Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2014; **26**:172–219
Published online 29 August 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1616

Extended comprehensive study of association measures for fault localization[‡]

Lucia *,[†], David Lo, Lingxiao Jiang, Ferdian Thung and Aditya Budi

School of Information Systems, Singapore Management University, Singapore

ABSTRACT

Spectrum-based fault localization is a promising approach to automatically locate root causes of failures quickly. Two well-known spectrum-based fault localization techniques, Tarantula and Ochiai, measure how likely a program element is a root cause of failures based on profiles of correct and failed program executions. These techniques are conceptually similar to *association measures* that have been proposed in statistics, data mining, and have been utilized to quantify the relationship strength between two variables of interest (e.g., the use of a medicine and the cure rate of a disease). In this paper, we view fault localization as a measurement of the relationship strength between the execution of program elements and program failures. We investigate the effectiveness of 40 association measures from the literature on locating bugs. Our empirical evaluations involve single-bug and multiple-bug programs. We find there is no best single measure for all cases. Klosgen and Ochiai outperform other measures for localizing single-bug programs. Although localizing multiple-bug programs, Added Value could localize the bugs with on average smallest percentage of inspected code, whereas a number of other measures have similar performance. The accuracies of the measures in localizing multi-bug programs are lower than single-bug programs, which provokes future research. Copyright © 2013 John Wiley & Sons, Ltd.

Received 8 April 2012; Revised 4 July 2013; Accepted 19 July 2013

KEY WORDS: Association Measures; Fault Localization; Program Spectra

1. INTRODUCTION

Software debugging is a difficult and expensive activity to perform. US National Institute of Standards and Technology has reported that software bugs cost US economy $59.5bn annually [1], and testing and debugging activities account for 30–90% of labor expended for a project [2]. When a program failure occurs, the execution trace may be long and contain failure-irrelevant information. Often, full execution traces are not even available for debugging if programs fail in the field. Locating the root cause of the failure, which may be far away from the failure point, is non-trivial.

Many approaches have been proposed to help in automating debugging activities, especially in localizing root causes of failures (i.e., *faults*) in programs [3–11]. One family of approaches is spectrum-based fault localization [3, 4, 12–14], where program traces or abstractions of traces (called *program spectra*) are used to represent program behaviors, and the spectra of correct and failed executions are compared to identify potentially faulty program elements (e.g., statements, basic blocks, functions, and components). The comparison often employs statistical analysis, and the program elements that are observed more often in failed executions than in correct executions (or statistically correlate with failures) may be identified and presented to developers for further inspection.

*Correspondence to: Lucia, 80 Stamford Road, School of Information Systems, Singapore Management University, Singapore 178902.
[†]E-mail: lucia.2009@smu.edu.sg
[‡]Supporting Information may be found in the online version of this article.

Spectrum-based fault localization techniques are promising as they are lightweight and have good accuracy. Among spectrum-based fault localization techniques, two well-known techniques are Tarantula [4, 15] and Ochiai [13, 16, 17]. Both approaches compute a suspiciousness score for each program element based on the execution frequencies of the element in correct and failed executions, and rank all program elements according to their scores. Thus, higher suspiciousness scores indicate more likely faulty program elements, and the computation of the scores in effect answers the following question:

What is the strength of association between the executions of a particular program element with failures?

On the basis of this question, a general solution for fault localization can naturally emerge based on the proliferation of association measures in the literature: measure the strength of association of a program element's executions with failures, and the stronger the association is, the more likely the program element is a fault.

Besides Tarantula and Ochiai, which have been used for fault localization, various association measures in the data mining and statistics communities, such as odds ratio [18], Yule's Q [19], and Yule's Y [20] have been proposed to measure the strength of the association of two variables. For example, one might be interested in the association between an application of a particular medical treatment with a recovery from an illness, or in the association between an execution of a business strategy with the revenue change. There are several studies that evaluate the effectiveness of some similarity coefficients, for example, Jaccard [13, 21], Sorensen-Dice [21], Anderberg [21], simple matching [21], Rogers and Tanimoto [21], and Ochiai II [21]. However, there are rich varieties of association measures other than those coefficients that have not been studied for fault localization.

This paper aims to fill this gap by investigating the effectiveness of 40 popular association measures for the purpose of fault localization and comparing them with Tarantula and Ochiai. In particular, we are interested in answering the following research questions (RQs):

RQ 1. Are vanilla or off-the-shelf association measures accurate enough in localizing faults?
RQ 2. Which off-the-shelf association measures are more accurate to localize faults?
RQ 3. What is the relative performance of the off-the-shelf association measures as compared with well-known suspiciousness measures for fault localization, Tarantula and Ochiai in particular?
RQ 4. Is the accuracy of the off-the-shelf association measures, Tarantula, and Ochiai different for programs written in C as compared with programs written in Java?
RQ 5. What is the effectiveness of the off-the-shelf association measures, Tarantula, and Ochiai in localizing different types of bugs?
RQ 6. What is the accuracies of the off-the-shelf association measures, Tarantula, and Ochiai in localizing faulty programs containing multiple bugs?

To answer the previous questions, we investigate and compare the accuracies of Tarantula, Ochiai, and the additional 40 association measures on programs from the Siemens test suite [22], and three larger programs from software infrastructure repository (SIR) [23]. The latter includes Space, which is written in C, and NanoXml and XmlSecurity, which are written in Java. The programs come along with seeded bugs, test oracles to decide between failures and non-failures and test cases. We compute various accuracy metrics to evaluate the effectiveness of the association measures in localizing faults to answer the previous research questions. We show that a few association measures have better performance than Ochiai and many are better than Tarantula. We also split the programs into those written in C and those written in Java, and analyze the accuracies of the association measures, Tarantula, and Ochiai separately for each of the two sets of programs. Furthermore, we characterize the bugs in the programs and group them into several categories. We then evaluate the effectiveness of the various association measures, Tarantula, and Ochiai to localize different categories of bugs and also evaluate their accuracies in localizing bugs for programs.

The contributions of this work are as follows:

1. We comprehensively investigate the effectiveness of 40 association measures for fault localization.
2. We highlight a few promising association measures, which can outperform Tarantula and Ochiai and those that are comparable with the two well-known spectrum-based fault localization approaches.

DOI: 10.1002/smr

3. We provide a partial order of association measures in terms of their accuracies for fault localization.
4. We characterize the effectiveness of the association measures, Ochiai, and Tarantula on programs written in different programming languages.
5. We analyze different kinds of bugs and investigate the effectiveness of the 40 association measures, Ochiai, and Tarantula in localizing each of these categories of bugs.
6. We investigate the accuracies of the 40 association measures, Ochiai, and Tarantula in localizing faulty programs containing multiple bugs.

The structure of this paper is as follows. Section 2 discusses related work. Section 3 discusses the fundamental concepts of spectrum-based fault localization and association measures. Section 4 discusses the particular association measures considered in the paper. Section 5 describes our empirical evaluation and comparison of the association measures. Finally, we conclude our work in Section 6.

2. RELATED WORK

In this section, we describe closely related studies on fault localization and association measures. The survey here is by no means a complete list of all related studies.

2.1. Fault localization

Recently, there are many studies on fault localization and automated debugging. There are different ways to categorize these studies. Based on the data analyzed by the approaches, fault localization techniques can be classified into *spectrum-based* and *model-based*.

2.1.1. Spectrum-based fault localization. Spectrum-based fault localization techniques often use program spectra, which are program traces or abstractions of program traces that represent program runtime behaviors in certain ways, to correlate program elements (e.g., statements, basic blocks, functions, and components) with program failures (often with the help of statistical analysis).

Many spectrum-based fault localization techniques [4, 5, 12, 24, 25] take as inputs two sets of spectra, one for successful executions and the other for failed executions, and report candidate locations where root causes of program failures (i.e., faults) may reside. Given a failed program spectrum and a set of correct spectra, Renieris and Reiss present a fault localization tool WHITHER [5] that compares the failed execution to the nearest correct execution and reports the most suspicious locations in the program. Liblit *et al*. propose a technique to search for predicates whose true evaluation correlates with failures [24]. Chao *et al*. extend the work by incorporating information on the outcomes of multiple predicate evaluations in a program run in their tool called SOBER [12]. Santelices *et al*. combine several program spectra to better localize bugs in programs. [25]. All of these techniques need to compare spectra of failed executions with those of successful executions in some way. Evaluating the effectiveness of various association measures can complement all of these techniques by helping to locate the most failure-relevant program elements quickly and improving their performance.

Artzi *et al*. evaluate the effectiveness of several test generation techniques in generating enough test cases for localizing faults in Web applications with the help of Ochiai [26]. Artzi *et al*. extend their work by proposing a tool named Apollo that can generate test cases to expose failures for Web applications and also localize bugs that cause the failures [27]. Their fault localization technique uses Tarantula and a technique that keep information about which program statements are potentially responsible to produce a particular part of an output (e.g., a table in an HTML document). This approach is possible for Web applications but may not be applicable to other applications as the output is often not decomposable into parts (it could be a single number), and the number of program statements that are potentially responsible to produce an output are often large. Bandyopadhyay and Ghosh study how the properties of faults affect the effectiveness of Tarantula [28]. Three properties namely accessibility, original state failure condition, and impact are

DOI: 10.1002/smr

investigated. To answer RQ5, we also investigate the effectiveness of various fault localization techniques on different kinds of bugs. However, we consider a different categorization of faults—our categorization is based on the empirical study performed by Kim *et al*. on bug fixes [29]. Jiang *et al*. study the effectiveness of test case prioritization for fault localization using different types of prioritization criteria [30].

Other spectrum-based techniques [8, 31–33] only use failed executions as the input and systematically alter the program structure or program runtime states to locate faults. Zhang *et al*. [31] search for faulty program predicates by switching the states of program predicates at runtime. Sterling and Olsson use the concept of program chipping [32] to automatically remove parts of a program so that the part that contributes to the failure may become more apparent. Although their tool, ChipperJ, works on syntax trees for Java programs, Gupta *et al*. [8] work on program dependency graphs and use the intersection of forward and backward program slices to reduce the sizes of failure-relevant code for further inspection. Jeffrey *et al*. use a value profile-based approach to rank program statements according to their likelihood of being faulty [33]. These fault localization techniques do not compare the spectra of failed executions with those of successful executions, and association measures are generally not applicable to them. List of programs that have been used by those past studies in fault localization is shown in Table I.

2.1.2. Model-based fault localization. Compared with spectrum-based techniques, model-based debugging techniques [9, 34–37] are often more accurate, but heavyweight because they are based on more expensive logic reasoning over formal models of programs. Many static and dynamic analysis techniques [6, 38, 39] can be classified as model-based debugging as well. Abreu *et al*. propose a framework called BARINEL that combines spectrum-based fault localization and model-based debugging to localize single and multiple bugs in programs, and found that the approach is more accurate and heavyweight than spectrum-based fault localization [17]. Although few model-based techniques have employed the concept of failure association, incorporating association measures investigated in this study into program models can be a future direction to improve the performance of model-based debugging techniques.

In our study, we focus on comparisons with two well-known spectrum-based fault localization techniques, namely Tarantula [15, 4] and Ochiai [13, 16, 17]. We evaluate 40 association measures and find promising ones for fault localization.

Table I. Subject programs used in past fault localization studies.

Dataset	LOC	Papers	Dataset	LOC	Papers
SumPowers	27	[35, 36]	Power	763	[24]
BinSearch	29	[35, 36]	Compress	1590	[24]
BubbleSort	29	[35, 36]	Bh	2053	[24]
Hamming	48	[35, 36]	Webchess v.0.9.0	2226	[26, 27]
Adder	49	[35, 36]	Tetris	2403	[40]
Permutation	54	[35, 36]	Schoolmate v.1.5.4	4263	[26, 27]
Binomia	80	[35, 36]	Gzip	5680	[30, 17]
Polynom	105	[35, 36]	Space	6218	[4, 17, 15, 21]
Tcas	141	[4, 5, 12, 28, 30, 33, 17, 35–37, 16]	NanoXML	7646	[40, 25]
Schedule	292	[4, 5, 12, 28, 30, 8, 33, 17, 16, 13, 25]	Phpsysinfo v.2.5.3	7745	[26, 27]
Schedule2	301	[4, 5, 12, 28, 30, 8, 33, 17, 16, 13, 25]	Li	7761	[24]
Treeadd	385	[24]	Grep	10068	[30, 31]
Perimeter	395	[24]	Flex	10459	[30, 31]
Print_token2	399	[4, 5, 12, 28, 30, 8, 33, 17, 16, 13, 25]	Sed	14427	[30, 17]
Tot_info	440	[4, 5, 12, 28, 30, 33, 17, 16, 13, 25]	XMLsecurity	16800	[25]
Print_token	478	[4, 5, 12, 28, 30, 8, 33, 17, 16, 13, 25]	Bc-1.06	17042	[31]
Replace	512	[4, 5, 12, 28, 30, 8, 33, 17, 16, 13]	Tar-1.13.25	27137	[31]
Emad	557	[24]	Go	29315	[24]
Rest	617	[24]	Ijpeg	31371	[24]
Bisort	707	[24]	Make	35545	[31]
Health	725	[24]	JABA	37966	[25]
Faqforge v.1.3.2	734	[26, 27]			

DOI: 10.1002/smr

2.2. Studies on association measures

There have been a number of studies proposed in the statistics and data mining community on measures of association between variables since the early 20th century. These include measures such as Yule's Q and Yule's Y [19, 20]. Other measures, such as odds ratio [18], are also commonly considered and utilized in various domains, such as medical [41] and social science [42]. In the data mining community, Agrawal and Srikant have proposed association rule mining, which aims to infer associations from two item sets in a transaction dataset in the early 1990s [43]. In that work, the metrics of support and confidence for measuring the strength of an association are proposed. Various other metrics, such as interest and collective strength, are proposed later. We describe these measures in detail in Section 4.

Tan *et al*. investigate various association measures, compare their properties, and outline the benefits and limitations of each from a computational point of view [44]. The measures are revisited by Geng and Hamilton by including measures for aggregated data summaries [45]. In this paper, we extend their work in the specific domain of fault localization by comparing 40 association measures based on their ability to assign high-suspiciousness scores for buggy program elements and low scores for non-buggy ones.

Some of the association measures that we evaluate in this paper, have been studied for fault localizations, for example, Jaccard [13, 21], Sorensen-Dice [21], Anderberg [21], simple matching [21], Rogers and Tanimoto [21], and Ochiai II [21]. In this paper, we revisit the effectiveness of these measures as their effectiveness on various kinds of bugs and program containing multiple bugs has not been extensively evaluated.

3. CONCEPTS & DEFINITIONS

In this section, we formally introduce the problem of spectrum-based fault localization as the computation of association strengths between the executions of various program elements and failures. Also, we describe the concept of dichotomy matrix that is used in the calculation of these association strengths.

3.1. Spectrum-based fault localization

This problem starts with a faulty program, a set of test cases, and a test oracle. The set of test cases are run over the faulty program and observations of how the program runs on each of the test cases are recorded as program spectra. A program spectrum represents certain characteristics of an execution of a program, providing a behavior signature of the execution [46]. The signature of a behavior could be a set of counters, each of which indicates the number of times each program element (e.g., statement, basic block, path, etc.) is executed in one execution [47]. The counters could also simply be 0–1 flags that indicate whether an element is executed. A test oracle is available to label whether a particular output or execution of a test case is correct or wrong. Wrong executions are classified as program failures. The task of a fault localization tool is to find the program elements that are responsible for the failures (i.e., the faults or the root causes) based on the program spectra of both correct and wrong executions.

There have been various spectra proposed in the literature [13, 47]. Different spectra may have different effects on effectiveness of fault localization. The block-hit spectra are at a suitable profiling granularity because all code in the same basic block has the same execution pattern and there is no need to instrument individual instructions in a granularity finer than blocks, and because it has been shown in the literature that the instrumentation costs to obtain such spectra are relatively low and it can be used for effective fault localization [13, 16, 17, 21, 47]. The granularity has a balance between reducing instrumentation costs and having sufficient bug-revealing capabilities.

In this paper, we use *block-hit program spectra*, each of which consists of a set of flags to indicate whether each basic block is executed or not in each test case. An example of block-hit program spectra is shown in Figure 1. The first column contains identifiers of basic blocks. The second column contains the statements in the corresponding basic blocks. The other columns indicate whether each basic block

DOI: 10.1002/smr

Block ID	Program Elements	T15	T16	T17	T18
1	int count; int n; Ele *proc; List *src_queue, *dest_queue; if (prio >= MAXPRIO) /*maxprio=3*/	•	•	•	•
2	{return;}		•	•	•

3	src_queue = prio_queue[prio];
	dest_queue = prio_queue[prio+1];
	count = src_queue->mem_count;
	if (count > 1) /* Bug*//* supposed : count>0*/ {
		•	•	•	•

4	n = (int) (count*ratio + 1);
	proc = find_nth(src_queue, n);
	if (proc) {
			•	•	

5	src_queue = del_ele(src_queue, proc);
	proc->priority = prio;
	dest_queue = append_ele(dest_queue, proc); } }}
			•	•	

	Status of Test Case Execution :	Pass	Pass	Pass	Fail

Figure 1. Example of block-hit program spectra.

is executed in test cases T15, T16, T17, and T18 along with the information whether each of the test cases passes or fails. In this example, • denotes that a basic block is executed by a test case and an empty cell denotes that the block is not executed by the test case. In the code snippet, a bug lies in the condition of the `if` statement in Block 3, causing Blocks 4–5 to be skipped when the variable `count` is 1. Note that in test cases T16–T18, execution of Block 2 that contains return statement is followed by execution of Block 3. Normally, Block 3 should not be executed, but because this snipped code is being called inside a loop. Thus the traces of these test cases contain execution of Block 2 together with the following blocks.

On the basis of these spectra, we want to compute the suspiciousness score of each program element following Definition 3.1.

Definition 3.1
Suspiciousness Score

Consider a program $P = \{e_1, \ldots e_n\}$ and a set of program spectra $T = T_s \cup T_f$ for P, where P comprises of n elements $e_1, \ldots, e_n$ and T comprises of the spectra for correct executions T_s and the spectra for wrong executions T_f. We would like to measure the strength of the association between the *executions of each* e_i and program failures and assign this strength as the *suspiciousness score* of e_i denoted as *suspiciousness(e)*.

3.1.1. Tarantula. Jones and Harrold propose Tarantula [4] to rank program elements based on their suspiciousness scores. Intuitively, a program element is more suspicious if it appears in failed executions more frequently than in correct executions. Considering a program P and a test suite T, Table II introduces some common notations, which are used in the rest of the paper.

Tarantula's suspiciousness score for a program element e can be computed as follows:

$$suspiciousness(e) = \frac{\frac{n_f(e)}{n_f}}{\frac{n_s(e)}{n_s} + \frac{n_s(e)}{n_f}}$$

On the basis of block-hit program spectra shown in Figure 1, the suspiciousness score of Block 3 that contains the bug is $\frac{1/1}{3/3+1/1} = 0.5$. Block 1 has the same suspiciousness score. Interestingly, Block 2 receives the highest suspiciousness score: $\frac{1/1}{2/3+1/1} = 0.6$. Following the same calculation, Blocks 4 and 5 are not suspicious. There is no failure that executes these blocks, and hence, Tarantula returns a suspiciousness score of 0. Assuming developers inspect program elements one

DOI: 10.1002/smr

Table II. Some common notations.

Symbol	Definition
n	Total number of test cases in the test suite
$n(e)$	Number of test cases that executes a program element e
n_s	Number of test cases that pass
n_f	Number of test cases that fail
$n_s(e)$	Number of test cases that execute e and pass
$n_f(e)$	Number of test cases that execute e and fail

by one from the most suspicious to the least, the bug in Block 3 can be found after at most three blocks have been inspected.

3.1.2. Ochiai. Abreu *et al.* [16] proposed Ochiai, which assigns the suspiciousness score of a program element as follows:

$$suspiciousness(e) = \frac{n_f(e)}{\sqrt{n_f\left(n_f(e) + n_s(e)\right)}} = \frac{n_f(e)}{\sqrt{n_f n(e)}}$$

Similar to Tarantula, Ochiai considers an element more suspicious if it occurs more frequently in failed executions than in correct executions (the $\sqrt{\frac{n_f(e)}{n(e)}}$ part). Using the same example shown in Figure 1, Blocks 1 and 3 receive a suspiciousness score: $1/\sqrt{3*(1+0)} = 0.14$. Similar to Tarantula, Ochiai also returns Block 2 as the most suspicious block: $1/\sqrt{4*(1+0)} = 0.20$, whereas the remaining blocks are assigned suspiciousness scores of 0. As the case with Tarantula, by employing Ochiai, the bug can be found after three blocks have been inspected. In this study, we are interested to investigate other association measures that can possibly localize the bug earlier.

3.2. Dichotomous association

A common characteristic of the association measures evaluated in this paper is that they are all defined based on dichotomy matrices. The following are the necessary definitions.

