obtained from the methods found in the cluster through extracting, tokenizing, and stemming keywords found in the methods. Terms that are found in most of the methods are considered as final representative terms for each of these methods.

D. Handling Methods Having API/Library Function call

As developers also search for example code to understand the usage of an API, in this step, methods that have API call statements are gathered. For each identified method, terms are generated from API call statements to index against the terms. As a result, if a query term does not match with the signature of a method but matches with the API invocation statements, the method is retrieved as API usage example code.

E. Index Construction and Query Expansion

After generating appropriate terms for each method and merging similar ones, an index is built for searching desired methods. A posting list is created to construct index, which maps terms with corresponding methods. Later, user query is expanded to retrieve more relevant methods against the query.

Two procedures named $ConstructIndex$ and $Query$, are presented in Algorithm 2 to build index of methods obtained from the previous steps, and refine user query, respectively. To

construct the index, an empty posting list is declared, which maps each term to corresponding methods (Algorithm 2 Lines 2). A nested *for* loop is defined, where the outer loop iterates on a list of methods (M) given as input to the procedure (Algorithm 2 Lines 3-4). The inner loop iterates to get all the terms of each method in M. In addition, each term is checked whether the posting list contains it or not to add a new term in the list (Algorithm 2 Lines 5-7). Next, the method is added to the posting list against the term so that, when a query term will match with the term, the method will be retrieved (Algorithm 2 Lines 8). After adding all the methods, the list is returned by the procedure (Algorithm 2 Lines 11).

In procedure $Query$, a boolean query is given as an argument, from which terms are separated and stored in a variable named $queryTerms$ to expand the query (Algorithm 2, Lines 13-14). A nested *for* loop is defined, where the outer loop iterates on these terms (Algorithm 2, Lines 15-17). In each iteration, a temporary variable ($expandedTerm$) initialized with the corresponding term, is used to store synonyms of the term. To expand each term, synonyms are appended to $expandedTerm$ in the inner loop (Algorithm 2, Lines 17-19). Later, each term in $queryTerms$ is replaced with corresponding $expandedTerm$ for the expansion of the query (Algorithm 2, Lines 20). As a result of the expansion, the probability of matching a query string with the terms defined in the index increases. Finally, the query is executed in the index to retrieve intended methods, which are returned by the procedure (Algorithm 2, Lines 22-23).

Algorithm 2 Index Construction and Query Expansion

Require: A list of methods (M) containing signature, body and terms of each method

1: **procedure** CONSTRUCTINDEX(M)
2: $Map < String, List < Method >> postingList$
3: **for** each $m \in M$ **do**
4: **for** each $t \in m.terms$ **do**
5: **if** $!postingList.keys.contains(t)$ **then**
6: $postingList.keys.add(t)$
7: **end if**
8: $postingList[t].add(m)$
9: **end for**
10: **end for**
11: **return** $postingList$
12: **end procedure**
13: **procedure** QUERY($booleanQueryStr$)
14: $queryTerms$=get all terms from $booleanQueryStr$
15: **for** each $qt \in queryTerms$ **do**
16: $expandedTerm = qt$
17: **for** each $syn \in$ synonyms of qt **do**
18: $expandedTerms+ =$" OR "$+syn$
19: **end for**
20: $queryTerms$.replace($qt, expandedTerm$)
21: **end for**
22: $methods$ = obtain method from the index satisfying $queryTerms$
23: **return** $methods$
24: **end procedure**

IV. Implementation and Result Analysis

In order to perform comparative result analysis, the proposed technique (SBMF) was implemented in form of a software tool. 50 open source projects were selected as data sources for the experimental analysis. To evaluate the proposed technique, 170 queries representing 9 different features were executed by the tool. For comparative analysis, same queries were also run on Sourcerer that supports KBCS and IDCS.