Definition 3. 2.
(Dichotomy)

A *dichotomous outcome* is an outcome whose values could be split into two categories, for example, wrong or correct, executed or skipped, married or unmarried, etc. A *dichotomous variable* is a variable having a dichotomous outcome. A *dichotomy matrix* is a 2×2 matrix that tries to associate two dichotomous variables in the form of a 2×2 contingency table, which records the bivariate frequency distribution of the two variables.

An example of a dichotomy matrix *D(A,B)* relating variables *A* and *B* is shown in Table III. The value c_{00} corresponds to the number of observations in which the value of variable *A* equals to A_0 and the value of variable *B* equals to B_0. The values of the other three entries in the dichotomy matrix are similarly defined.

Base d on the concept of dichotomy matrix, we introduce dichotomous association in Definition 3.3.

Table III. An example of a dichotomy matrix. We refer to it as *D*(*A*,*B*).

	$A = A_0$	$A - A_1$
$B = B_0$	c_{00}	c_{01}
$B = B_1$	c_{10}	c_{11}

DOI: 10.1002/smr

Definition 3.3
(Dichotomous Association)

A *dichotomous association* is a special form of bivariate association [42], which measures the strength of association between two dichotomous variables, for example, application of a medical treatment and recovery from the disease, job satisfaction and productivity, and program element execution and program failure. The formulae for calculating dichotomous associations depend on the four entries in dichotomy matrices.

Given a dichotomy matrix relating two variables, two questions are often asked:

1. Is there a (dichotomous) association between the two variables?
2. How strong is the association between the two variables?

A common way to answer these two questions is to define a formula, referred to as an *association measure*, to calculate a score based on the four entries in a dichotomy matrix and consider the association exists (or is strong) if the score is beyond a particular threshold. We define association measure in Definition 3.4.

Definition 3.4
(Association Measure)

An association measure M of two variable A and B is a mathematical function of the four entries of a dichotomy matrix $D(A,B)$, and is denoted as $M(A,B,D(A,B))$ or simply $M(A,B,D)$ if it is clear from the context.

In fault localization, we could produce a dichotomy matrix that relates the executions of a program element with program failures. Consider a program element e. For each test case, a trace is generated when a subject program is executed on the test case. Some traces execute e, others do not. Some traces correspond to failures, others correspond to correct executions. After the test cases are run, a dichotomy matrix as shown in Table IV is produced for every program element e.

The notation $\overline{e}$ means e is not executed, and other notations are defined in Table II. Thus, $n_s(\overline{e})$ is the number of test cases that do not execute e and pass, and $n_f(\overline{e})$ is the number of test cases that do not execute e and fail.

Considering variables E and F to represent a program element being executed and a program failure occurs respectively, we are interested to compute $M(E, F, De(E, F))$ (i.e., the association between the execution of e and a failure), where $De(E, F)$ represents the dichotomy matrix of the two variables E and F for a program element e. The formulae of the 40 association measures are given in Section 4.

4. ASSOCIATION MEASURES

In this section, we first describe how a dichotomy matrix can be constructed. Next, we present how the 40 association measures can be computed from a dichotomy matrix. Finally, we give an example how we sort program elements for inspection using the association measures.

4.1. Constructing a dichotomy matrix

Dichotomy matrix construction requires programs instrumentation that could support collection of program execution traces and a test oracle. In this paper, we instrument all buggy programs in our dataset in basic block level. We manually instrument for C programs and use Cobertura[§] to instrument Java programs. In order to construct dichotomy matrix, we collect the program execution traces, which contain information about which basic blocks are executed by each test case. The test oracle would give us information on whether a test case fails or passes.

Next, for each basic block, we construct a dichotomy matrix by counting the number of times the basic block is executed by the passing test cases which is denoted as $n_s(e)$, number of times the basic block is not executed by the passing test cases which is denoted as $n_s(\overline{e})$, number of times the basic block is executed

[§]http://cobertura.sourceforge.net/

DOI: 10.1002/smr

Table IV. Dichotomy matrix for fault localization.

	e Executed	*e* Not Executed
Test Passed	$n_s(e)$	$n_s(\overline{e})$
Test Failed	$n_f(e)$	$n_s(\overline{e})$

by failing test cases which is denoted as $n_f(e)$, and number of times the basic block is not executed by failing test cases which is denoted as $n_f(\overline{e})$. The information in this dichotomy matrix is then used as an input to the association measures to calculate how likely the corresponding basic block contains a bug.

4.2. Association measures

The 40 association measures that we consider are as follows: ϕ-Coefficient [42], Odds Ratio [18], Yule's Q [19], Yule's Y [20], Kappa [48], J-Measure [49], Gini Index [50], Support [43], Confidence [43], Clark and Boswell's Laplace accuracy [51], Conviction [52], Interest [52], Cosine [44], Piatetsky-Shapiro's Leverage [53], Certainty Factor [54], Added Value [44], Collective Strength [55], Jaccard [56], Klosgen [57], Information Gain [58, 59], Coverage [45, 60], Accuracy [45, 60], Leverage [45, 60], Relative Risk [45, 60], Interestingness Weighting Dependency [45, 60], Goodman and Kruskal [44, 45, 60, 61], Normalized Mutual Information [44, 45, 60], One-Way Support [45, 60], Two-Way Support [45, 60], Two-Way Support Variation [45, 60], Loevinger [45], Sebag-Schoenauer [45], Least Contradiction [45], Odd Multiplier [45], Example and Counterexample Rate [45], Zhang [45], Sorensen-Dice [62, 63], Anderberg [64], Simple-Matching [65], Rogers and Tanimoto [66], and Ochiai II [67].

The mathematical formulae of association measures are defined in terms of probabilities, instead of frequencies, but we can substitute frequencies recorded in dichotomous matrices for probabilities during actual calculations. For easier reference, the formulae for those *A|B* related term are shown in Table V. The mathematical formulae for calculating the association measures, including Tarantula and Ochiai, are given in Tables VI and VII. The ranges of values that these association measures can take are given in Tables VIII and IX.

4.3. From association to suspiciousness

From Section 3, the suspiciousness score for a program element *e* with dichotomy matrix D_e is defined as the strength of the association between the executions of *e* with failures (i.e., *M(E, F,De))*. *M* refers to one of the 40 association measures presented in Section 4.2.

Next, we illustrate how an association measure could be used to rank program elements for inspection. Using the example in Figure 1, we observe that there are five blocks: Blocks 1, 2, 3, 4, and 5. The bug resides at the `if` statement in block 3. The suspiciousness score of block 3 is determined by the association strength between the executions of block 3 and failures. For example, by using one of the association measures, for example, Coverage, blocks 3 and 1 receive the highest suspiciousness score, that is, 1, followed by block 2 whose suspiciousness score is 0.75. The suspiciousness scores of blocks 4 and 5 are 0.5. This particular measure can rank the block containing the bug such that no other block receives a score higher than it. However, Tarantula and Ochiai are unable to do so—Sections 3.1.1 and 3.1.2.

Table V. Preliminary definitions and equations.

$P(A) = \frac{n_f(e) + n_s(e)}{n}$	$P(A,B) = \frac{n_f(e)}{n}$	$P(B\|A) = \frac{P(A,B)}{P(A)}$
$P(\overline{A}) = \frac{n_f(\overline{e}) + n_s(\overline{e})}{n}$	$P(\overline{A},B) = \frac{n_f(\overline{e})}{n}$	$P(A\|B) = \frac{P(A,B)}{P(B)}$
$P(B) = \frac{n_f(e) + n_f(\overline{e})}{n}$	$P(A,\overline{B}) = \frac{n_s(e)}{n}$	
$P(\overline{B}) = \frac{n_s(e) + n_s(\overline{e})}{n}$	$P(\overline{A},\overline{B}) = \frac{n_s(\overline{e})}{n}$	

DOI: 10.1002/smr

Table VI. Definitions of association measures, Tarantula, and Ochiai (Part I).

Name	Formula
ϕ-Coefficient (M_1)	$\frac{P(A,B)-P(A)P(B)}{\sqrt{P(A)P(B)(1-P(A))(1-P(B))}}$
Odds Ratio (M_2)	$\frac{P(A,B)P(\overline{A},\overline{B})}{P(A,\overline{B})P(\overline{A},B)}$
Yule's Q (M_3)	$\frac{P(A,B)P(\overline{A,B})-P(A,\overline{B})P(\overline{A},B)}{P(A,B)P(\overline{A,B})-P(A,\overline{B})P(\overline{A},B)} = \frac{\alpha-1}{\alpha+1}$
Yule's Y (M_4)	$\frac{\sqrt{P(A,B)P(\overline{A,B})}-\sqrt{P(A,\overline{B})P(\overline{A},B)}}{\sqrt{P(A,B)P(\overline{A,B})}-\sqrt{P(A,\overline{B})P(\overline{A},B)}} = \frac{\sqrt{\alpha}-1}{\sqrt{\alpha}+1}$
Kappa(M_5)	$\frac{P(A,B)+P(\overline{A},\overline{B})-P(A)P(B)-P(\overline{A})P(\overline{B})}{1-P(A)P(B)-P(\overline{A})P(\overline{B})}$
J-Measure(M_6)	$\max P(A,B)\log\left(\frac{P(B\|A)}{P(B)}\right) + P(A\overline{B})\log\left(\frac{P(\overline{B}\|A)}{P(\overline{B})}\right),$ $P(A,B)\log\left(\frac{P(A\|B)}{P(A)}\right) + P(\overline{A}B)\log\left(\frac{P(\overline{A}B)}{P(B)}\right),$
Gini Index (M_7)	$\max(P(A)\left[P(B\|A)^2 + P(\overline{B}\|A)^2\right] + P(\overline{A})[P(B\|\overline{A})^2 +$ $P(\overline{B}\|\overline{A})^2] - P(B)^2 - P(\overline{B})^2, P(B)\left[P(A\|B)^2 + P(\overline{A}\|B)^2\right] +$ $P(\overline{B})\left[P(A\|\overline{B})^2 + P(\overline{A}\|\overline{B})^2\right] - P(\overline{A})^2 - P(A)^2)$
Support (M_8)	$P(A,B)$
Confidence (M_9)	$max(P(B\|A), P(A\|B))$
Laplace (M_{10})	$max\left(\frac{P(A,B)+1}{P(A)+2}, \frac{P(A,B)+1}{P(B)+2}\right)$
Conviction (M_{11})	$max\left(\frac{P(A)P(\overline{B})}{P(A\overline{B})}, \frac{P(B)P(\overline{A})}{P(B\overline{A})}\right)$
Interest (M_{12})	$\frac{P(A,B)}{P(A)P(B)}$
Piatetsky-Shapiro's (M_{13})	$P(A,B) - P(A)P(B)$
Certainty Factor (M_{14})	$max\left(\frac{P(B\|A)-P(B)}{1-P(B)}, \frac{P(A\|B)-P(A)}{1-P(A)}\right)$
Added Value (M_{15})	$max(P(B\|A) - P(B), P(A\|B) - P(A))$
Collective Strength (M_{16})	$\frac{P(A,B)+P(\overline{A},\overline{B})}{P(A)P(B)+P(\overline{A})P(\overline{B})} \times \frac{1-P(A)P(B)-P(\overline{A})P(\overline{B})}{1-P(A,B)-P(\overline{AB})}$
Jaccard (M_{17})	$\frac{P(A,B)}{P(A)+P(B)-P(A,B)}$
Klosgen (M_{18})	$\sqrt{P(A,B)}max(P(B\|A) - P(B), P(A\|B) - P(A))$
Information Gain (M_{19})	$\left(-P(B)logP(B) - P(\overline{B})logP(\overline{B})\right) - (P(A)$ $\times(-P(B\|A)logP(B\|A)) - P(\overline{B}\|A)logP(\overline{B}\|A) - P(\overline{A})$ $\times\left(-P(B\|\overline{A})logP(B\|\overline{A})\right) - P(\overline{B}\|\overline{A})logP(\overline{B}\|\overline{A})))$

5. EMPIRICAL EVALUATION

In this section, we describe our datasets, our evaluation metrics, and evaluation results.

5.1. Datasets

Based on availability of the programs listed in Table I and used in previous studies, we choose our subject programs as follows. We analyze different programs from Siemens test suite [68]. Siemens programs are injected with realistic bugs and often analyzed for fault localization studies

DOI: 10.1002/smr

Table VII. Definitions of association measures, Tarantula, and Ochiai (Part II).

Name	Formula
Coverage (M_{20})	$P(A)$
Accuracy (M_{21})	$P(A,B)+P(\overline{A},\overline{B})$
Leverage (M_{22})	$P(B\|A)-P(A)P(B)$
Relative Risk (M_{23})	$P(B\|A)/P(B\|\overline{A})$
Interestingness Weighting Dependency (M_{24})	$\left(\left(\frac{P(A,B)}{P(A)P(B)}\right)^{k}-1\right)P(P,B)^{m}$ where k,m are coefficients of dependency and generality respectively weighting the relative importance of the two factors.
Goodman and Kruskal (M_{25})	$\frac{\sum_i \max_j P(A_i-B_j)+\sum_j \max_i P(A_i,,B_j)-\max_i P(A_i)-\max_j P(B_j}{2-\max_i P(A_i)-\max_j P(B_j))}$
Normalized Mutual Information (M_{26})	$\sum_i\sum_j P(A_i,B_j)\log_2\frac{P(A_iB_j)}{P(A_i)P(B_j)}/\{-\sum_i P(A_i)\log_2 P(A_i)\}$
One-Way Support (M_{27})	$P(B\|A)log_2\frac{P(A,B)}{(P(A)P(B))}$
Two-Way Support (M_{28})	$P(A,B)log_2\frac{P(A,B)}{(P(A)P(B))}$
Two-Way Support Variation (M_{29})	$P(A,B)\log_2\frac{P(A,B)}{(P(A)P(B))}+P(A,\overline{B})\log_2\frac{P(A,\overline{B})}{(P(A)P(B))}+$ $P(\overline{A},B)\log_2\frac{P(\overline{A},B)}{(P(\overline{A})P(B))}+P(\overline{A}\overline{B})\log_2\frac{P(\overline{A},\overline{B})}{(P(\overline{A})P(\overline{B}))}+$
Loevinger (M_{30})	$1-\frac{P(A)P(\overline{B})}{P(A,\overline{B})}$
Sebag-Schoenauer (M_{31})	$\frac{P(A,B)}{P(A,\overline{B})}$
Least Contradiction (M_{32})	$\frac{P(A,B)-P(A,\overline{B})}{P(B)}$
Odd Multiplier (M_{33})	$\frac{P(A,B)P(\overline{B})}{P(B)P(A\overline{B})}$
Example and Counterexample Rate (M_{34})	$1-\frac{P(A,\overline{B})}{P(A,B)}$
Zhang (M_{35})	$\frac{P(A,B)-P(A)P(B)}{max(P(A,B)P(\overline{B}),P(B)P(A,\overline{B})}$
Sorensen-Dice (M_{36})	$\frac{2P(A,B)}{2P(A,B)+P(\overline{A},B)+P(A,\overline{B})}$
Anderberg (M_{37})	$\frac{P(A,B)}{P(A,B)+2(P(A,B)+P(A,\overline{B}))}$
Simple-Matching (M_{38})	$P(A,B)+P(\overline{A},\overline{B})$
Rogers and Tanimoto (M_{39})	$\frac{P(A,B)+P(\overline{A},\overline{B})}{P(A,B)+P(\overline{A},\overline{B})+2(P(\overline{A},B)+P(A,\overline{B}))}$
Ochiai II (M_{40})	$\frac{P(A,B)+P(\overline{A},\overline{B})}{\sqrt{(P(A,B)+P(\overline{A},B))(P(A,B)+P(A,\overline{B}))(P(\overline{A},\overline{B})+P(\overline{A},B))(P(\overline{A},\overline{B})+P(A,\overline{B}))}}$
Tarantula	$\frac{\frac{P(A,B)}{P(B)}}{\frac{P(A,\overline{B})}{P(\overline{B})}+\frac{P(A,B)}{P(B)}}$
Ochiai	$\frac{P(A,B)}{\sqrt{P(A)P(B)}}$

[4, 5, 8, 12, 13, 16, 17, 25, 28, 30, 33, 35–37]. We also analyze other three real programs from SIR [22] namely: Space [4, 15, 17, 21], NanoXML [25], and XML-Security [25]. Space is written in C, whereas NanoXML and XML-Security are written in Java. The average lines of code for various versions of Space, NanoXML, and XML-Security are 6 218, 4 223, and 21 275, respectively.

Siemens test suite was originally used for research in test coverage adequacy and was developed by Siemens Corporation Research. We use the variant provided at www.cc.gatech.edu/aristotle/Tools/subjects/. Each program contains many different versions where each version has one bug. These bugs comprise a wide array of realistic bugs. The Siemens test suite comes with seven programs:

DOI: 10.1002/smr

Table VIII. Value ranges of the association measures, Tarantula, and Ochiai (Part I).

Name	Range	No	Perfect
ϕ-Coefficient (M_1)	–1…0…1	10	1
Odds Ratio (M_2)	$0 \dots 1 \dots \infty$	1	∞
Yule's Q (M_3)	1…0…1	0	1
Yule's Y (M_4)	1…0…1	0	1
Kappa (M_5)	$-1 \dots 0 \dots 1$	0	1
J-Measure (M_6)	$0 \dots 1$	0	1
Gini Index (M_7)	$0 \dots 1$	0	1
Support (M_8)	$0 \dots 1$	0	1
Confidence (M_9)	$0 \dots 1$	0	1
Laplace (M_{10})	$0 \dots 1$	0	1
Conviction (M_{11})	$0.5 \dots 1 \dots \infty$	1	∞
Interest (M_{12})	$0 \dots 1 \dots \infty$	1	∞
Piatetsky-Shapiro's (M_{13})	$-0.25 \dots 1 \dots 0.25$	0	0.25
Certainty Factor (M_{14})	$-1 \dots 0 \dots 1$	0	1
Added Value (M_{15})	$-0.5 \dots 0 \dots 1$	0	1
Collective Strength (M_{16})	$0 \dots 1 \dots \infty$	1	∞
Jaccard (M_{17})	0…1	0	1
Klosgen (M_{18})	$\left(\frac{2}{\sqrt{3}}-1\right)^{1/2}\left[2-\sqrt{3}-\frac{1}{\sqrt{3}}\right]\dots 0\dots\frac{2}{3\sqrt{3}}$	0	$\frac{2}{3\sqrt{3}}$
Information Gain (M_{19})	0…1	0	1

The third and fourth columns give the values corresponding to no association and perfect association respectively. Values of measures with ranges in the format of $a \dots b$ (e.g., J-Measure ($0 \dots 1$), etc.) indicate positive associations with failures. Values of measures with ranges in the format of $a \dots b \dots c$ (e.g., ϕ-coefficient ($-1..0..1$), etc.) indicate positive associations with failures (if they are between b and c), or negative associations with failures (if they are between a and b). Values closer to a imply stronger associations with passing executions.

Table IX. Value ranges of the association measures, Tarantula, and Ochiai [Part II].

Name	Range	No	Perfect
Coverage (M_{20})	0…1	0	1
Accuracy (M_{21})	0…1	0	1
Leverage (M_{22})	$-1 \dots 0 \dots 1$	0	1
Relative Risk (M_{23})	$0 \dots 1 \dots \infty$	1	∞
Interestingness Weighting Dependency (M_{24})	–1…0…1	0	1
Goodman and Kruskal (M_{25})	$0 \dots 1$	0	1
Normalized Mutual Information (M_{26})	$0 \dots 1$	0	1
One-Way Support (M_{27})	$-1 \dots 0 \dots 1$	0	1
Two-Way Support (M_{28})	$-1 \dots 0 \dots 1$	0	1
Two-Way Support Variation (M_{29})	$0 \dots 1$	0	1
Loevinger (M_{30})	$-1 \dots 0 \dots 1$	0	1
Sebag-Schoenauer (M_{31})	$0 \dots 1 \dots \infty$	1	∞
Least Contradiction (M_{32})	$0 \dots 1$	0	1
Odd Multiplier (M_{33})	$0 \dots 1 \dots \infty$	1	∞
Example and Counterexample Rate (M_{34})	$-\!- \infty \dots 0 \dots 1$	0	1
Zhang (M_{35})	$-1 \dots 0 \dots 1$	0	1
Sorensen-Dice (M_{36})	$0 \dots 1$	0	1
Anderberg (M_{37})	$0 \dots 1$	0	1
Simple-Matching (M_{38})	$0 \dots 1$	0	1
Rogers and Tanimoto (M_{39})	$0 \dots 1$	0	1
Ochiai II (M_{40})	$0 \dots 1$	0	1
Tarantula	$0 \dots 1$	0	1
Ochiai	$0\sqrt{P(A,B)\dots 1}$	0	1

print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info. The total number of buggy versions is 132, as shown in Table X. We manually instrumented the buggy versions at basic block level. Because our instrumentation cannot reach the bugs that reside in variable declarations, we exclude versions that contain this type of bugs, that is, versions 6, 10, 19, and 21 of tot_info

DOI: 10.1002/smr

Table X. Dataset descriptions

Dataset	LOC	Language	Num. of Faulty Version	Num. of Test Cases
print_token	478	C	5	4130
print_token2	399	C	10	4115
replace	512	C	31	5542
schedule	292	C	9	2650
schedule2	301	C	9	2710
tcas	141	C	36	1608
tot_info	440	C	19	1051
space	6218	C	35	13,585
NanoXML v1	3497	Java	6	214
NanoXML v2	4007	Java	7	214
NanoXML v3	4608	Java	9	216
NanoXML v5	4782	Java	8	216
XML security v1	21 613	Java	6	92
XML security v2	22 318	Java	6	94
XML security v3	19 895	Java	4	84

dataset, version 12 of replace dataset, and versions 13, 14, 15, 36, and 38 of tcas dataset. We exclude versions 4 and 6 of print_token because they are identical with the original version. We also exclude version 9 of schedule2 because there is no test case that results in a failure. Thus, in total, we use 119 buggy versions from the Siemens test suite.

Space is an interpreter for array definition language used by European Space Agency. We analyze all 35 faulty versions of Space downloaded from SIR. NanoXML is a utility for parsing XML. SIR contains five versions of NanoXML. We exclude NanoXML_v4 because there is no buggy version. For each version, SIR provides a few bugs. In total, there are 32 buggy versions for NanoXML_v1, NanoXML_v2, NanoXML_v3, and NanoXML_v5, and we analyze 30 of them. We exclude two buggy versions because there is no test case that results in a failure. XML-Security is a Java library that supports digital signature and encryption. SIR contains three versions of XML-Security. Again, for each version, SIR provides a few bugs. In total, there are 52 buggy versions for $XMLSec\_v1, XMLSec\_v2, and\ XMLSec\_v3$. We exclude 16 buggy versions because there is no test case that results in a failure. Thus, the total number of buggy versions that we analyze for the three programs are 81.

Table X provides the number of lines, the programming language in which the program is written, the number of faulty versions, and the number of test cases of each subject program.

5.2. *Evaluation metrics*

We use 40 association measures, Tarantula, and Ochiai to rank program elements based on their suspiciousness. Developers could then use this list to investigate the more suspicious program elements first. We assume that developers would be able to identify the buggy program element when they inspect it.

We consider two commonly used evaluation metrics: average percentage of code inspected to find all bugs, and proportion of bugs found when a given proportion of the code is inspected. We describe these two metrics in the following paragraphs.

5.2.1. Percentage of Code Inspected. We evaluate the performance of the measures by the number of elements that are ranked as high or higher than the program element containing the fault/the bug. For a suspiciousness score to be effective, buggy program elements should have a relatively larger value of suspiciousness scores than the non-buggy elements.

When a buggy program element has the same suspiciousness score with several other elements, the largest rank of the elements that has this suspiciousness score is used as the rank of the buggy element. For example, consider the case where the two highest suspiciousness scores are 0.92 and 0.91, two elements have suspiciousness scores of 0.92, and three elements have suspiciousness scores of 0.91. If a buggy element is given the suspiciousness score of 0.91, then the rank of this buggy element is

DOI: 10.1002/smr

5, instead of 3. Because we do not know how the programmer will traverse elements that have the same suspiciousness score, we use the worst case scenario where the programmer inspects all elements having the same score. In the situation when a buggy version contains multiple bugs or one single bug that involves multiple program elements, we use the largest rank among the buggy elements as it is the worst case rank that represents scenarios when programmers would like to find all the buggy elements by using the given list of most suspicious program elements.

Ranking program elements by using suspiciousness score is useful to evaluate the accuracy of a fault-localization approach. However, it may not be representative enough to know how programmers really locate the bugs with their expertise, as this ranking approach does not provide the context of the bugs to help programmers in understanding the root cause of the bugs, as the study by Parnin and Orso has shown [40]. It is an interesting future study to augment suspiciousness ranking with some additional information to more effectively guide programmers to locate all buggy program elements.

Suspiciousness measures that rank buggy elements first are more effective than those that rank them last. We then use this rank to compute the percentage of program elements that need to be inspected to find the buggy elements by the formula:

$$\frac{largest\ \ rank\ \ among\ \ the\ \ buggy\ \ elements}{total\ \ elements}$$

In our experiment, we thus choose to use basic block as the granularity of the elements and the previous percentage is applied as the first accuracy criterion of the association measures. The accuracy of a particular measure to localize bug in all buggy versions of our datasets is then evaluated by calculating the overall mean of all percentages of code inspected for the measure, which is the average of the percentages of code inspected of the measure for all buggy versions in our datasets. The smaller the overall mean, the more accurate the measure in localizing bug. However, the overall mean might not necessarily reflect that the measure would localize bugs with the same accuracy for all buggy versions. A measure could have a good accuracy in localizing one bug, but might not have a good accuracy in localizing other bugs. Thus, we also calculate the overall standard deviation of a measure to evaluate the variance of percentages of code inspected of the measure for all buggy versions. The smaller the overall standard deviation, the better the overall mean reflects the accuracy of the measure because of less variation of percentage of code inspected for all buggy versions.

5.2.2. Proportion of Bugs Localized. Next, we calculate the proportion of bugs that could be localized assuming that the developers are only willing to investigate a given proportion of code. To compute this measure, we vary the proportion of code that the developers are willing to inspect and for each proportion we compute the proportion of bugs that can be localized.

5.3. Evaluation results

We describe the accuracy of the association measures in comparison with Ochiai and Tarantula in the following paragraphs.

5.3.1. Percentage of Code Inspected. The overall means and standard deviations of percentage of code inspected of the 40 association measures along with those of Ochiai and Tarantula for all subject programs are shown in Table XI. The smallest mean is 25.24%, which is achieved by Klosgen (M_{18}). Ochiai and Tarantula achieve 25.45 and 27.13%, respectively. Other measures that have similar accuracy to Klosgen (M_{18}) and Ochiai (i.e., having a mean in the range of 25.66–26.24%) are Collective Strength (M_16), Normalized Mutual Information (M_{26}), ϕ-Coefficient (M_1), Added Value (M_{15}), Two-Way Support (M_{28}), and Interestingness Weighting Dependency (M_{24}). There are 11 measures that have similar accuracy as Tarantula (i.e., having a mean between 26.60–27.12%), that is, J-Measure (M_6), Information Gain M_{19}, Two-Way Support Variation (M_{29}), Example and Counterexample Rate (M_{34}), Confidence (M_9), Kappa (M_5), Sebag (M_{31}), Odd Multiplier (M_{33}), Interest (M_{12}), Zhang

DOI: 10.1002/smr

Table XI. Overall mean and standard deviation (in parentheses) of accuracy values (smaller the better)

Association Measures	Mean (%)	SD (%)	Association Measures	Mean (%)	SD (%)
Klosgen (M_{18})	25.24	27.13	Anderberg(M_{37})	27.60	27.30
Ochiai	25.45	26.54	Sorensen-Dice(M_{36})	27.61	27.28
Collective Strength (M_{16})	25.66	26.86	Ochiai II(M_{40})	28.23	28.88
Normalized Mutual Information(M_{26})	25.68	26.38	Gini Index(M_7)	28.23	28.43
φ-Coefficient(M_1)	25.73	27.07	Leverage(M_{22})	29.46	28.12
Added Value(M_{15})	25.74	27.50	Least Contradiction(M_{32})	33.42	28.82
Two-Way Support(M_{29})	26.03	27.25	Rogers and Tanimoto(M_{39})	33.47	28.85
Interestingness Weighting Dependency(M_{24})	26.24	27.42	Accuracy(M_{21})	33.68	28.81
J-Measure(M_6)	26.60	27.51	Simple-Matching(M_{38})	33.68	28.81
Information Gain(M_{19})	26.73	27.53	Odds Ratio(M_2)	41.61	21.34
Two-Way Support Variation(M_{29})	26.73	27.53	Yule's Q (M_3)	41.68	21.45
Example and Counterexample Rate(M_{34})	26.96	27.12	Yule's Y(M_4)	41.69	21.44
Confidence(M_9)	26.99	27.25	Certainty Factor(M_{14})	41.73	21.26
Kappa(M_5)	27.04	27.21	Conviction(M_{11})	41.73	21.27
Sebag(M_{31})	27.09	27.18	Relative Risk(M_{23})	44.82	22.01
Odd Multiplier(M_{33})	27.09	27.20	Laplace(M_{10})	45.34	21.63
Interest(M_{12})	27.10	27.19	Support(M_8)	45.44	21.65
Zhang(M_{35})	27.11	27.17	Goodman and Kruskal(M_{25})	45.74	37.82
One-Way Support(M_{27})	27.12	27.26	Coverage(M_{20})	47.02	24.37
Tarantula	27.13	27.17	Piatetsky-Shapiro's(M_{13})	57.37	24.50
Jaccard(M_{17})	27.47	27.36	Loevinger(M_{30})	57.57	25.35

DOI: 10.1002/smr

(M_{35}), and one-way support (M_{27}). We notice that most of the association measures have similar standard deviations of around 20% (ranging from 21%–29%), except for Goodman and Kruskal (M_{25}), which has the highest standard deviation among all measures, that is, 38%.

In our evaluation dataset, we have eight C programs and seven Java programs, the accuracy of an association measure in localizing bugs for different programs might be different. We are interested to evaluate the accuracy of the association measures in localizing bugs for each program. Thus, we also calculate the mean and standard deviation of the measures for buggy versions in each program. Tables XII and XIII show the detail of the accuracy values (i.e., mean and standard deviation) of each measure for each C program, whereas Tables XIV and XV show the detail for each of Java program. Based on the results, the measures have different accuracy for different program and different programming languages. The measures generally require more than 10% of percentage of code to be inspected in order to localize bugs for five C programs (i.e., print_tokens, schedule2, replace, tcas, and tot_info) and the Java programs. For *schedule* buggy versions, more than half of the measures could localize bugs in these versions when less than 10% of code is inspected.

We also perform statistical tests for each pair of measures including Tarantula and Ochiai using Wilcoxon signed rank test [69] at 0.05 statistical significance threshold to see if some measures are statistically significantly better than others. We use this statistical test because it does not assume that the data follows normal distribution. We visualize the statistical significance relationship as a partial order in Figure 2. A link from A to B in the partial order denotes that A is statistically significantly better than B. The relationships expressed in the partial order are transitive: if A outperforms B, and B outperforms C, then A outperforms C too. To reduce the number of links in the partial order, we omit links that could be captured by this transitivity property.

It is interesting to note that Klosgen (M_{18}) and Normalized Mutual Information (M_{26}) perform comparably with Ochiai. Klosgen (M_{18}), Normalized Mutual Information (M_{26}), and Ochiai are significantly better than Tarantula. We also notice that seven other measures also perform significantly better than Tarantula. These are: ϕ – Coefficient (M_1), Added Value (M_{15}), Collective Strength (M_{16}), Two-Way Support (M_{28}), Interestingness Weighting Dependency (M_{24}), Example and Counterexample Rate (M_{34}), and Kappa (M_5). Measures that perform comparably with Tarantula are Interest (M_{12}), One-Way Support (M_{27}), Sebag (M_{31}), Odds Multiplier (M_{33}), and Zhang (M_{35}). On the other hand, Piatetsky-Shapiro's (M_{13}), and Loevinger (M_{30}) perform worse than other measures for fault localization.

5.3.2. Proportion of Bugs Localized. We also plot the curve showing the proportion of codes that are investigated (x-axis) versus the proportion of bugs localized (y-axis) for all dataset. We split the large graphs into several smaller graphs so that measures that have similar accuracies would be grouped together, as shown in Figures 3 – 8. For each graph, we compare a number of association measures with Tarantula and Ochiai.

Association measures included in Figure 3 perform better than Ochiai and Tarantula or as good as Ochiai. When only 10% of program elements are inspected, Tarantula and Ochiai could localize 39 and 45% of the bugs. Klosgen (M_{18}) and Added Value (M_{15}) could localize more bugs than Tarantula and Ochiai—they could localize 47 and 48% of the bugs, respectively. Two-Way Support (M_{28}) could localize the same proportion of bugs as Ochiai.

Figures 5 and 6 show association measures that perform better than Tarantula but not as good as Ochiai when only 10% of program elements are inspected. The measures are φ-Coefficient (M_1), J-Measure (M_6), Gini Index (M_7), Collective Strength (M_{16}), Jaccard (M_{17}), Information Gain (M_{19}), Interestingness Weighting Dependency (M_{24}), Normalized Mutual Information (M_{26}), Loevinger(M_{29}), Sorensen-Dice (M_{36}), Anderberg (M_{37}), and Ochiai II (M_{40}).

Measures that perform similar to Tarantula are shown in Figure 4. They localize 39–40% of the bugs when 10% of program elements are inspected. The association measures included in Figures 7 and 8 perform worse than Tarantula and Ochiai.

5.4. Effectiveness for various programming languages

We are interested in evaluating the accuracy of measures for different programming language (C and Java). We find that for different programming language, measures could perform differently. Based

DOI: 10.1002/smr

Table XII. Detailed means and standard deviations (in parentheses) of percentages of code inspected to find all bugs in the C programs (Part I).

	Programs							
Association Measures	print_token %(%)	print_token2 %(%)	schedule %(%)	schedule2 %(%)	replace %(%)	tcas %(%)	tot_info %(%)	space %(%)
ϕ-Coefficient	21(25)	13(17)	9(15)	47(24)	13(19)	56(27)	19(15)	*10(24)
Odds Ratio	43(19)	44(2)	49(28)	53(7)	34(16)	64(15)	49(20)	23(20)
Yule's *Q*	43(19)	44(3)	48(28)	53(7)	34(16)	64(15)	49(20)	23(20)
Yule's *Y*	44(18)	44(3)	48(28)	53(7)	34(16)	64(15)	49(20)	23(20)
Kappa	28(26)	16(20)	7(6)	51(24)	16(21)	56(27)	21(16)	*10(24)
J-Measure	22(27)	11(13)	14(19)	49(23)	*12(18)	56(29)	18(15)	12(25)
Gini index	26(32)	16(22)	12(12)	58(26)	13(19)	58(29)	24(19)	12(25)
Support	43(17)	45(3)	50(30)	56(8)	34(16)	67(14)	51(20)	26(21)
Confidence	29(26)	13(18)	7(6)	51(24)	16(21)	57(27)	23(16)	11(24)
Laplace	43(17)	45(3)	50(29)	56(8)	35(16)	67(14)	50(20)	25(20)
Conviction	42(16)	44(3)	48(28)	53(7)	34(16)	64(15)	49(20)	24(20)
Interest	29(26)	17(20)	7(6)	51(24)	16(21)	56(27)	22(16)	11(24)
Piatetsky-Shapiro's	61(32)	77(16)	71(29)	68(28)	64(22)	66(26)	63(28)	50(18)
Certainty Factor	42(16)	44(3)	48(28)	53(7)	34(16)	64(15)	49(20)	24(20)
Added Value	23(28)	14(19)	7(10)	50(25)	13(20)	56(28)	18(15)	11(24)
Collective Strength	24(31)	13(16)	11(20)	44(23)	13(18)	55(27)	18(15)	*10(24)
Jaccard	26(26)	12(18)	7(6)	50(24)	15(21)	56(27)	21(15)	11(24)
Klosgen	21(25)	11(15)	8(12)	48(24)	*12(19)	55(28)	*17(13)	11(24)
Information Gain	22(27)	11(14)	14(19)	49(23)	*12(18)	56(29)	19(16)	12(25)
Coverage	63(26)	63(20)	69(30)	*36(26)	53(23)	*47(25)	51(28)	40(24)
Accuracy	38(26)	28(22)	11(5)	60(24)	27(22)	60(29)	37(29)	15(26)
Leverage	30(28)	20(22)	7(4)	55(26)	17(20)	59(28)	28(23)	14(25)
Relative Risk	44(19)	45(3)	50(28)	56(8)	35(16)	68(14)	50(20)	23(20)
Int. Weighting Dependency	24(28)	15(20)	8(9)	50(24)	14(20)	56(27)	21(16)	*10(24)
Goodman and Kruskal	35(44)	30(44)	67(38)	83(9)	40(40)	70(30)	66(41)	19(31)
Normalized Mutual Info.	23(31)	*9(11)	16(27)	41(20)	*12(17)	54(27)	*17(13)	11(24)
One-Way Support	29(26)	17(20)	7(6)	51(24)	16(20)	57(27)	22(16)	11(24)
Two-Way Support	23(28)	14(19)	8(10)	50(25)	13(18)	56(28)	20(15)	*10(24)
Two-Way Support Variation	22(27)	11(14)	14(19)	49(23)	*12(18)	56(29)	19(16)	12(25)
Loevinger	71(26)	78(21)	74(30)	44(26)	65(22)	53(23)	70(26)	61(25)
Sebag	29(26)	17(20)	7(6)	51(24)	16(21)	56(27)	22(16)	11(24)
Least Contradiction	34(30)	28(22)	11(5)	59(24)	26(21)	60(29)	36(29)	15(26)
Odd Multiplier	29(26)	17(20)	7(6)	51(24)	16(21)	56(27)	22(16)	11(24)
Example and Counter.	29(26)	17(20)	7(6)	51(24)	16(21)	56(27)	22(16)	11(24)
Zhang	29(26)	17(20)	7(6)	51(24)	16(21)	56(27)	22(16)	11(24)

*marks the measure(s) with the lowest mean and the underline marks measures that have a mean within 10%.

DOI: 10.1002/smr

Table XIII. Detailed means and standard deviations (in parentheses) of percentages of code inspected to find all bugs in the C programs (Part II).

	Programs							
Association Measures	print_token %(%)	print_token2 %(%)	schedule %(%)	schedule2 %(%)	replace %(%)	tcas %(%)	tot_info %(%)	space %(%)
Sorensen-Dice	27(26)	16(20)	7(6)	50(24)	15(21)	56(27)	20(14)	11(24)
Anderberg	26(26)	16(20)	7(6)	51(24)	16(21)	56(27)	20(14)	11(25)
Simple-Matching	38(26)	28(22)	11(5)	60(24)	27(22)	60(29)	37(29)	15(26)
Rogers and Tanimoto	34(30)	28(22)	11(5)	60(24)	27(22)	60(29)	36(29)	15(26)
Ochiai II	27(32)	16(21)	*6(4)	57(26)	15(21)	57(28)	28(27)	11(24)
Tarantula	29(26)	17(20)	7(6)	51(24)	16(21)	56(27)	22(16)	11(24)
Ochiai	*20(29)	*9(10)	13(28)	42(22)	*12(19)	53(26)	*17(12)	11(24)

*marks the measure(s) with the lowest mean and the underline marks measures thathave a mean within 10%.

DOI: 10.1002/smr

Table XIV. Detailed means and standard deviations (in parentheses) of percentages of code inspected to find all bugs in the Java programs (Part I).

	Programs						
Association Measures	Nano_v1 %(%)	Nano_v2 %(%)	Nano_v3 %(%)	Nano_v5 %(%)	XML-sec_v1 %(%)	XML-sec_v2 %(%)	XML-sec_v3 %(%)
ϕ-Coefficient	*21(28)	32(25)	*31(30)	20(15)	29(0)	31(0)	40(0)
Odds Ratio	45(18)	43(18)	44(19)	23(10)	29(0)	31(0)	40(0)
Yule's Q	44(18)	43(18)	44(19)	25(14)	29(0)	31(0)	40(0)
Yule's Y	44(18)	43(18)	44(19)	25(14)	29(0)	31(0)	40(0)
Kappa	*21(28)	32(25)	*31(30)	20(15)	29(0)	31(0)	40(0)
J-Measure	25(30)	31(26)	*31(30)	26(17)	29(0)	31(0)	40(0)
Gini Index	25(30)	32(25)	*31(30)	26(19)	29(0)	31(0)	40(0)
Support	67(11)	51(12)	55(10)	41(18)	29(0)	31(0)	40(0)
Confidence	*21(28)	26(29)	*31(30)	20(15)	29(0)	31(0)	40(0)
Laplace	67(11)	51(12)	55(10)	41(18)	29(0)	31(0)	40(0)
Conviction	44(18)	42(18)	44(19)	25(14)	29(0)	31(0)	40(0)
Interest	*21(28)	*26(29)	*31(30)	20(15)	29(0)	31(0)	40(0)
Piatetsky-Shapiro's	52(20)	46(16)	36(13)	46(22)	29(0)	31(0)	40(0)
Certainty Factor	44(18)	42(18)	44(19)	25(14)	29(0)	31(0)	40(0)
Added Value	*21(28)	*26(29)	*31(30)	20(15)	29(0)	31(0)	40(0)
Collective Strength	*21(28)	31(26)	*31(30)	20(15)	29(0)	31(0)	40(0)
Jaccard	*21(28)	53(9)	*31(30)	19(15)	29(0)	31(0)	40(0)
Klosgen	*21(28)	28(28)	*31(30)	19(14)	29(0)	31(0)	40(0)
Information Gain	25(30)	32(25)	*31(30)	26(18)	29(0)	31(0)	40(0)
Coverage	59(21)	47(19)	34(15)	40(19)	29(0)	31(0)	40(0)
Accuracy	25(30)	38(27)	32(30)	24(21)	29(0)	31(0)	40(0)
Leverage	25(30)	28(28)	*31(30)	24(17)	29(0)	31(0)	40(0)

*marks the measure(s) with the lowest mean and the underline marks measures that have a mean within 10%.