A. Environmental Setup

This section outlines the softwares and frameworks required for the experimental analysis. SBMF was implemented using C# programming language. Moreover, some other tools were also used, which are addressed as follows:

- JavaParser: An open source library used to parse Java source code (https://github.com/javaparser)
- Apache Lucene: A popular search engine infrastructure used to index java methods and query over the index (https://lucene.apache.org/)
- Luke: Open source lucene client used to execute query on the lucene index and visualize the search results (https://github.com/DmitryKey)

B. Dataset Selection

In order to perform experimental analysis, 50 open source projects from sourceforge (https://sourceforge.net/) were selected. Fraser and Arcuri showed that these projects are statistically sound and representatives of open source projects [22].

A set of features were selected from the existing works in code search [7], [16], [23], [24] as shown in Table I. On the other hand, to evaluate the proposed technique, a set of queries is selected from [7]. Here, each query is related to a particular functionality shown in Table I and all the queries are created randomly. 15 subjects were employed to identify relevant methods for the functionalities. Among 15 subjects, 5 of them were senior Java developers and rest 10 were masters student. The reason of choosing students in this study is that they can play important role in software engineering experiments [25]. All the experimental datasets are available in this link[1].

C. Comparative Result Analysis

For comparative result analysis, SBMF was run on the experimental datasets and the relevance of retrieved methods were checked for each user query. Moreover, Sourcerer which supports KBCS and IDCS, was also run on the same datasets and search results obtained by this were compared to SBMF. Three metrics were used to evaluate the performance of SBMF in comparison with KBCS and IDCS. These were recall, number of retrieved methods, and feature successfulness. Detailed

[1] http://tinyurl.com/zdqmoqz

TABLE I
SELECTED FUNCTIONALITIES WITH FREQUENCY

#	Functionality	# methods	# queries
1	decoding String	13	20
2	encrypting password	3	27
3	decoding a URL	3	21
4	generating MD5 hash	3	16
5	rotating array	2	25
6	resizing image	3	25
7	scaling Image	3	19
8	encoding string to html	2	6
9	joining string	47	36

result analysis with respect to each of the metrics is discussed as follows.

Recall Analysis: Recall is one of the most commonly used metrics to measure the performance of traditional IR system. As the intent of the paper is to improve recall in code search, it is considered as an important metric to evaluate the proposed technique. In this experiment, recall is defined as follows.

$$recall = \frac{\text{number of retrieved relevant methods}}{\text{number of relevant methods in the repository}}$$

Fig. 1 depicts a comparative recall analysis among SBMF, KBCS, and IDCS where X axis denotes the feature no. as shown in TABLE I and Y axis represents the measured recall. For feature 1 (Decoding String), approximately 15% recall is shown in Fig. 1 for both KBCS and IDCS whereas 100% recall is found for SBMF. There are 13 methods in the repository that implement the feature. Among these, two methods are found which contain keywords *decode* and *string* in method name and parameter respectively. As a result, these methods are retrieved by both KBCS and IDCS. However, these techniques cannot retrieve other 11 methods because signatures of these methods do not contain any term related to *decode*. While analyzing the source code of these methods, it is seen that the bodies of these methods use third party APIs like *URLDecoder.decode(String, String), Hex.decode(String), Base64.decode(base64), etc.* to implement the feature. SMBF takes terms from API call statements and indexes against the terms to provide example codes regarding API usage. So, it retrieves all these 13 methods.

For feature 2 (Encrypting Password), IDCS cannot find any methods but 66.67% and 33.3% relevant methods are retrieved by SBMF and KBCS respectively as shown in Fig. 1. To get the methods that implement this feature, the following query is provided to IDCS.

name:(encrypt) AND return:(String) AND parameter:(String)

Although there is a single method found in the code base that has *encrypt* keyword in its name but does not have *String* in its parameter. So, IDCS cannot obtain this method but KBCS retrieves because query keyword matches with the method name. However, SBMF retrieves one more method having signature *crypt(String strpw,String strsalt)*. The reason is that *encrypt* and *crypt* both express the same intent as detected by the query expansion part of SBMF (Algorithm 2).