DOI: 10.1002/smr

Table XV. Detailed means and standard deviations (in parentheses) of percentages of code inspected to find all bugs in the Java programs (Part II).

	Programs						
Association Measures	Nano_v1 %(%)	Nano_v2 %(%)	Nano_v3 %(%)	Nano_v5 %(%)	XML-sec_v1 %(%)	XML-sec_v2 %(%)	XML-sec_v3 %(%)
Relative Risk	67(11)	51(12)	55(10)	40(16)	29(0)	31(0)	40(0)
Int. Weighting Dependency	*21(28)	27(28)	*31(30)	20(15)	29(0)	31(0)	40(0)
Goodman and Kruskal	29(35)	38(27)	40(30)	42(30)	29(0)	31(0)	40(0)
Normalized Mutual Info.	*21(28)	43(18)	*31(30)	21(14)	29(0)	31(0)	40(0)
One-Way Support	*21(28)	*26(29)	*31(30)	21(18)	29(0)	31(0)	40(0)
Two-Way Support	*21(28)	30(27)	*31(30)	20(13)	29(0)	31(0)	40(0)
Two-Way Support Variation	25(30)	32(25)	31(30)	26(18)	29(0)	31(0)	40(0)
Loevinger	63(20)	50(21)	35(15)	45(21)	29(0)	31(0)	40(0)
Sebag	*21(28)	*26(29)	*31(30)	20(15)	29(0)	31(0)	40(0)
Least Contradiction	25(30)	38(27)	32(30)	24(21)	29(0)	31(0)	40(0)
Odd Multiplier	*21(28)	*26(29)	*31(30)	20(15)	29(0)	31(0)	40(0)
Example and Counter.	*21(28)	*26(29)	*31(30)	20(15)	29(0)	31(0)	40(0)
Zhang	*21(28)	*26(29)	*31(30)	20(15)	29(0)	31(0)	40(0)
Sorensen-Dice	*21(28)	53(9)	42(35)	19(15)	29(0)	31(0)	40(0)
Anderberg	*21(28)	53(9)	42(35)	19(15)	29(0)	31(0)	40(0)
Simple-Matching	25(30)	38(27)	43(35)	24(21)	29(0)	31(0)	40(0)
Rogers and Tanimoto	25(30)	38(27)	43(35)	24(21)	29(0)	31(0)	40(0)
Ochiai II	25(30)	32(25)	42(35)	24(18)	29(0)	31(0)	40(0)
Tarantula	*21(28)	*26(29)	*31(30)	20(15)	29(0)	31(0)	40(0)
Ochiai	*21(28)	53(10)	*31(30)	*18(15)	29(0)	31(0)	40(0)

*marks the measure(s) with the lowest mean and the underline marks measures that have a mean within 10%

DOI: 10.1002/smr

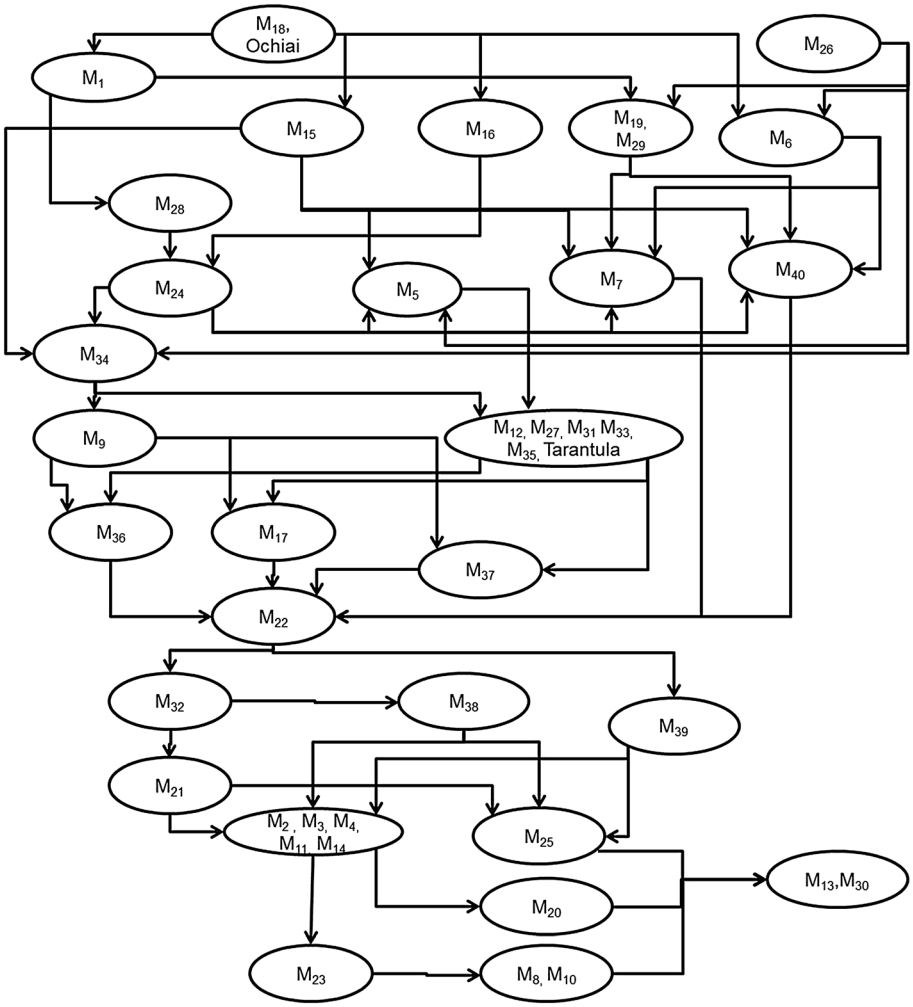

Figure 2. Accuracy partial order.

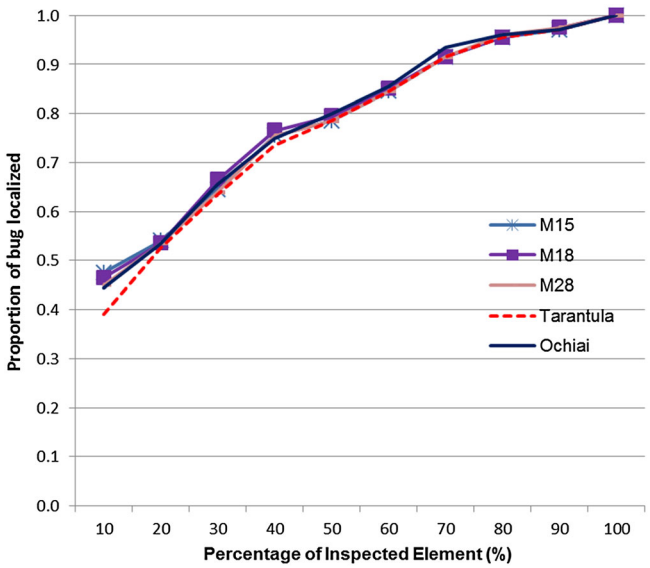

Figure 3. Comparing(M_{15}) and (M_{18}) with Ochiai and Tarantula for all datasets.

DOI: 10.1002/smr

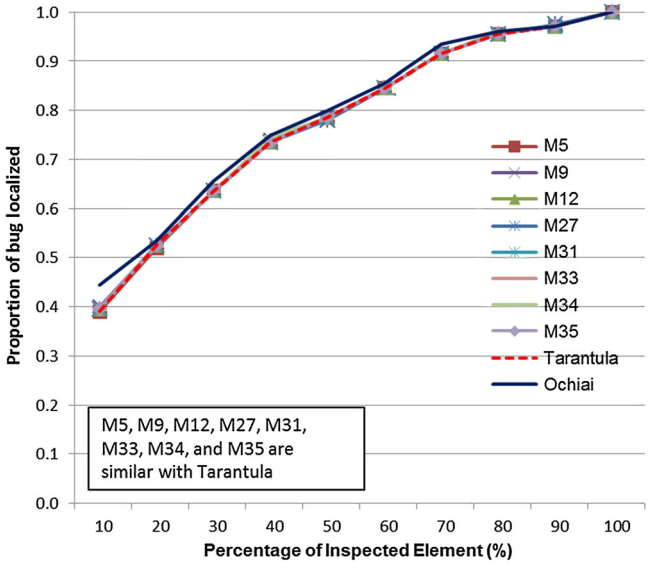

Figure 4. Comparing M_5,M_9, M_{12}, M_{27}, M_{31}, M_{33}, M_{15}, and $-M_{35}$ with Ochiai and Tarantula for all datasets.

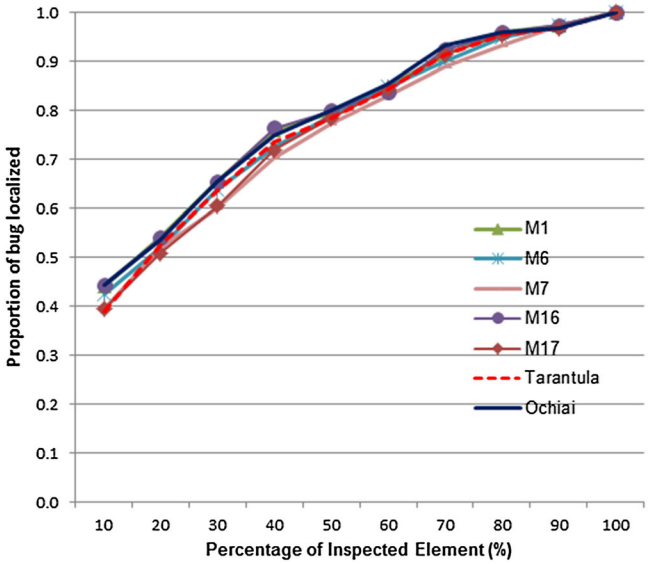

Figure 5. Comparing M_1, M_6, M_7, M_{16}, and M_{17} with Ochiai and Tarantula for all datasets.

on the first accuracy criterion (i.e., percentage of code inspected), we generate two partial orders for the C and Java programs separately to evaluate which measures perform well in each of the programming languages. We highlight measures that are at the top of the partial orders (i.e., no other measures perform statistically significantly better than them). For C programs, Ochiai and Klosgen (M_{18}) are measures that are at the top of the partial order. For Java programs, a number of measures are at the top of the partial order namely Klosgen (M_{18}), Ochiai, Confidence (M_9), Interest (M_{12}), Added Value (M_{15}), Sebag (M_{31}), Example and Counterexample Rate (M_{34}), Zhang (M_{35}), Tarantula, Interestingness Weighting Dependency (M_{24}), and Normalized Mutual Information (M_{26}).

We also evaluate the measures based on the second accuracy criterion (i.e., proportion of bugs found). For C programs, Tarantula and Ochiai could localize 43 and 52% of total bugs within 10% of inspected program elements. Ochiai performs better than Tarantula. Figure 9 shows measures that

DOI: 10.1002/smr

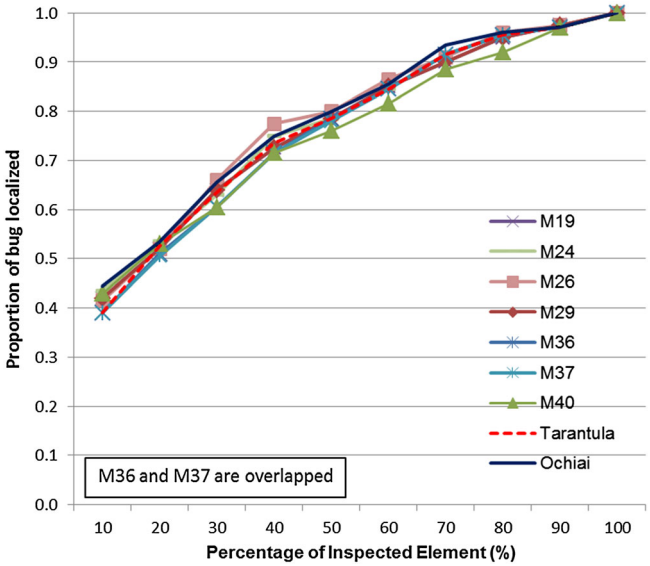

Figure 6. Comparing M_{19}, M_{24}, M_{26}, M_{29}, M_{36}, M_{37}, and M_{40} with Ochiai and Tarantula for all datasets.

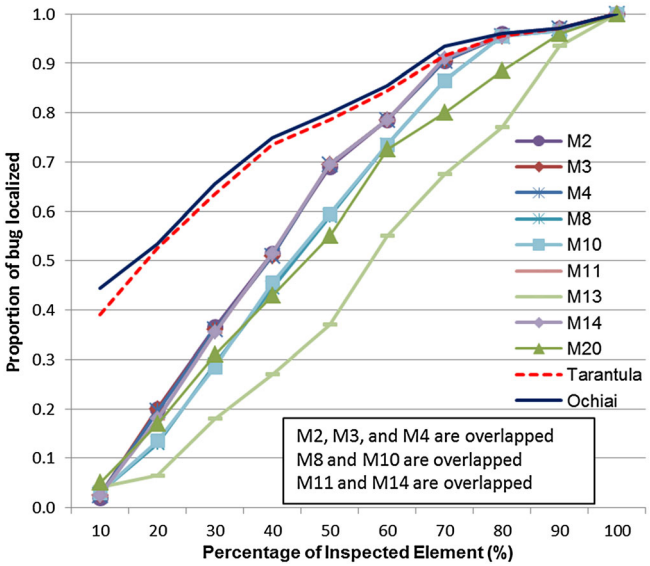

Figure 7. Comparing M_4, M_8, M_{10}, M_{11}, M_{13}, M_{14}, and M_{20} with Ochiai and Tarantula for all datasets.

perform better than Ochiai and Tarantula or similar with Ochiai when only 10% of program elements are inspected. Added Value (M_{15}) and Klosgen (M_{18}) could localize 54 and 53% of the bugs, respectively. Two-Way Support (M_{28}) could localize slightly lower than Ochiai, 51%.

Figures 10 and 11 show measures that perform better than Tarantula but not as good as Ochiai. ϕ-Coefficient (M_1), Collective Strength (M_{16}), J-Measure (M_6), Information Gain (M_{19}), Interestingness Weighting Dependency (M_{24}), Normalized Mutual Information (M_{26}), Two-Way Support Variation (M_{29}), Ochiai II (M_{40}), Gini Index (M_7), Jaccard (M_{17}), Sorensen-Dice (M_{36}), Anderberg (M_{37}) could localize 51, 51, 49, 49, 49, 49, 49, 49, 45, 45, 45, and 45% of the bugs, respectively. Measures that perform similar to Tarantula are shown in Figure 12. They localize 43–44% of the bugs when 10% of program elements are inspected. The association measures included in Figures 13 and 14 perform worse than Tarantula and Ochiai.

DOI: 10.1002/smr

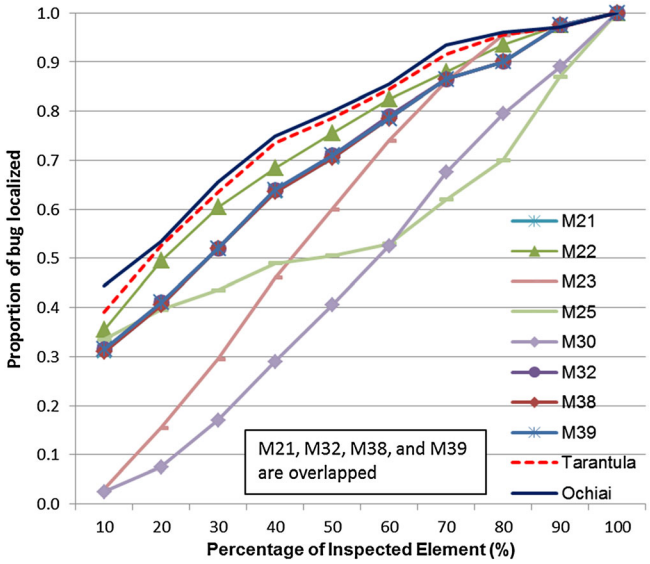

Figure 8. Comparing $M_{21} - M_{23}$, M_{25}, M_{30}, M_{32}, M_{38}, and M_{39} with Ochiai and Tarantula for all datasets.

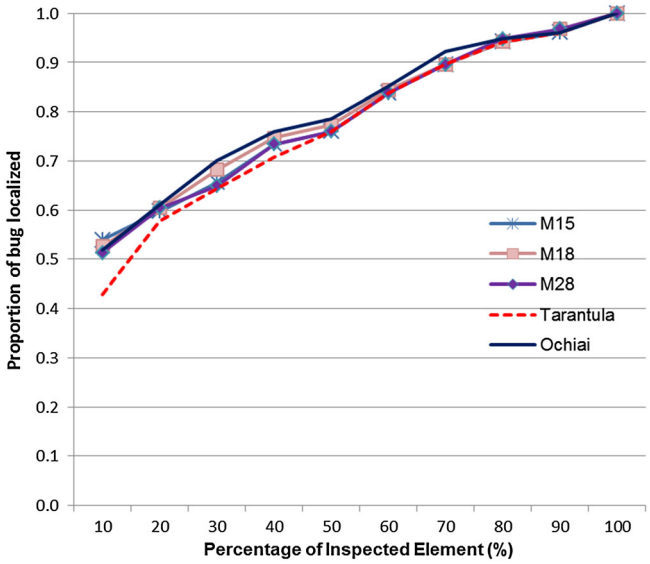

Figure 9. Comparing M_{15}, M_{18}, and M_{28} with Ochiai and Tarantula for C programs.

For localizing bugs in Java programs, Tarantula performs better than Ochiai. Notice that when 10% of program elements are inspected, Tarantula and Ochiai could localize 26 and 20% of the bugs. Figure 15 shows measures that have similar performance as Tarantula. They could localize 26% of the bugs within 10% of inspected program elements. The measures are Confidence (M_9), Interest (M_{12}), Added Value (M_{15}), Sebag (M_{31}), Odd Multiplier (M_{33}), and Example and Counterexample Rate (M_{34}).

Figures 16 –18 show measures that have performance between Tarantula and Ochiai when 10% of program elements are inspected. Figures 19 and 20 show measures that perform worse than Ochiai and Tarantula when localizing bugs in Java programs. Based on the results, we notice that there are a number of measures that could perform as good as Tarantula and Ochiai when using different programming language.

DOI: 10.1002/smr

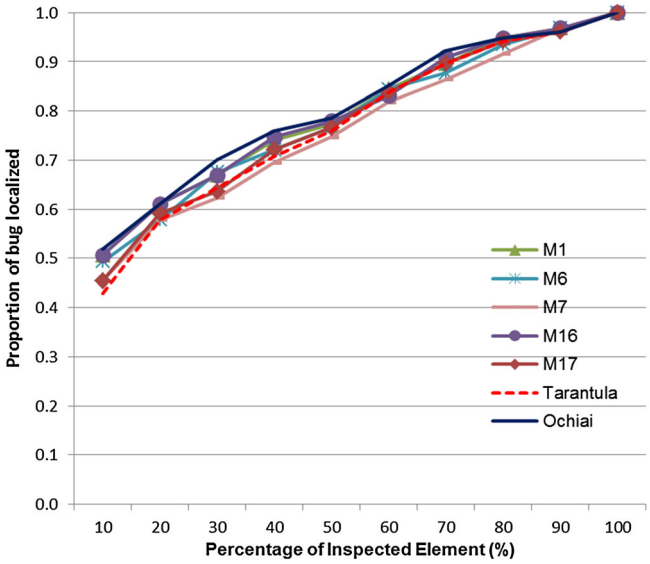

Figure 10. Comparing M_1, M_6, M_7, M_{16}, and M_{17} with Ochiai and Tarantula for C programs.

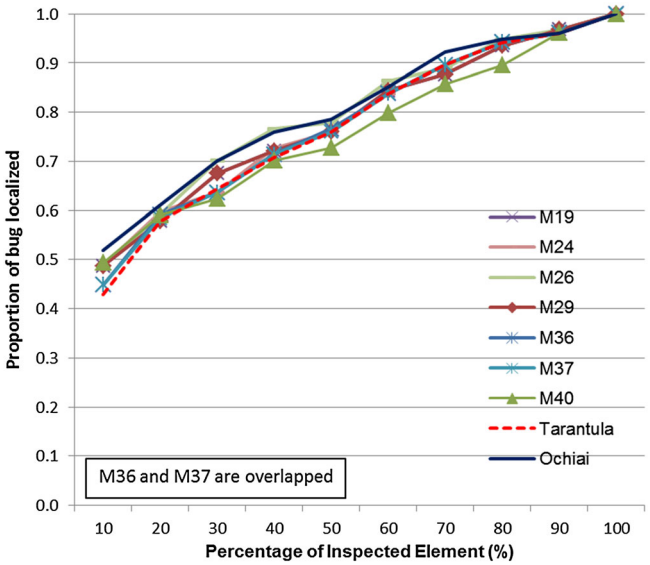

Figure 11. Comparing M_{19}, M_{24}, M_{26}, M_{29}, M_{36}, M_{37}, and M_{40} with Ochiai and Tarantula for C programs.

The differences in accuracies for localizing bug in C and Java programs possibly imply that there could be specific characteristic of spectra produced by different program languages that could advantage or disadvantage spectrum-based fault localization. For example, in Java, because of object oriented model, more program elements (e.g., class constructors) may be always executed together with actual faulty ones, and thus obscure the suspiciousness scores for the actual faults.

5.5. *Effectiveness for various kinds of bugs*

We divide the bugs into several groups based on the bug-fix categorization by Kim *et al*. [29]. The following paragraphs describe our bug categories and present the effectiveness of the various association measures, Tarantula, and Ochiai on different bug categories.

DOI: 10.1002/smr

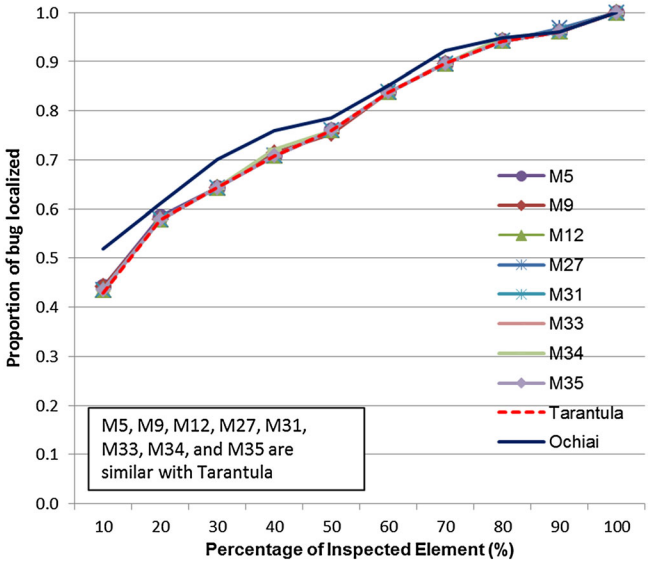

Figure 12. Comparing M_5, M_9, M_{12}, M_{27}, M_{31}, M_{33}, and M_{35} with Ochiai and Tarantula for C programs.

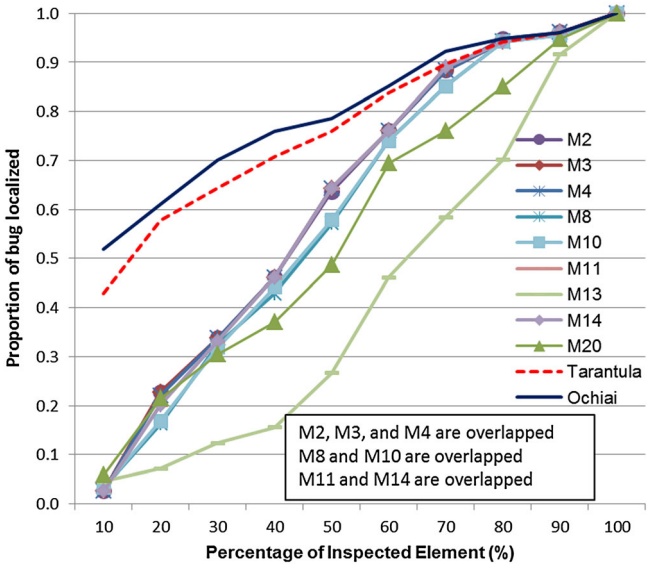

Figure 13. Comparing $M_2 - M_4$, M_8, M_{10}, M_{11}, M_{13}, M_{14}, and M_{20} with Ochiai and Tarantula for C programs.

5.5.1. Bug Categories. Kim *et al*. [29] describe a number of bug categories based on the way bugs are fixed. In this paper, we refer to their categories to analyze the bugs in our subject programs. We categorize bugs in our subject programs into eight categories. Table XVI shows the categories and number of buggy versions for each type of bugs. We add two categories that are not included by Kim *et al*. [29]: change of return expression (CH-RET) and others (OTH).

We categorize a bug into addition or removal of conditional statement category (CH-CS) when a bug could be fixed by adding a conditional check statement (e.g. if statement) or removing an inappropriate check statement. This type of bugs occurs when there is a missing precondition or post condition check of some variables, or an extraneous conditional check. Bugs that could be fixed by adding or removing statement, which is not a conditional check statement are categorized

DOI: 10.1002/smr

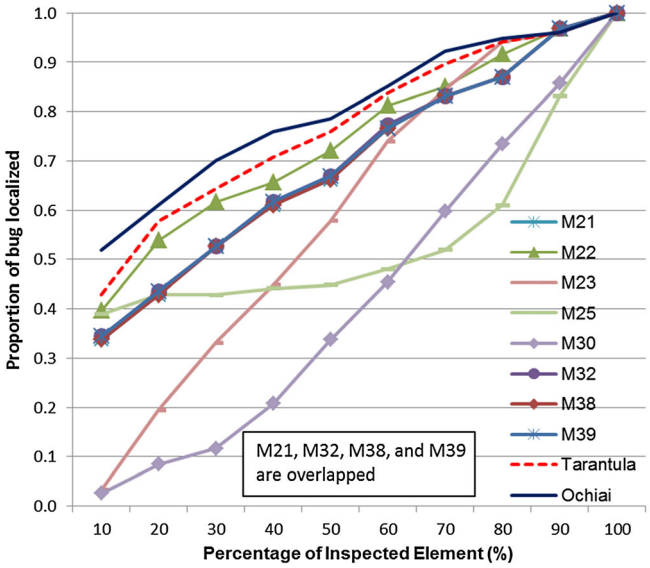

Figure 14. Comparing $M_{21} - M_{23}$, M_{25}, M_{30}, M_{35}, M_{38}, and M_{39} with Ochiai and Tarantula for C programs.

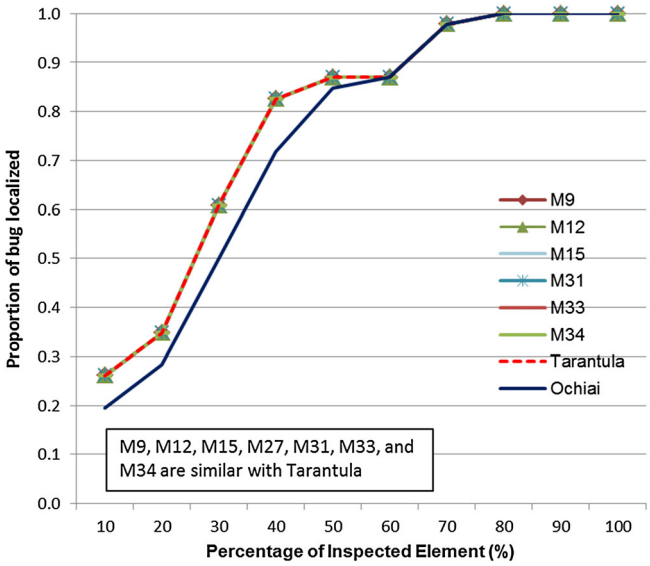

Figure 15. Comparing M_9, M_{12}, M_{15}, M_{27}, M_{31}, M_{33}, and M_{34} with Ochiai and Tarantula for Java programs.

into addition/removal of non-conditional statement (CH-NCS) category. An example of this type of bug is missing or extraneous assignment statements.

Change in method calls (CH-MC) category includes bugs that can be fixed by adding or removing a method call in a program, or by changing the parameter values of a method call. Bugs that could be fixed by changing the right hand side of an assignment expression are categorized into change of assignment expression category. When a bug could be fixed by modifying a conditional expression within an if statement, we categorize the bug into change of if condition expression category. Similarly, when a bug could be fixed by modifying a conditional expression in a looping statement, then we put this bug into change of loop predicate (LP-CC) category. When a bug could be fixed by changing the value or the expression in a return statement, then we categorize the bug into (CH-RET) category.

DOI: 10.1002/smr

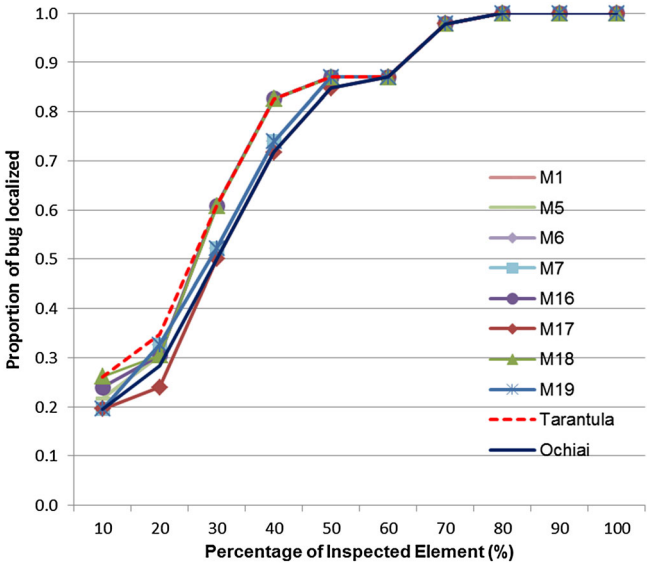

Figure 16. Comparing M_1, M_5, M_6, M_7, M_{16}, M_{17}, M_{18}, and M_{19} with Ochiai and Tarantula for Java programs.

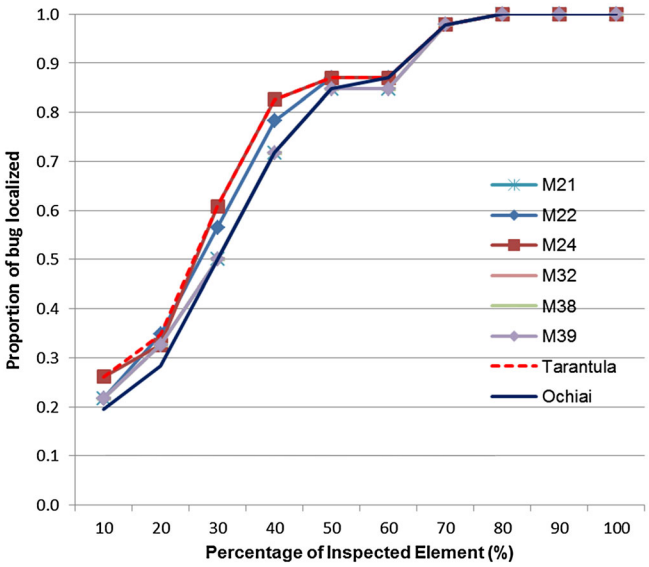

Figure 17. Comparing M_{21}, M_{22}, M_{24}, M_{32}, M_{38}, and M_{39} with Ochiai and Tarantula for Java programs.

We create a category named (OTH) to include other bugs that are not covered by the previous categories, for example, a bug is fixed by changing an if statement to a for loop statement and so on. We ignore bugs in category (OTH).

5.5.2. Effectiveness. We first evaluate the effectiveness of the various measures on each category by the first accuracy criterion (i.e., percentage of code inspected). Based on this criterion, we create several partial orders based on statistically significantly better relationships among the measures. We highlight measures that are at the top of the partial orders as shown in Table XVII.

On the basis of these partial orders, Tarantula is at the top of the partial orders of three bug categories. Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{26}), Two-Way Support Variation

DOI: 10.1002/smr

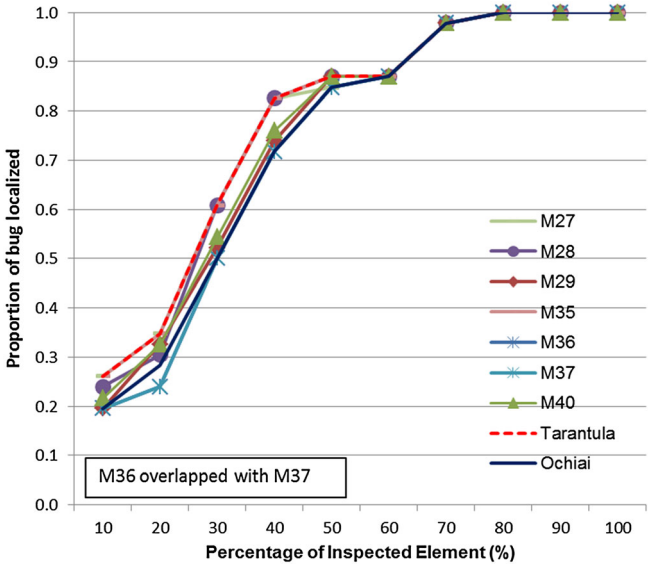

Figure 18. Comparing $M_{27} - M_{29}$, M_{35}, $-M_{37}$, and M_{40} with Ochiai and Tarantula for Java programs.

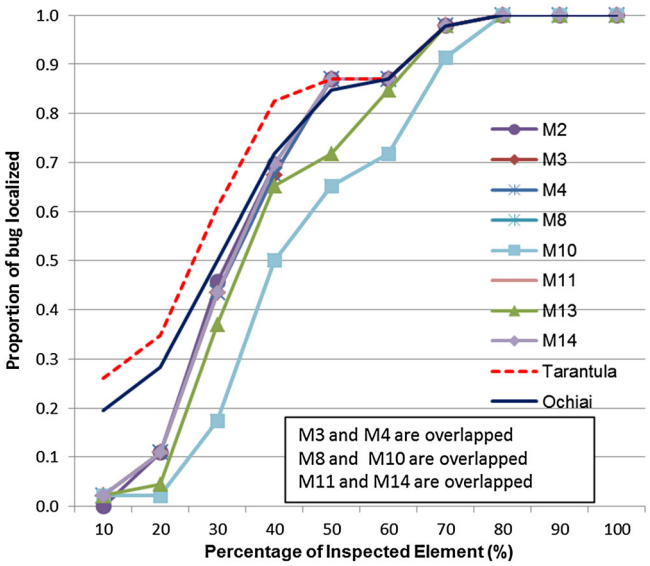

Figure 19. Comparing $M_2 - M_4$, M_8, M_{10}, M_{11}, and M_{14} with Ochiai and Tarantula for Java programs.

(M_{29}) are at the top of the partial orders of six bug categories. Klosgen (M_{18}) is at the top of the partial orders of all bug categories.

Next, we evaluate the effectiveness of the measures on each bug category by the second accuracy criterion (i.e., proportion of bugs localized). We compute the percentage of bugs localized when up to 10% of program elements are inspected, as shown in Table XVIII and XIX.

For each category shown in Tables XIII and XIX, different measures have different effectiveness to localize bugs. The star (*) marks the measures that could localize most bugs in each category. φ-Coefficient (M_1), J-Measure (M_6), Added Value (M_{15}), Collective Strength (M_{16}), Information Gain (M_{19}), Normalized Mutual Information (M_{26}), Two-Way Support (M_{28}), Two-Way Support Variation (M_{29}), and Ochiai could localize the most number of bugs in (CH-CS) category. They could localize 43% of the bugs in this category. φ-Coefficient (M_1), Added Value (M_{15}), Collective Strength (M_{16}), Klosgen (M_{18}), Interestingness Weighting Dependency (M_{24}), Two-Way Support (M_{28}),

DOI: 10.1002/smr

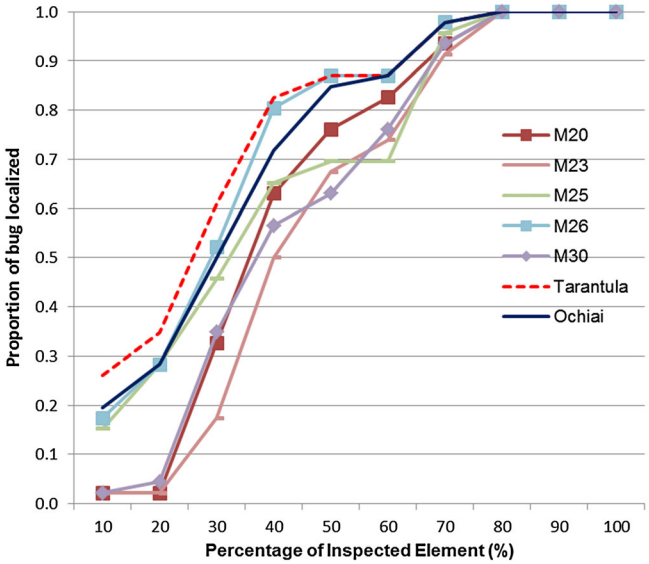

Figure 20. Comparing M_{20}, M_{23}, M_{25}, M_{26}, and M_{30} with Ochiai and Tarantula for Java programs.

Table XVI. Bug categories

Bug category	Total version
Addition/removal of conditional statement (CH-CS)	28
Addition/removal of non-conditional statement (CH-NCS)	10
Change in method calls (CH-MC)	26
Change of assignment expression (AS-CE)	60
Change of if condition expression (IF-CC)	44
Change of loop predicate (LP-CC)	11
Change of return expression (CH-RET)	18
Others (OTH)	3

Ochiai II (M_{40}), and Ochiai could better localize bugs in (CH-NCS) category than other measures. They could localize 70% of the bugs in this category.

For localizing (CH-MC) bugs, at most 35% of the bugs in this category could be localized by the measures. There are number of measures that could localize 35% of the bugs in this category, they are Confidence (M_9), Interest (M_{12}), Added Value (M_{15}), Accuracy (M_{21}), Interestingness Weighting Dependency (M_{24}), One-Way Support (M_{27}), Sebag (M_{31}), Least Contradiction (M_{32}), Odd Multiplier (M_{33}), Example and Counterexample Rate (M_{34}), Zhang (M_{35}), Simple-Matching (M_{38}), Rogers and Tanimoto (M_{39}), and Tarantula.

Added Value (M_{15}), Collective Strength (M_{16}), Klosgen (M_{18}) localize 48% of the bugs in change of assignment expression category. J-Measure (M_6), Added Value (M_{15}), Klosgen (M_{18}), and Ochiai localize 48% of the bugs in change of if condition expression (IF-CC) category.

Added Value (M_{15}), Collective Strength (M_{16}), Interestingness Weighting Dependency (M_{24}), Two-Way Support (M_{27}), and Ochiai II (M_{40}) could localize 64% of the bugs in (LP-CC) category. For bugs in (CH-RET) category, Klosgen (M_{18}) could localize these bugs better than others, that is, 61% of the bugs could be localized.

By only inspecting up to 10% of the program elements, bugs in (CH-MC) category are not easy to be localized. The best measures could only localize 35% of these bugs. On the other hand, bugs in (CH-NCS), (LP-CC), and (CH-RET) categories could be better localized by the measures (i.e., up to 70, 64, and 61% respectively). We notice that Added Value (M_{15}) is the measure that could localize the most number of bugs in six categories. Klosgen (M_{15}) and Two-Way Support (M_{28}) could

DOI: 10.1002/smr

Table XVII. Measures that are at the top of the partial orders for each bug categories.

Bug category	Top Measures
Addition/removal of conditional statement (CH-CS)	Klosgen (M_{18}), Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{26}), Two-Way Support Variation (M_{29}),Tarantula, ϕ-Coefficient (M_1), Kappa (M_5), J-Measure (M_6), Confidence (M_9), Added Value (M_{15}), Collective Strength (M_{16}), Interestingness Weighting Dependency (M_{24}), One-Way Support (M_{27}), Two-Way Support (M_{28}), Sebag (M_5), Odd Multiplier (M_{33}), Example and Counterexample Rate (M_{34}), Zhang (M_{35}), Sorensen-Dice (M_{36}), Anderberg (M_{37}), interest(M_{12}), Leverage(M_{22}), and Jaccard(M_{17})
Addition/removal of non-conditional statement (CH-NCS)	Klosgen (M_{18}), Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{26}), Two-Way Support Variation (M_{29}), φ-Coefficient (M_1), Kappa (M_5), Gini Index (M_7), J-Measure (M_6), Added value (M_{15}), Collective Strength (M_{16}), Jaccard (M_{17}), Interestingness Weighting Dependency (M_{24}), Two-Way Support (M_{28}), Anderberg (M_{37}), and Ochiai II (M_{40})
Change in method calls (CH-MC)	Klosgen (M_{18}), Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{26}), Two-Way Support Variation (M_{29}), Tarantula, ϕ-Coefficient (M_1), Confidence (M_9), Interest (M_{12}), Added Value (M_{15}), Collective Strength (M_{16}), Interestingness Weighting Dependency (M_{24}), One-Way Support (M_{27}), Two-Way Support (M_{28}), Sebag (M_{31}), Odd Multiplier (M_{33}), Example and Counterexample Rate (M_{34}), and Zhang (M_{35})
Change of assignment expression (AS-CE)	Klosgen (M_{18}), Ochiai, and Normalized Mutual Information (M_{26})
Change of if condition expression (IF-CC)	Klosgen (M_{18}), Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{26}), Two-Way Support Variation (M_{29}), and J-Measure (M_6)
Change of loop predicate (LP-CC)	Klosgen (M_{18}), Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{26}), Two-Way Support Variation (M_{29}), Tarantula, ϕ-Coefficient (M_1), Kappa (M_5), Collective Strength (M_{16}), J-Measure (M_6), Gini Index (M_7), Confidence (M_9), Added Value (M_{15}), Jaccard (M_{17}), Interestingness Weighting Dependency (M_{24}), Goodman Kruskal (M_{25}), One-Way Support (M_{27}), Two-Way Support (M_{28}), Sebag (M_{31}), Example and Counterexample Rate (M_{34}), Zhang (M_{35}), Sorensen-Dice (M_{36}), Anderberg (M_{37}), Interest(M_{12}), Leverage(M_{22}), Odd Multiplier (M_{33}), Support(M_8), and Ochiai II (M_{40})
Change of return expression (CH-RET)	Klosgen (M_{18}), Information Gain (M_{19}), Two-Way Support Variation (M_{29}), J-Measure (M_6), Gini Index (M_7), Added Value (M_{15}), Interestingness Weighting Dependency (M_{24}), Two-Way Support (M_{28}), Accuracy(M_{21}), Least Contradiction(M_{32}), Simple-Matching (M_{38}), and Rogers and Tanimoto (M_{39})

The underline marks measures that are at the top of the partial orders of more than five bug categories.