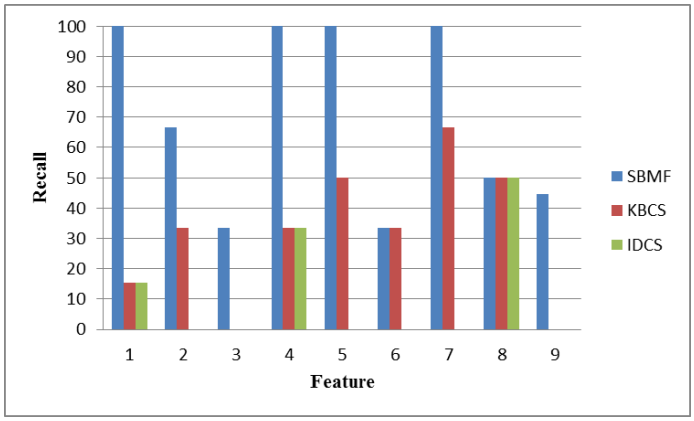


Fig. 1. Recall Analysis

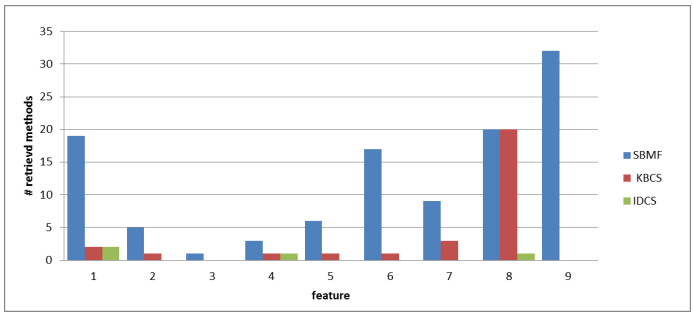


Fig. 2. Number of Retrieved Methods

There are 3 relevant methods in the experimental projects that implement feature no. 3 (Decoding a URL). According to Fig. 1 only a single method is retrieved by SBMF that produces recall 33.33%. On the contrary, KBCS and IDCS cannot retrieve any method related to the feature. This is because no method contains *decode* and *URL* simultaneously in the signature. Although one of these methods named *getPath* does not provide any semantic information representing the feature, it invokes a library method - *URLDecoder.decode(path, "UTF-8")* which implements the feature. SBMF considers the invocation statement for getting more relevant terms and thus, retrieves this method. Two other methods cannot be retrieved by SBMF due to finding no structural similarity among these and no keywords representing the feature.

According to Fig. 1, 100% recall is obtained for SBMF, and 33.33% for KBCS and IDCS individually with respect to feature no. 4 (Generating MD5 hash). It is clear that SBMF

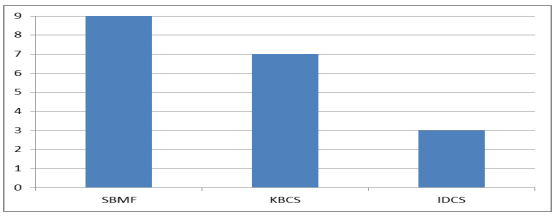


Fig. 3. Feature Successfulness Analysis

has higher recall than other two approaches. The reason is that most of the methods implementing this feature do not have proper names to represent their intent. There are 5 methods relevant to this feature and only one method is found having name consistent with the feature. KBCS and IDCS fail to retrieve all these methods because both techniques extract terms from individual method and do not consider appropriateness of the terms. However, SBMF finds that these methods are semantically similar. So, these methods are indexed under common terms. As a result, when user query matches with one of these methods, other three methods are also retrieved with this.

For feature no. 5, 6 and 7, IDCS cannot retrieve any method from the code base used in this experiment. The reason is that appropriate parameter type is not determined in the user queries used for this feature. However, KBCS shows 50%, 33.33%, and 66.67% recall for feature no. 5, 6 and 7 respectively. On the other hand, SBMF shows 100% for features no. 5 and 7, and 33.33% for feature no. 6 as illustrated in Fig. 1. For feature no. 5, two relevant methods are found which names are *transpose* and *rotate* correspondingly. These two methods are feature-wise similar which is detected by SBMF and indexed under common terms (i.e., rotate and transpose). On the other hand, KBCS does not check similarity, and analyzes each method individually during indexing. So, only *rotate* method is retrieved by KBCS. For feature no. 7, SBMF retrieves one more method than KBCS because this method does not contain any term related to *image* but it uses a field of type *Image*. SBMF considers this usage since scaling operation is performed on this field by the method, and adds additional term *Image* against the method.

SBMF, KBCS, and IDCS show equal performance for feature no. 8 (Encoding String to HTML) in terms of recall. However, 50% relevant methods cannot be retrieved because no HTML keyword is found in these method.

Only SBMF is able to retrieve 21 relevant methods whereas other techniques cannot fetch a single method for feature no. 9 (Joining String). Here, SBMF outperforms because it identifies many structurally similar methods which have different names but all these perform string concatenation. Among these, several methods are found which have proper keywords in their body. These keywords are attached to the term list of each similar method by SBMF. As a result, these are indexed under common appropriate terms and all these are retrieved simultaneously. However, other 26 relevant methods cannot be retrieved since no signature matching is found among these.

Number of Retrieved Methods (NRM) and Feature Successfulness Analysis: As NRM is an important measure to perceive recall of a search engine, a comparative result analysis with respect to NRM is performed here. A bar diagram is shown in Fig. 2 depicting feature-wise NRM by SBMF, KBCS, and IDCS. According to the diagram, SBMF retrieves more methods than KBCS and IDCS because of adding common terms to each method.

Although IDCS produces better precision than KBCS and SBMF, it cannot retrieve a single method for some features (such as feature No, 2, 3, 5, 6, 7, 9). The reason is that user queries do not have proper parameter type or return type. This scenario is common when developers have little or no knowledge about the implementation of a feature. KBCS and SBMF mitigate the problem by retrieving more relevant methods adopting free text search. In order to determine whether a feature is successful or not, a metric named feature successfulness is introduced. A feature is said to be successful if at least one relevant method is retrieved that implements the feature. Fig. 3 presents the number of successful features among SBMF, KBCS, and IDCS. According to this figure, SBMF is successful for all the 9 features whereas 7 and 3 successful features are found for KBCS and IDCS respectively. This measure provides a notion that having higher precision is not effective if number of successful feature is low. In addition, improving recall increases the chances of having higher number of successful feature. For this reason, SBMF performs better than KBCS and IDCS.

V. Threats to Validity

In this section, limitations of the experimental study are discussed in terms of internal, external, and construct validity.

a) Internal Validity: In the experiment, there was no control over the skills of the subjects. However, the risks of this threat are reduced by applying repetitive measurement approach because same user created queries for KBCS, IDCS, and SBMF and evaluated the search results.

b) External Validity: The set of features selected may not generalize to the population of software functions. However, these features are among the most common features used for the evaluation in code search. Another possible threat is that projects used in the experiment may not be sufficient enough. However, these projects are statistically representative of open source projects as highlighted in [7].

c) Construct Validity: Existing code clone detection technique can be used to improve recall in code search. However, SBMF differs from code clone detection in several points. SBMF can detect similar methods written in different programming languages and only the execution of method is platform dependent. Another point is that code clone detection technique may provide false positive results to feature-wise clone detection (usually known as Type IV) if values of certain parameters are not defined properly [26]. As a result search engine may retrieve irrelevant methods. However, SBMF checks dynamic behavior through executing method and matches the output for corresponding input to detect feature-wise similar methods. Such mechanism ensures that methods providing the same output, are feature-wise similar and thus no irrelevant method is added to these methods. Besides recall, two other metrics are used in the study to observe the effectiveness of the technique. Although precision is not shown directly in the result analysis due to space limitation, it can be obtained by using data given in TABLE I and Fig. 2.

VI. Conclusion

The recall of a code search engine reduces if similar code fragments are indexed under common proper terms. So, a

technique named SBMF is proposed in this paper which indexes both syntactically and semantically similar methods under common terms. The technique is implemented as a complete software, that constructs index and retrieves relevant methods against the user query.