DOI: 10.1002/smr

Table XVIII. Effectiveness of bug localization for (M_1) to (M_{35}) for each bug category when up to 10% program elements are inspected (Part I).

Measures	CH-CS (%)	CH-NCS (%)	CH-MC (%)	AS-CE (%)	IF-CC (%)	LP-CC (%)	CH-RET (%)
ϕ-Coefficient(M_1)	*43	*70	31	47	41	55	50
Odds Ratio(M_2)	0	0	0	3	5	0	0
Yule's Q(M_3)	0	0	4	3	5	0	0
Yule's Y(M_4)	0	0	4	3	5	0	0
Kappa(M_5)	39	60	27	43	30	55	50
J-Measure(M_6)	*43	60	23	45	*48	36	50
Gini Index(M_7)	39	60	23	45	36	36	50
Support(M_8)	0	0	4	5	5	0	0
Confidence(M_9)	39	60	*35	40	32	55	56
Laplace(M_{10})	0	0	4	5	5	0	0
Conviction(M_{11})	0	0	4	3	5	0	0
Interest(M_{12})	39	60	*35	40	30	55	56
Pietatsky-Shapiro(M_{13})	4	0	4	8	2	0	0
Certainty Factor(M_{14})	0	0	4	3	5	0	0
Added Value(M_{15})	*43	*70	*35	*48	*48	*64	56
Collective Strength(M_{16})	*43	*70	31	*48	39	*64	50
Jaccard(M_{17})	39	60	27	43	32	55	50
Klosgen(M_{18})	39	*70	31	*48	*48	55	*61
Information Gain(M_{19})	*43	60	23	45	45	36	50
Coverage(M_{20})	4	0	8	5	7	9	0
Accuracy(M_{21})	36	40	*35	37	18	18	39
Leverage(M_{22})	36	40	31	35	27	55	56
Relative Risk(M_{23})	0	0	4	5	5	0	0
Int. Weighting Dependency(M_{24})	39	*70	*35	43	39	*64	56
Goodman and Kruskal(M_{25})	29	30	23	42	25	45	50
Normalized Mutual Info.(M_{26})	*43	60	27	43	43	36	50
One-Way Support(M_{27})	39	60	*35	40	30	55	56
Two-Way Support(M_{28})	*43	*70	31	47	43	*64	50
Two-Way Support Variation(M_{29})	*43	60	23	45	45	36	50
Loevinger(M_{30})30	0	0	4	5	2	0	0
Sebag(M_{31})	39	60	*35	40	30	55	56
Least Contradiction(M_{32})	36	40	*35	37	20	18	39
Odd Multiplier(M_{33})	39	60	*35	40	30	55	56
Example and Counter.(M_{34})	39	60	*35	40	30	55	56
Zhang(M_{35})	39	60	*35	40	30	55	56

*marks the measures that localize bugs the most for each category

DOI: 10.1002/smr

Table XIX. Effectiveness of (M_{36}) to (M_{40}), Tarantula, and Ochiai for each bug category when up to 10 program elements are inspected (Part II).

Measures	CH-CS (%)	CH-NCS (%)	CH-MC (%)	AS-CE (%)	IF-CC (%)	LP-CC (%)	CH-RET (%)
Sorensen-Dice (M_{36})	39	60	27	43	30	55	50
Anderberg (M_{37})	39	60	27	43	30	55	50
Simple-Matching (M_{38})	36	40	*35	37	18	18	39
Rogers and Tanimoto (M_{39})	36	40	*35	37	20	18	39
Ochiai II (M_{40})	39	*70	31	45	39	*64	50
Tarantula	39	60	*35	40	27	55	56
Ochiai	*43	*70	27	45	*48	55	50

*marks the measures that localize bugs the most for each category

localize the most number of bugs in four bug categories. Ochiai and Tarantula could localize the most number of bugs in three and one categories respectively.

Figures 21–24 show the effectiveness of the measures in localizing bugs for each category. As shown in Figure 24, Coverage (M_{20}), Pietatsky-Shapiro (M_{13}), Support (M_8), Laplace (M_{10}), Relative

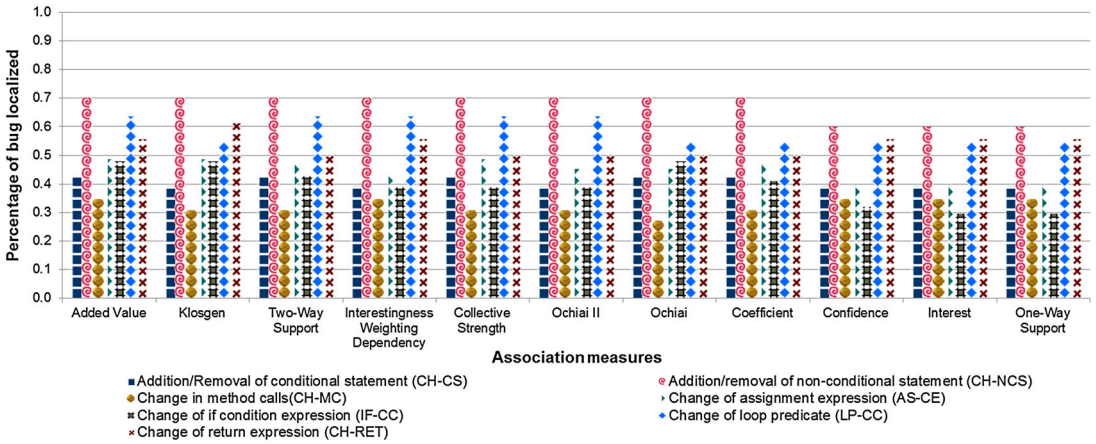

Figure 21. Effectiveness of measures to localize bugs by inspecting up to 10% of code (Part I).

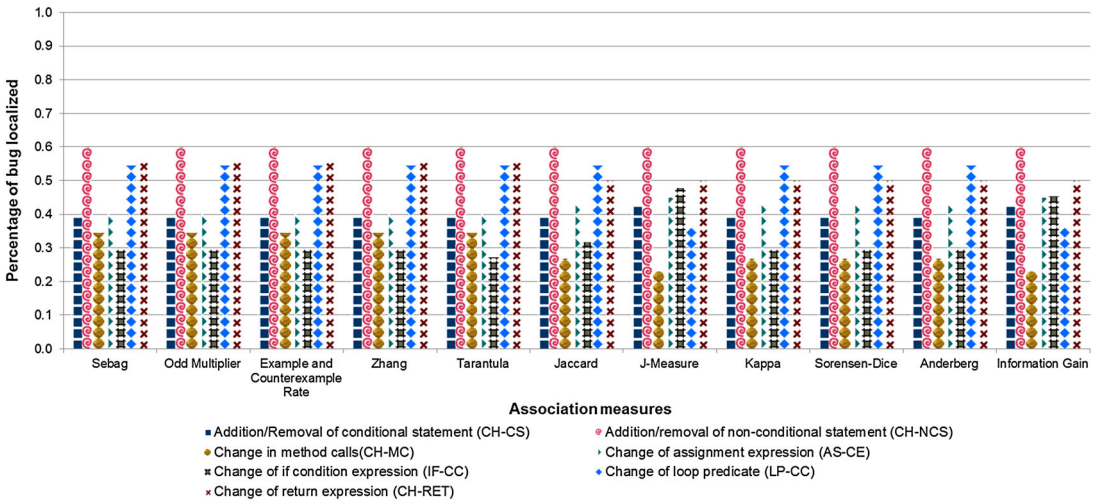

Figure 22. Effectiveness of measures to localize bugs by inspecting up to 10% of code (Part II).

DOI: 10.1002/smr

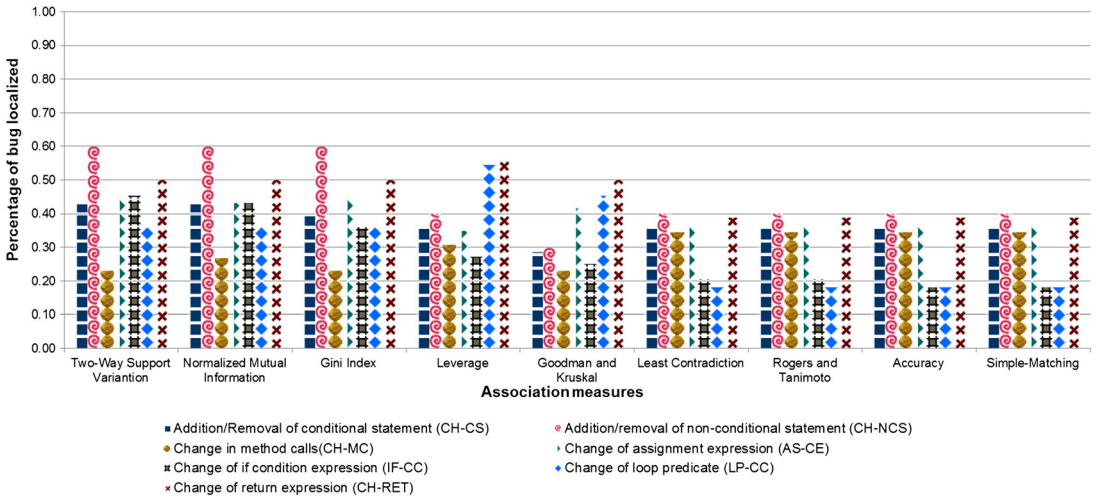

Figure 23. Effectiveness of measures to localize bugs by inspecting up to 10% of code (Part III).

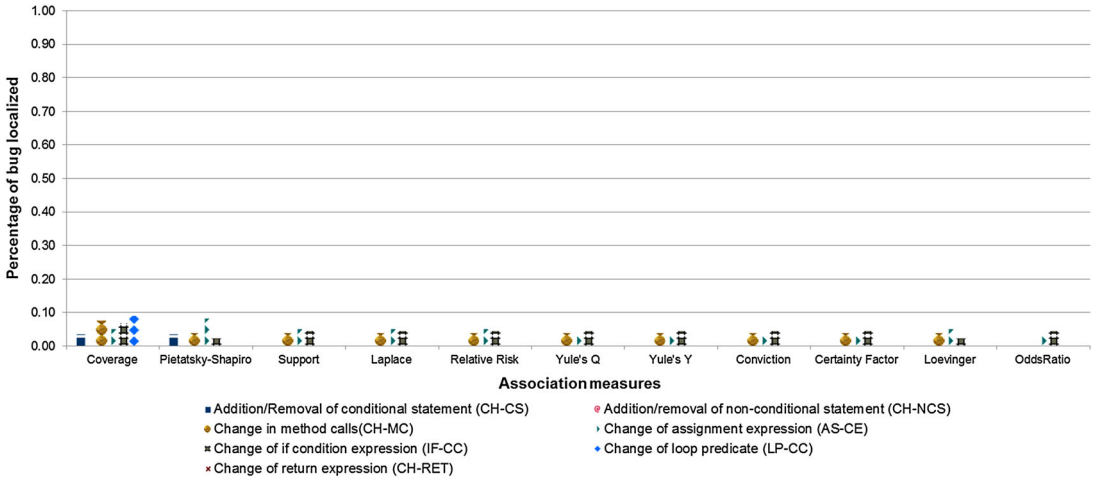

Figure 24. Effectiveness of measures to localize bugs by inspecting up to 10% of code (Part IV).

Risk (M_{23}), Yule's Q (M_3), Yule's Y (M_4), Conviction (M_{11}), Certainty Factor (M_{14}), Loevinger (M_{30}), and Odds Ratio (M_2) could only localize a small number of bugs for each bug category. Figures 21, 22, and 23 show measures that could localize more bugs for most of the categories. A number of measures could localize at least 50% of the bugs in (CH-NCS), (LP-CC), and (CH-RET) categories.

5.6. *Effectiveness on multiple-bug versions*

We evaluate the effectiveness of association measures, Tarantula, and Ochiai to localize multiple bugs in programs. We refer these programs as multiple-bug versions. A multiple-bug version contains a number of bugs where each bug only involves one line in the program and different bugs affect different lines [15, 17].

We generate 173 multiple-bug versions of C programs as shown in Table XX. As print_token and schedule2 datasets only have four and seven bugs that involve one line, we randomly insert two bugs for every version, whereas for other datasets, we randomly insert five bugs for every multiple-bug version. Also, we ensure that each bug has been inserted at least in one of the versions.

DOI: 10.1002/smr

Table XX. Multiple-bug datasets.

Dataset	Num. Of bug in a version	Num. Of single bug in dataset	Language	Num. Of buggy version
print_token	2	4	C	10
schedule2	2	5	C	10
print_token2	5	9	C	10
replace	5	25	C	32
schedule	5	7	C	9
tcas	5	30	C	41
tot_info	5	19	C	23
space	5	19	C	38

We generate multiple-bug versions for each dataset as many as the number of single-bug versions in the dataset. For example, there are 38 single-bug versions for Space, so we randomly generate 38 multiple-bug versions for Space, each of which contains five bugs. For each print_tokens and schedule2 dataset, we generate 10 multiple-bug versions. Thus, we have 20 multiple-bug versions that contain two bugs (minimum number of multiple bugs) and 153 versions that contain five bugs.

We evaluate the effectiveness of 40 association measures, Tarantula, and Ochiai to localize multiple bugs in programs. Tables XXI–XXIII show the overall mean and standard deviation of percentage of code inspected of the measures to localize bugs in all multiple-bug versions, versions containing five bugs, and versions containing two bugs, respectively.

Generally, the effectiveness of all measures to localize multiple bugs is not as good as localizing single bugs. The overall ranges of mean of percentage of code inspected of the measures to localize all multiple-bug versions, five bugs, and two bugs are between 74 and 91%, 76 and 92%, and 43 and 87%, respectively. Measures that could localize all multiple-bug versions, five bugs, and two bugs in the programs with the smallest percentage of code inspected are Added value (M_{15}), Relative Risk (M_{23}), and Two-Way Support (M_{28}), respectively.

Table XXI. Overall mean and standard deviation (in parentheses) of accuracy values (smaller the better) of all multiple-bug versions.

Association Measures	Mean (%)	SD (%)	Association Measures	Mean (%)	SD (%)
Added Value (M_{15})	73.88	23.36	Klosgen (M_{18})	74.92	22.39
One-Way Support (M_{27})	73.98	23.50	Ochiai II (M_{40})	75.07	22.61
Collective Strength (M_{16})	73.99	23.36	Yule's Q (M_3)	75.34	20.43
Ochiai	74.03	22.95	Yule's Y (M_4)	75.36	20.38
ϕ-Coefficient (M_1)	74.1	22.80	Leverage (M_{22})	75.76	21.42
Tarantula	74.14	23.28	Odds Ratio (M_2)	76.58	19.99
Odd Multiplier (M_{33})	74.16	23.30	Laplace (M_{10})	76.61	19.87
Zhang (M_{35})	74.17	23.28	Gini Index (M_7)	76.97	21.48
Interest (M_{12})	74.17	23.29	Rogers (M_{39})	77.37	20.33
Sebag (M_{31})	74.17	23.28	Accuracy (M_{21})	77.40	20.34
Confidence (M_9)	74.20	23.29	Simple-Matching (M_{38})	77.40	20.34
Interestingness Weighting Dependency (M_{24})	74.20	23.39	Least Contradiction (M_{32})	77.41	20.45
Kappa (M_5)	74.34	22.99	Information Gain (M_{19})	77.58	20.39
Two-Way Support (M_{28})	74.36	22.96	Two-Way Support Variation (M_{29})	77.58	20.39
Jaccard (M_{17})	74.40	21.71	J-Measure (M_6)	77.60	20.33
Relative Risk (M_{23})	74.43	22.41	Normalized Mutual Information (M_{26})	77.79	20.52
Certainty Factor (M_{14})	74.46	21.09	Support (M_8)	77.94	19.34
Conviction (M_{11})	74.46	21.11	Coverage (M_{20})	80.72	17.40
Example and Counter (M_{34})	74.78	23.28	Loevinger (M_{30})	86.24	15.32
Sorensen-Dice (M_{36})	74.79	21.74	Piatetsky-Shapiro's (M_{13})	87.34	12.69
Anderberg (M_{37})	74.81	21.68	Goodman and Kruskal (M_{25})	91.31	9.25

DOI: 10.1002/smr

Table XXII. Overall mean and standard deviation (in parentheses) of accuracy values (smaller the better) of versions containing five bugs.

Association Measures	Mean (%)	SD (%)	Association Measures	Mean (%)	SD (%)
Relative Risk (M_{23})	75.56	23.17	Confidence (M_9)	78.21	20.59
Certainty Factor (M_{14})	76.03	21.34	Laplace (M_{10})	78.24	19.53
Conviction (M_{11})	76.04	21.36	Kappa (M_5)	78.35	20.20
Ochiai	76.16	21.33	Example and Counter (M_{34})	78.41	20.76
Yule's Q (M_3)	76.71	20.85	Two-Way Support (M_{28})	78.46	20.06
Yule's Y (M_4)	76.75	20.81	Ochiai II (M_{40})	78.52	20.85
Jaccard (M_{17})	77.34	20.00	Leverage (M_{22})	79.36	19.08
Coefficient (M_1)	77.52	20.67	Support (M_8)	79.67	18.81
Added Value (M_{15})	77.74	21.08	J-Measure (M_6)	80.10	18.98
Anderberg (M_{37})	77.78	19.96	Information Gain (M_{19})	80.11	19.04
Sorensen (M_{36})	77.78	19.97	Two-Way Support Variation (M_{29})	80.11	19.04
Collective Strength (M_{16})	77.92	20.33	Normalized Mutual Information (M_{26})	80.22	18.92
One-Way Support (M_{27})	77.97	20.91	Gini Index (M_7)	80.27	19.66
Klosgen (M_{18})	78.10	20.45	Rogers (M_{39})	80.74	18.43
Odds Ratio (M_2)	78.12	20.27	Accuracy (M_{21})	80.78	18.43
Tarantula	78.14	20.60	Simple-Matching (M_{38})	80.78	18.43
Odd Multiplier (M_{33})	78.17	20.61	Least Contradiction (M_{32})	80.78	18.57
Zhang (M_{35})	78.18	20.59	Coverage (M_{20})	82.64	16.34
Interest (M_{12})	78.18	20.59	Loevinger (M_{30})	88.58	13.68
Interestingness Weighting Dependency (M_{24})	78.18	20.73	Piatetsky-Shapiro's (M_{13})	89.52	11.38
Sebag (M_{31})	78.18	20.59	Goodman and Kruskal (M_{25})	91.82	9.61

Table XXIII. Overall mean and standard deviation (in parentheses) of accuracy values (smaller the better) of versions containing two bugs.

Association Measures	Mean (%)	SD (%)	Association Measures (%)	Mean (%)	SD (%)
Two-Way Support (M_{28})	42.95	19.45	Gini Index (M_7)	51.65	17.95
Confidence (M_9)	43.50	19.90	Jaccard (M_{17})	51.90	21.55
Interest (M_{12})	43.50	19.90	Sorensen-Dice (M_{36})	51.90	21.55
One-Way Support (M_{27})	43.50	19.90	Anderberg (M_{37})	52.10	21.24
Sebag (M_{31})	43.50	19.90	Ochiai	57.75	28.46
Odd Multiplier (M_{33})	43.50	19.90	Information Gain (M_{19})	58.25	20.44
Zhang (M_{35})	43.50	19.90	Two-Way Support Variation (M_{29})	58.25	20.44
Tarantula	43.50	19.90	J-Measure (M_6)	58.40	20.54
Kappa (M_5)	43.60	19.92	Normalized Mutual Information (M_{26})	59.25	23.20
Interestingness Weighting Dependency (M_{24})	43.75	20.21	Conviction (M_{11})	62.40	14.45
Collective Strength (M_{16})	43.90	23.58	Certainty Factor (M_{14})	62.40	14.45
Added Value (M_{15})	44.35	18.65	Laplace (M_{10})	64.15	18.33
Example and Counter (M_{34})	47.00	23.27	Support (M_8)	64.70	18.65
φ-Coefficient (M_1)	47.95	21.76	Odds Ratio (M_2)	64.80	12.85
Leverage (M_{22})	48.20	18.38	Yule's Q (M_3)	64.80	12.85
Ochiai II (M_{40})	48.65	17.84	Yule's Y (M_4)	64.80	12.85
Klosgen (M_{18})	50.60	22.10	Relative Risk (M_{23})	65.80	12.75
Accuracy (M_{21})	51.60	15.29	Coverage (M_{20})	66.00	18.64
Least Contradiction (M_{32})	51.60	15.29	Loevinger (M_{30})	68.35	15.69
Simple-Matching (M_{38})	51.60	15.29	Piatetsky-Shapiro's (M_{13})	70.70	9.55
Rogers (M_{39})	51.60	15.29	Goodman and Kruskal (M_{25})	87.40	4.17

DOI: 10.1002/smr

We notice that performance of the measures in localizing five and two bugs in the programs are different. Tarantula performs better than Ochiai when localizing two bugs (i.e., 43.05% for Tarantula and 57.75% for Ochiai), in contrast Ochiai performs slightly better than Tarantula when localizing five bugs in the programs (i.e., 78.14% for Tarantula and 76.14% for Ochiai). There are three measures that perform better than Tarantula and Ochiai in localizing all multiple-bug versions in the programs. Among these measures, Added Value (M_{15}) and Collective Strength (M_{16}) also have good performance in localizing single bugs. The overall mean of percentage of code inspected for Added Value (M_{15}) and Collective Strength (M_{16}) are 25.74 and 25.66% respectively, which are only slightly smaller than the smallest mean of percentage of code inspected to localize all single-bug programs (25.24%). In addition, Added Value (M_{15}) and Collective Strength (M_{16}) are in the top of partial order of five and four bug categories out of seven categories.

We also perform statistical tests for each pair of measures including Tarantula and Ochiai using Wilcoxon signed rank test [69] at 0.05 statistical significance threshold to see if some measures are statistically significantly better than others in localizing multiple-bug versions. Table XXIV shows the measures that are on the top of the partial order (no other measures that statistically significantly perform better than the measure) for localizing both all multiple-bug versions, versions containing five bugs, and versions containing two measures. We notice that Added Value (M_{15}), Collective Strength (M_{16}), Ochiai, Ochiai II (M_{40}), One-Way Support (M_{27}), Relative Risk (M_{23}), Two-Way Support (M_{28}), and ϕ-Coefficient (M_1) are in the top of the partial order of all multiple-bug versions and at least in the top order of one of partial order of two or five bugs versions. Notice that $\phi - \text{Coefficient}(M_1)$ is not in the top order of both two or five bugs, but this measure is in the top order position when we evaluate all multi-bug versions..

We also plot the curve showing the proportion of code that is investigated (x-axis) versus the proportion of bugs localized (y-axis) for all multiple-bug versions. We split the large graphs into several smaller graphs so that measures that have similar accuracies would be grouped together, as shown in Figures 25–30. For each graph, we compare a number of association measures with Tarantula and Ochiai. Figure 25 shows a measure that could localize similar number of buggy versions as compared with Ochiai when less than 10% of code being inspected. The measure localizes more buggy versions than Tarantula and Ochiai when more than 40% of inspected code. Ochiai could localize 1% of multi-bug versions when 10% of code being inspected, whereas Tarantula could not localize any multi-bug version within 10% of code inspection. Measures that have similar performance with Tarantula are shown in Figure 26. Figures 27 and 28 show measures that could localize similar numbers of buggy versions as compared with Ochiai when less than 10% of code is inspected and also have similar performance with Tarantula and Ochiai. Measures that perform worse than Tarantula and Ochiai are shown in Figures 29 and 30. In this paper, we omit showing the curves for versions that contain five bugs because the curves are similar to the curves for all multiple-bug versions. For localizing five-bug programs, measure that have similar performance as the measure in Figure 25 is Relative Risk (M_{23}).

We also plot the curves showing the proportion of code that is investigated (x-axis) versus the proportion of bugs localized (y-axis) for versions that contain two bugs, as shown in Figures 31–36. Figures 31 shows measures that could localize more bugs as compared with Tarantula and Ochiai when more than 10% of code is inspected. Measures that have similar

Table XXIV. Measures that are at the top of the partial orders for different types of multi-bug versions.

Num. of Bugs in a Version	Top Measures
Two and five bugs	Ochiai, Added Value (M_{15}), Relative Risk (M_{23}), Ochiai II (M_{40}), Collective Strength (M_{16}), One-Way Support (M_{27}), Two-Way Support (M_{28}), and φ-Coefficient (M_1)
Five bugs	Ochiai, Relative Risk (M_{23}), and Ochiai II (M_{40})
Two bugs	Ochiai, Added Value (M_{15}), Collective Strength (M_{16}), One-Way Support (M_{27}), Two-Way Support (M_{28}), Klosgen (M_{18}), Confidence (M_9), Interest (M_{12}), Interestingness Weighting Dependency (M_{24}), and Kappa (M_5)

The underline marks measures that are at the top of the partial orders of two types of multiple-bug versions.

DOI: 10.1002/smr

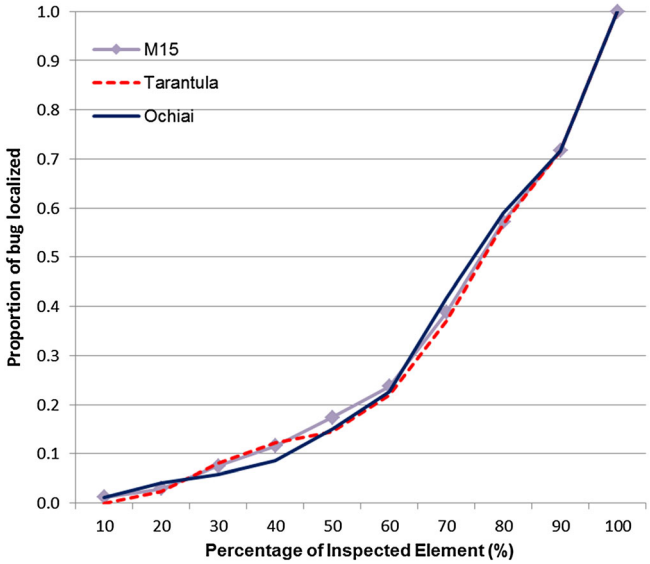

Figure 25. M_{15}, Ochiai, and Tarantula for all multiple-bug programs.

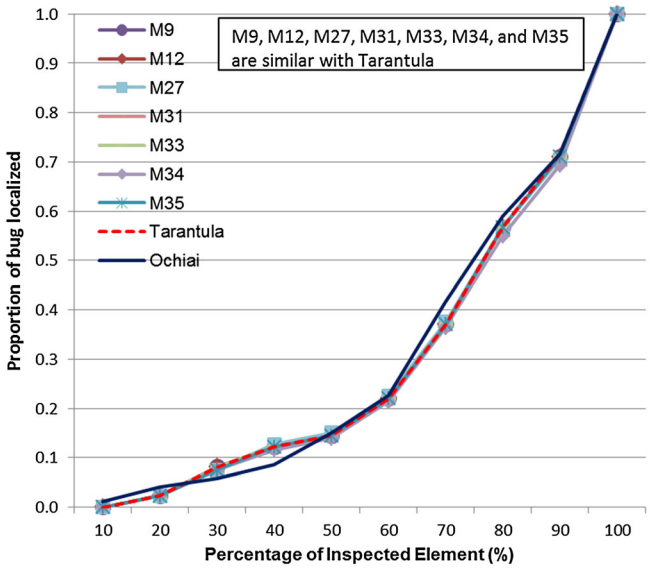

Figure 26. M_9, M_{12}, M_{27}, M_{31}, M_{33}, M_{34}, $M_{35}M_{15}$, Ochiai, and Tarantula for all multiple-bug programs.

performance with Tarantula are shown in Figure 32. Figure 33 shows measures that could localize more bugs as compared with Ochiai when more than 10% of code is inspected. Measures that perform worse than Tarantula and Ochiai are shown in Figure 34, 35, and 36. We summarize the findings of this section in the answer for RQ6 in the next Section.

5.7. Discussion

In this section, we summarize the answers to the research questions mentioned in Section 1.

RQ1. We are interested to find if off-the-shelf association measures are powerful enough to locate bugs. Based on the mean accuracy values of the measures, it could be noted that the 40 association measures could help to find all the bugs when an average of 25–58% of the program elements

DOI: 10.1002/smr

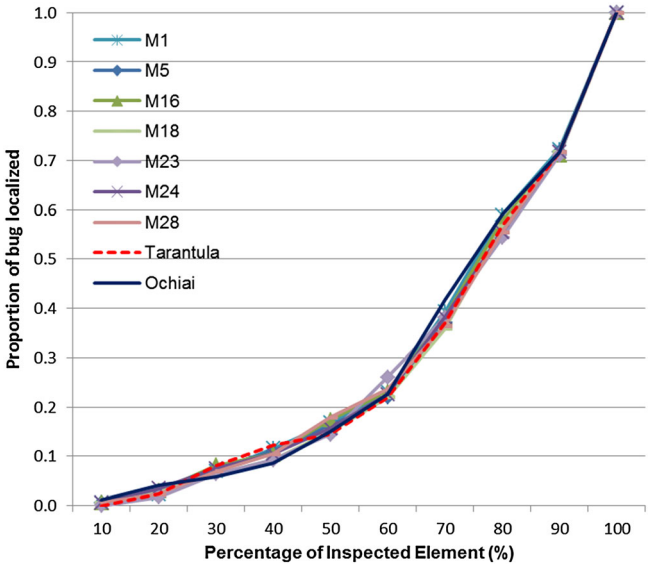

Figure 27. M_1, M_5, M_{16}, M_{18}, M_{23}, M_{24}, M_{28}, Ochiai, and Tarantula for all multiple-bug programs.

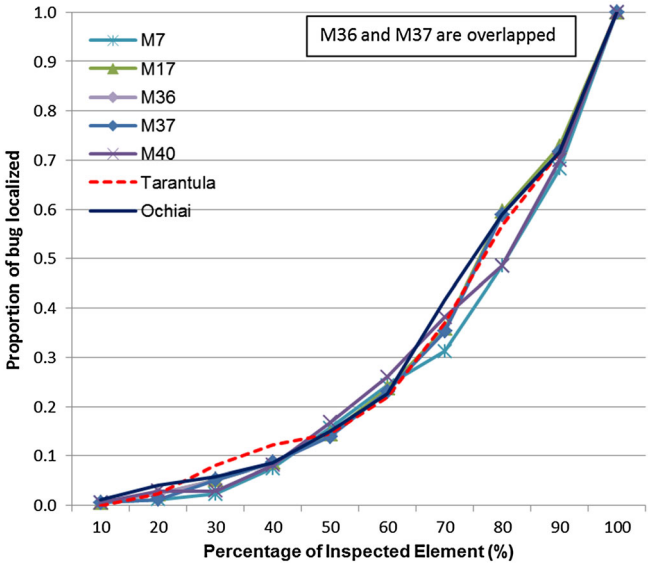

Figure 28. M_7, M_{17}, M_{36}, M_{37}, M_{40}, Ochiai, and Tarantula for all multiple-bug programs.

are inspected. Fifty percent of the association measures are able to help find bugs by inspecting an average of 25–27% of elements, whereas Tarantula and Ochiai require debuggers to inspect approximately 27 and 26% of program elements respectively f elements to find bugs.

RQ2. Next, we are interested to find which association measures are better than others in localizing single-bug programs. The answer to this research question is the partial order shown in Figure 2. At the top of the partial order there are two off-the-shelf association measures namely: Klosgen (M_{18}), and Normalized Mutual Information (M_{26}) that perform comparably to Ochiai. They are statistically significantly better than the other off-the-shelf association measures and Tarantula. There are seven other measures that perform statistically significantly better than Tarantula. These are: φ – Coefficient(M_1), Added Value (M_{15}), Collective Strength (M_{16}), Two-Way Support (M_{28}), Interestingness Weighting Dependency (M_{24}), Example and Counterexample Rate (M_{34}), and Kappa (M_5).

DOI: 10.1002/smr

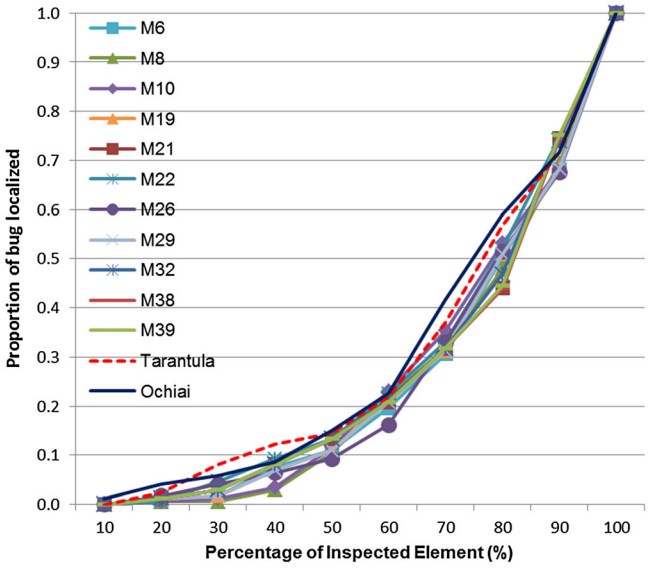

Figure 29. M_6, M_8, M_{10}, M_{19}, M_{21}, M_{22}, M_{26}, M_{29}, M_{32}, M_{38}, M_{39}, Ochiai, and Tarantula for all multi-bug programs.

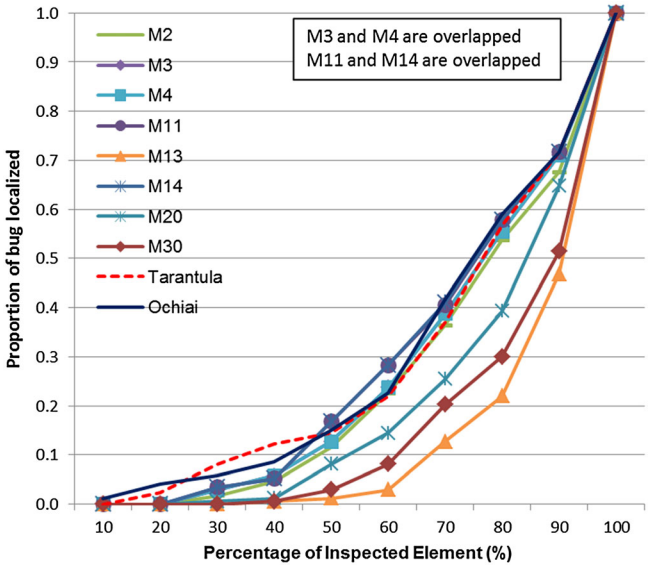

Figure 30. M_2, M_3, M_4, M_{11}, M_{13}, M_{14}, M_{20}, and M_{30}, together with Ochiai and Tarantula for all multi-bug programs.

RQ3. Finally, we would like to know the relative accuracy of the association measures versus those of well-known suspiciousness measures for fault localization for single-bug versions. By applying statistical significance tests under 0.05 significance threshold, Klosgen (M_{18}) and Normalized Mutual Information (M_{26}) are comparable with Ochiai and are statistically significantly better than Tarantula. Based on the proportion of bugs localized, Klosgen (M_{18}) and Added Value (M_{15}) could localize more bugs than Tarantula and Ochiai by inspecting up to 10% of the program elements. They could localize 47 and 48% of the bugs respectively, whereas Tarantula and Ochiai could localize 39 and 45% of the bugs respectively.

RQ4. We find that most measures perform better for the C programs than the Java programs.

DOI: 10.1002/smr

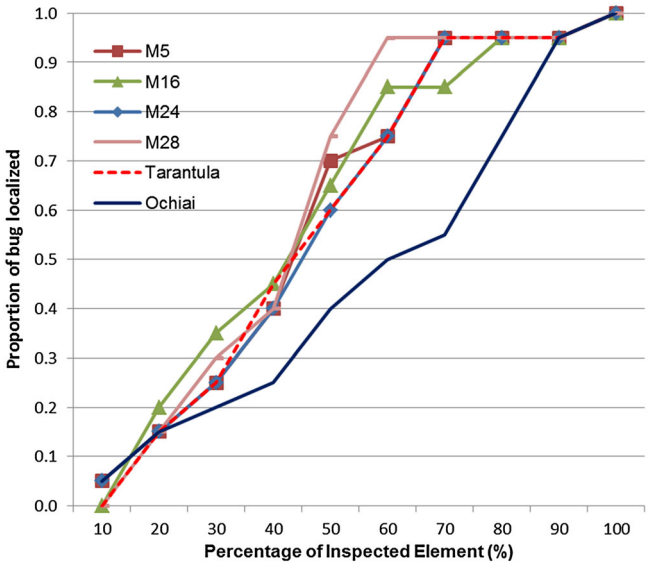

Figure 31. M_5, M_{16}, M_{24}, M_{28}, Ochiai, and Tarantula for versions containing two bugs.

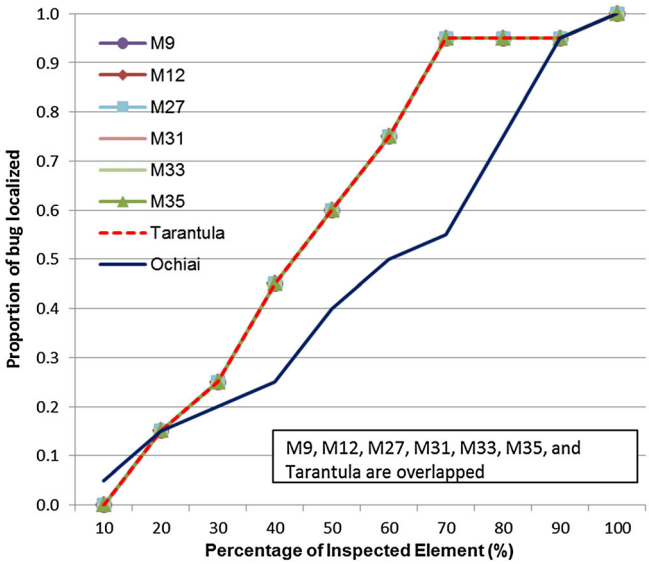

Figure 32. M_9, M_{12}, M_{27}, M_{31}, M_{33}, M_{35}, Ochiai, and Tarantula for versions containing two bugs.

To evaluate the measures in terms of percentage of code inspected, we compute two partial orders of the 40 association measures, Tarantula, and Ochiai, by performing statistical significance tests. For the C programs, Ochiai, and Klosgen (M_{18})) are the measures that are at the top of the partial orders (i.e., no measures are statistically significantly better than them). For the Java programs, a number of measures are at the top including Ochiai, Klosgen (M_{18}), Confidence (M_{18}), Interest (M_{12}), Added Value (M_{15}), Sebag (M_{31}), Example and Counterexample Rate (M_{34}), Zhang (M_{35}), Tarantula, Interestingness Weighting Dependency (M_{24}), and Normalized Mutual Information (M_{26}). In terms of proportion of bugs localized when up to 10% of code is inspected, for the C programs, Klosgen (M_{18}) and Added Value (M_{15}) outperform Ochiai and Tarantula. Tarantula and Ochiai could localize 43 and 52% of the bugs respectively while these measures could localize 56% of the bugs. For the Java programs, Tarantula, Confidence (M_9), Interest (M_{12}), Added Value (M_{15}), Klosgen (M_{18}), Relative Risk (M_{23}), Loevinger (M_{30}), Normalized Mutual Information (M_{26}), Odd Multiplier (M_{33}),

DOI: 10.1002/smr

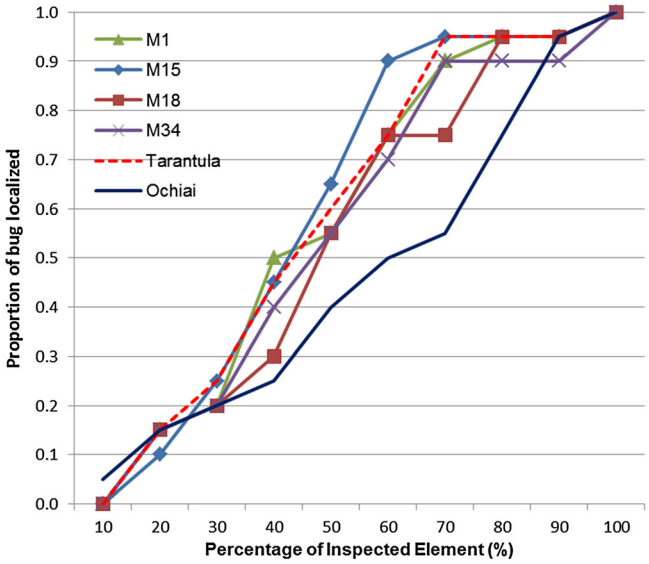

Figure 33. M_1, M_{15}, M_{18}, M_{34}, Ochiai, and Tarantula for two-bug versions.