SBMF first identifies all the methods in the code base by parsing the source code. It converts the methods into reusable methods by resolving data dependency and redefining method signature. Feature-wise similar methods are detected through checking signature and executing methods. Here, methods that produce the same output set for a randomly generated set of input values, are considered as similar methods and these are kept under a cluster. Thus, all the methods are distributed into a set of clusters where each cluster contains feature-wise similar methods and any two clusters differ from one another in implemented feature. All these methods are indexed against the terms that are found in more than half of the methods in the cluster. At last, query expansion is performed to increase the probability of retrieving more methods.

For experimental analysis of the technique, 50 open source projects were selected to build the code base and 9 features were chosen to generate queries. An existing technique named Sourcerer was used to compare the results to SBMF. While analyzing the results it has been seen that SBMF shows 38% improvement in recall than KBCS and 58% than IDCS. It also retrieves relevant methods for all the 9 features, whereas KBCS and IDCS retrieves for 7 and 3 features, respectively. In future, the experiment will be conducted on a large scale dataset to observe the behavior of the technique.

Acknowledgment

This research is supported by ICT Division, Ministry of Posts, Telecommunications and Information Technology, Bangladesh. 56.00.0000.028.33.065.16-747, 14-06-2016.

References

[1] Ruben Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, 1991.

[2] Renuka Sindhgatta. Using an information retrieval system to retrieve source code samples. In *Proceedings of the 28th international conference on Software engineering*, pages 905–908. ACM, 2006.

[3] Hinrich Schütze. Introduction to information retrieval. In *Proceedings of the international communication of association for computing machinery conference*, 2008.

[4] Randy Smith and Susan Horwitz. Detecting and measuring similarity in code clones. In *Proceedings of the International workshop on Software Clones (IWSC)*, 2009.

[5] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[6] Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. Mining concepts from code with probabilistic topic models. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 461–464. ACM, 2007.

[7] Otávio Augusto Lazzarini Lemos, Adriano Carvalho de Paula, Hitesh Sajnani, and Cristina V Lopes. Can the use of types and query expansion help improve large-scale code search? In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 41–50. IEEE, 2015.

[8] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682. ACM, 2006.

[9] Andrew Begel. Codifier: a programmer-centric search user interface. In *Proceedings of the workshop on human-computer interaction and information retrieval*, pages 23–24, 2007.

[10] Susan Elliott Sim and Rosalva E Gallardo-Valencia. *Finding source code on the web for remix and reuse*. Springer, 2013.

[11] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213. ACM, 2007.

[12] Reid Holmes, Robert J Walker, and Gail C Murphy. Strathcona example recommendation tool. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 237–240. ACM, 2005.

[13] Amy Moormann Zaremski and Jeannette M Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(2):146–170, 1995.

[14] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. Codegenie:: a tool for test-driven source code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 917–918. ACM, 2007.

[15] Werner Janjic and Colin Atkinson. Leveraging software search and reuse with automated software adaptation. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE), 2012 ICSE Workshop on*, pages 23–26. IEEE, 2012.

[16] Steven P Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, pages 243–253. IEEE Computer Society, 2009.

[17] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: mining for sample code. *ACM Sigplan Notices*, 41(10):413–430, 2006.

[18] Wayne C Lim. Effects of reuse on quality, productivity, and economics. *IEEE software*, 11(5):23–30, 1994.

[19] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. Code reuse in open source software. *Management Science*, 54(1):180–193, 2008.

[20] William B Frakes and Brian A Nejmeh. Software reuse through information retrieval. In *ACM SIGIR Forum*, volume 21, pages 30–36. ACM, 1986.

[21] Susan Elliott Sim, Charles LA Clarke, and Richard C Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 180–187. IEEE, 1998.

[22] Gordon Fraser and Andrea Arcuri. Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 178–188. IEEE Press, 2012.

[23] OtáVio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Information and Software Technology*, 53(4):294–306, 2011.

[24] Otávio AL Lemos, Adriano C de Paula, Felipe C Zanichelli, and Cristina V Lopes. Thesaurus-based automatic query expansion for interface-driven code search. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 212–221. ACM, 2014.

[25] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28(8):721–734, 2002.

[26] Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao Xie. Xiao: tuning code clones at hands of engineers in practice. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 369–378. ACM, 2012.