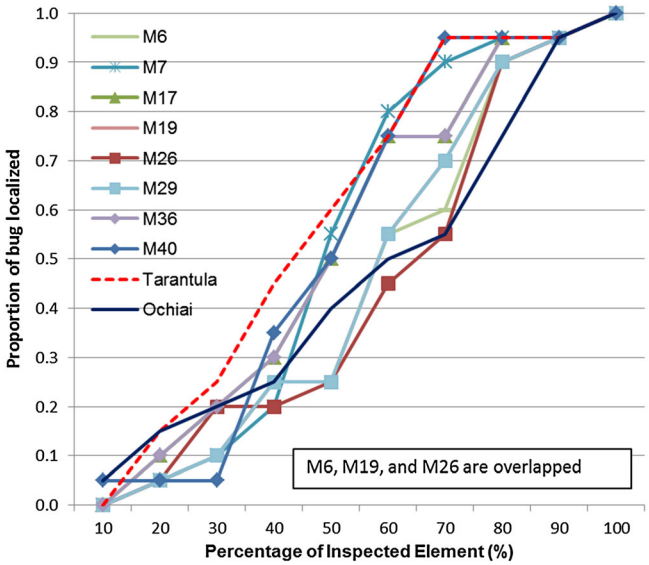

Figure 34. M_6, M_7, M_{17}, M_{19}, M_{26}, M_{29}, M_{36}, Ochiai, and Tarantula for two-bug versions.

Example and Counterexample Rate (M_{34}), and Least Contradiction (M_{32}) could localize the most number of bugs (i.e., 26% of the bugs could be localized). Ochiai, on the other hand, could only localize 20% of the bugs.

RQ5. In terms of percentage of code inspected, we also compute several partial orders (one per bug category) by performing statistical significance tests. Tarantula is at the top of three partial orders. Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{24}), and Two-Way Support Variation (M_{29}) are at the top of six partial orders. Klosgen (M_{18}) is at the top of seven partial orders. In terms of proportion of bugs localized, we notice that Klosgen (M_{18}) could localize the most number of bugs for all categories as compared with other measures when up to 10% of the program elements are inspected. The categories of bugs that could be better localized by most of the measures are (CH-CS), (LP-CC), and (CH-RET).

DOI: 10.1002/smr

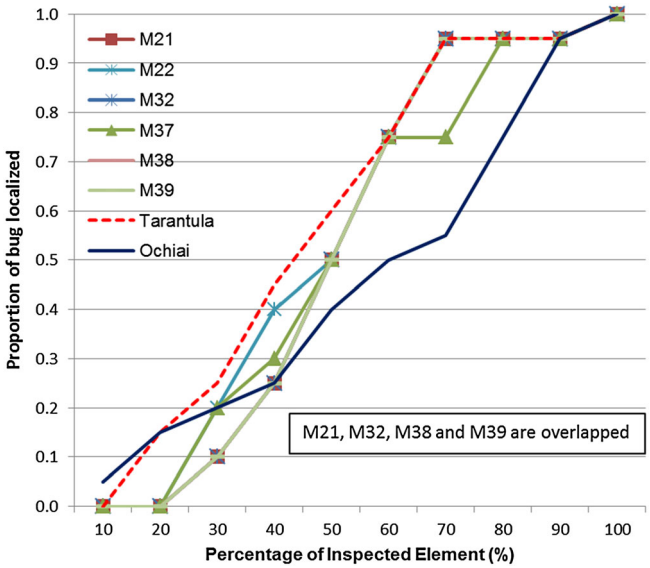

Figure 35. M_{21}, M_{22}, M_{32}, M_{37}, M_{38}, M_{39}, Ochiai, and Tarantula for two-bug versions.

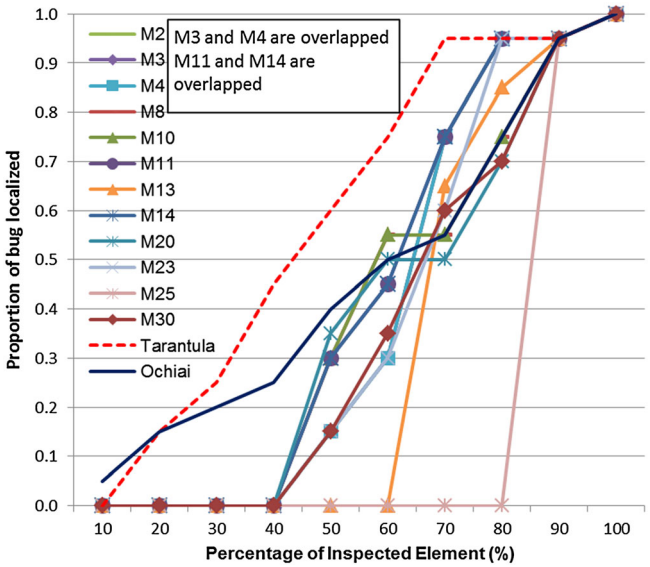

Figure 36. $M_2 - M_4$, M_8, $M_{10} - M_{14}$, M_{20}, M_{23}, M_{25}, M_{30}, M_{40}, Ochiai and Tarantula for two-bug versions.

RQ6. When localizing multiple bugs in programs, no measure performs as good as localizing single bugs in the programs. The percentage of code inspected required to localize all multi-bug versions, versions containing five bugs, and versions containing two bugs are 74–91%, 76%–92%, and 43–84%, respectively. Measures that are at the top of the partial order for localizing all multiple-bug versions are Added Value (M_{15}), Collective Strength (M_{16}), Ochiai, Ochiai II (M_{40}), One-Way Support (M_{27}), Relative Risk (M_{23}), Two-Way Support(M_{28}), and ϕ – Coefficient(M_1).

We notice that Added Value (M_{15}) and Collective Strength (M_{16}) also have good accuracies in localizing single-bug programs. The overall mean of percentage of code inspected for Added Value (M_{15}) and Collective Strength (M_{16}) are 25.74 and 25.66% respectively, which are only slightly smaller than the smallest mean of percentage of code inspected to localize all single-bug programs

DOI: 10.1002/smr

(25.24%). Also, Added Value (M_{15}) and Collective Strength (M_{16}) are in the top of partial order of five and four bug categories out of seven categories.

On the basis of our results, we find that there is no single best measure for all cases. For localizing C and Java programs containing a single bug, Klosgen (M_{18}) could localize the bug with the on average smallest percentage of code inspected. Also, when we evaluate the effectiveness of the measures to localize different bug categories, there is no other measures that could outperform Klosgen (M_{18}) for all bug categories. Although localizing multiple bugs, Added Value (M_{15}) could localize bugs with the on average smallest percentage of code inspected. We notice that Added Value also has good accuracy in localizing single bugs; its accuracy is only slightly lower than Ochiai and Klosgen.

Localizing bugs in multi-bug programs require much more code to be inspected than localizing bugs in single-bug programs, which can provoke future research. Also, it can be interesting future work to explore ways to compose various measures together so that the combined *meta-measure* may perform better for all cases than individual measures.

5.8. Threats to validity

Threats to construct validity refer to the suitability of our evaluation criteria. In this work, we use two criteria: percentage of program elements inspected to find all bugs and proportion of bugs localized when at most a given percentage of program elements are inspected. We believe these two evaluation criteria reasonably measure the effectiveness of a fault localization approach. They have also been used before in prior studies on fault localization [4, 16].

Threats to internal validity include bias and human errors. The accuracy of a measure in localizing bugs is influenced by the granularity level considered during program instrumentation and trace generation (statement, basic block, or method levels). Different granularity levels may produce different accuracies because there would be different total numbers of elements, which would affect the percentages of inspected elements. We choose to use block-hit spectra in our evaluation because it has a suitable balance between instrumentation costs and bug-reveal powers, and the focus of our study is to compare the effectiveness of different association measures on the *same* spectra. We hypothesize that the relative performance of different association measures on other spectra may remain the same as that on block-hit spectra, but it remains interesting future work for us to verify. Also, we manually instrument the C programs; we might miss instrumenting some blocks or add extraneous instrumentation code. For Java program, we automatically instrument the programs where there could be possible implementation errors. We manually assign the bugs into categories; there might be some errors in our assignments. In order to minimize such errors, we carefully checked the instrumented programs and assigned bug category labels.

Threats to external validity refer to the generalizability of our findings. We have tried to reduce this threat by considering a number of programs of various sizes written in two popular programming languages, C and Java. In the future, we would like to reduce this threat further by analyzing more programs written in various programming languages.

6. CONCLUSION

In this work, we investigate the effectiveness of a comprehensive number of association measures for fault localization. These measures gauge the strength of association between two variables expressible as a dichotomy matrix. We consider and compare 40 association measures with two well-known fault localization measures, namely Tarantula and Ochiai.

For programs that contain a single bug, we find that Klosgen outperforms Ochiai and Tarantula in terms of the average percentage of code inspected. Many measures, including Normalized Mutual Information, J-Measure, Information Gain, Two-Way Support Variation, Example and Counterexample Rate, Confidence, Kappa, Sebag, Odd Multiplier, Interest, Zhang, and One-Way Support outperform Tarantula. The percentages of code inspected for different buggy program versions are different; such percentage values for each measure form a distribution, and we employ statistical significance tests to

DOI: 10.1002/smr

compare the accuracy of different measures. We find that three measures, Klosgen, Normalized Mutual Information, and Ochiai can be statistically significantly better than other measures when localizing single-bug program. In terms of proportions of bugs found when up to 10% of code is inspected, Klosgen outperforms Ochiai. Also, Klosgen, Ochiai, ϕ-Coefficient, Added Value, Collective Strength, Two-Way Support, Interestingness Weighting Dependency, Example and Counterexample Rate, and Kappa outperform Tarantula. Thus, we can conclude that association measures are also promising to be used for fault localization.

We find that most measures perform better for the C programs than the Java programs. For the C programs, in terms of proportions of bugs localized, Added Value and Klosgen outperform Ochiai and Tarantula. For the Java programs, in terms of proportions of bugs localized, Tarantula, Confidence, Interest, Added Value, Klosgen, Relative Risk, Loevinger, Normalized Mutual Information, Odd Multiplier, Example and Counterexample Rate, and Least Contradiction measures, outperform the other measures and Ochiai.

We have also grouped the bugs into seven categories and analyze the effectiveness of the measures to localize each category of bugs. The categories of bugs that could be better localized by most of the measures are (CH-CS), (LP-CC), and (CH-RET). Klosgen is among the best measures for all bug categories.

The effectiveness of all measures in localizing multiple bugs in programs is not as good as localizing buggy programs that contain a single bug. Added value could localize the bugs with the smallest percentage of inspected code. There are also other measures that have comparable performance as Added Value, that is, Ochiai, Relative Risk, Ochiai II, Collective Strength, One-Way Support, Two-Way Support, and ϕ-Coefficient.

On the basis of our results, we find that there is no single best measure for all cases. Although there are several measures that perform statistically comparable for localizing single-bug programs, Klosgen could localize bugs with the smallest on average percentage of code inspected. As for localizing multi-bug programs, Added Value could localize bugs with the smallest on average percentage of code inspected, but a number of measures have comparable performance as well. Thus, solely based on fault localization accuracy, Klosgen, and Added Value can be best measures to use. However, for a real-world situation where many bugs may co-exist in the same large program, one may still have to take many factors into consideration when choosing a right measure. For example, because the accuracies of the measures in localizing multi-bug programs are much lower than localizing single-bug programs, more future research may be needed to improve fault localization for multi-bug programs before it can be practically useful. Also, as pointed out by Parnin and Orso [40], showing the right contexts for developers to understand the root causes of a bug may be more important for automated debugging techniques to be useful than purely pursuing localization accuracy. In the future, we would like to integrate promising association measures for fault localization into popular Integrated Development Environments (IDEs) and debugging tools such as Eclipse and Visual Studio Net to help improve the effectiveness and impact of fault localization techniques.

Dataset

Our dataset and tool are made publicly (Available from: http://www.mysmu.edu/phdis2009/lucia.2009/jsme/Dataset.htm).

REFERENCES

1. Tassey G. The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology. Planning Report 02-3*.2002 2002.
2. Beizer B. *Software Testing Techniques*. 2nd edn. International Thomson Computer Press: Boston, 1990.
3. Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI. Scalable Statistical bug Isolation. *Proc. ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, 2005; 15–26.
4. Jones J, Harrold MJ. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. *Proc. of International Conference on Automated Software Engineering (ASE'05)*, 2005; 273–282.
5. Renieris M, Reiss S. Fault Localization with Nearest Neighbor Queries. *Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, 2003; 141–154.

DOI: 10.1002/smr

6. Manevich R, Sridharan M, Adams S, Das M, Yang Z. PSE: Explaining Program Failures via Postmortem Static Analysis. *Proc. of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'04)*, 2004; 63–72. URL citeseer.ist.psu.edu/manevich04pse.html.
7. Ko AJ, Myers BA. Debugging Reinvented: Asking and Answering why and why not Questions About Program Behavior. *Proc. of the 30th International Conference on Software Engineering (ICSE'08)*, 2008; 301–310.
8. Gupta N, He H, Zhang X, Gupta R. Locating Faulty Code Using Failure-Inducing Chops. *ASE*, 2005; 263–272.
9. Mayer W, Stumptner M. Model-based debugging – state of the Art and future challenges. *Electronic Notes in Theoretical Computer Science (ENTCS'07)* 2007; 61–82.
10. Zeller A. *Why Programs Fail: A Guide to Systematic Debugging* (2nd edition) Morgan Kaufmann, 2009.
11. Qi D, Roychoudhury A, Liang Z, Vaswani K. *Darwin: An Approach for Debugging Evolving Programs*, *Proc. of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE'09)*, 2009; 33–42.
12. Liu C, Yan X, Fei L, Han J, Midkiff S. Sober: Statistical Model-Based bug Localization. *Proc. of Joint Meeting of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE'05)*, 2005; 286–295.
13. Abreu R. Spectrum-Based Fault Localization in Embedded Software. PhD Thesis, Delft University of Technology 2009.
14. Chilimbi TM, Liblit B, Mehra K, Nori AV, Vaswani K. Holmes: Effective Statistical Debugging via Efficient Path Profiling. *Proc. of IEEE 31st International Conference on Software Engineering (ICSE'09)*, IEEE Computer Society, 2009; 34–44, doi:10.1109/ICSE.2009.5070506.
15. Jones J, Harrold MJ, Stasko J. Visualization of Test Information to Assist Fault Localization. *Proc. of the 22nd International Conference on Software Engineering (ICSE'02)*, 2002; 467–477.
16. Abreu R, Zoeteweij P, van Gemund AJC. On the Accuracy of Spectrum-Based Fault Localization. *Mutation Testing: Academic and Industrial Conference Practice and Research Techniques (TAICPART-MUTATION'07)*, 2007.
17. Abreu R, Zoeteweij P, van Gemund AJ. Spectrum-Based Multiple Fault Localization. *Proc. of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, 2009; 88–99.
18. Agresti A. *An Introduction to Categorical Data Analysis*. John Wiley & Sons, 1996.
19. Yule GU. On the association of attributes in statistics. *Philosophical Transcriptions of the Royal Society* 1900; **A194**: 257–319.
20. Yule GU. On the methods of measuring association between two attributes. *Journal of the Royal Statistical Society* 1912; **75**:579–642.
21. Abreu R, Zoeteweij P, Golsteijn R, van Gemund AJC. A practical evalution on spectrum-based fault localization. *The Journal of System and Software* 2009; **82**:1780–1792.
22. Harrold MJ, Rothermel G. Siemens Programs, HR Variants. (Available from: http://www.cc.gatech.edu/aristotle/Tools/subjects/)
23. Do H, Elbaum SG, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal* 2005; **10**(4):405–435.
24. Liblit B, Aiken A, Zheng AX, Jordan MI. Bug Isolation via Remote Program Sampling. *Proc. ACM SIGPLAN* 2003 *Conf. Programming Language Design and Implementation (PLDI 2003'03)*, 2003; 141–154.
25. Santelices R, Jones J, Yu Y, Harrold MJ. Lightweight Fault-Localization Using Multiple Coverage Types. *Proc. of the 31st International Conference on Software Engineering (ICSE'09)*, 2009; 56–66.
26. Artzi S, Dolby J, Tip F, Pistoia M. Directed Test Generation for Effective Fault Localization. *Proc. of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*, 2010; 49–60.
27. Artzi S, Dolby J, Tip F, Pistoia M. Practical Fault Localization for Dynamic web Applications. *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, 2010; 265–274.
28. Bandyopadhyay A, Ghosh S. On the Effectiveness of the Tarantula Fault Localization Technique for Different Fault Classes. *Proc. of the 14th International IEEE Symposium on High-Assurance Systems Engineering (HASE'11)*, 2011; 317–324.
29. Pan K, Kim S, Je JW. Toward understanding bug fix patterns. *Empirical Software Engineering* 2009; **14**: 286–315.
30. Jiang B, Chan WK, Tse TH. On Practical Adequate Test Suites for Integrated Test Case Prioritization and Fault Localization. *Proc. of the 11th International Conference on Quality Software (QSIC'11)*, 2011; 21–30.
31. Zhang X, Gupta N, Gupta R. Locating Faults Through Automated Predicate Switching. *Proc. of the 28th International Conference on Software Engineering (ICSE'06)*, 2006; 272–281.
32. Sterling CD, Olsson RA. Automated bug isolation via program chipping. *Software: Practice and Experience (SP&E)* 2007; **37**(10): 1061–1086. John Wiley & Sons, Inc.
33. Jeffrey D, Gupta N, Gupta R. Fault localization using value replacement. *Proc. of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08),* 2008; 167–178.
34. Feldman A, van Gemund A. A two-Step Hierarchical Algorithm for Model-Based Diagnosis. *Proceedings of the 21st National Conference on On Artificial Intelligence*, AAAI Press: Boston, Massachusetts, 2006; 827–833.
35. Mayer W, Stumptner M. Evaluating Models for Model-Based Debugging. *Prof. of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)E*, 2008; 128–137.
36. Mayer W, Stumptner M. Abstract Interpretation of Programs for Model-Based Debugging. *Proc. of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, 2007; 471–476.

DOI: 10.1002/smr

37. Mayer W, Abreu R, Stumptner M, van Gemund A. Prioritising Model-Based Debugging Diagnostic Reports. *Proceedings of the 20th International Workshop on Principles of Diagnosis (DX'08)*, 2008; 127–134.
38. Liblit B, Aiken A. Building a Better Backtrace: Techniques for Postmortem Program Analysis. *Technical Report CSD-02-1203*, UC Berkeley 2002.
39. Tallam S, Tian C, Gupta R. Dynamic Slicing of Multithreaded Programs for Race Detection. *Proc. of the 24th IEEE International Conference on Software Maintenance (ICSM'08)*, 2008; 97–106.
40. Parnin C, Orso A. Are Automated Debugging Techniques Actually Helping Programmers? *Proc. of the 20th International Symposium on Software Testing and Analysis (ISSTA'11)*, 2011; 199–209.
41. Altman RB, Klein TE. Challenges for Biomedical Informatics and Pharmacogenomics. *Annual Review of Pharmacology and Toxicology* 2002; **42**:113–133.
42. Healey JF. *Statistics: A Tool for Social Research*, 8th edn. Wadsworth Publishing, 2008. (Available from: http://www.amazon.com/Statistics-Research-Joseph-F-Healey/dp/ 0495096555)
43. Agrawal R, Srikant R. Fast Algorithms for Mining Association Rules. *Proc. of the 20th International Conference on Very Large Data Bases (VLDB'94)*, 1994; **1215**:487–499.
44. Tan PN, Kumar V, Srivastava J. Selecting the Right Interestingness Measure for Association Patterns. *Proc. 2002 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'02)*, Edmonton, Canada, 2002; 32–41.
45. Geng L, Hamilton H. Interestingness measures for data mining: A survey. *ACM Computing Surveys* 2006; **38**(3):9.
46. Reps T, Ball T, Das M, Larus J. The use of program profiling for software maintenance with applications to the year 2000 problem. *Proc. of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE'97)* 1997; 432–449.
47. Harrold M, Rothermel G, Sayre K, Wu R, Yi L. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability* 2000; **10**(3):171–194.
48. Cohen J. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement* 1960; **20**(1): 37–46.
49. Smyth P, Goodman R. An information theoretic approach to rule induction from databases. *IEEE Trans. Knowledge and Data Eng.* 1992; **4**(4):301–316.
50. Gini C. Variability and mutability, contribution to the study of statistical distributions and relations. *Studi Economico-Giuricici della R. Universita de Cagliari* 1912; **3**(part 2)(i-iii): 3–159.
51. Clark P, Boswell R. Rule Induction with cn2: Some Recent Improvements. *In Machine Learning - EWSL-91*, 1991; 151–163.
52. Brin S, Motwani R, Ullman JD, Tsur S. Dynamic Itemset Counting and Implication Rules for Market Basket Analysis. *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*, Tucson, AZ, 1997; 255–264.
53. Piatetsky-Shapiro G. Discovery, analysis, and presentation of strong rules. *Knowledge Discovery in Databases*. MIT Press, Cambridge. 1991; 229–238.
54. Shortliffe E, Buchanan B. A model of inexact reasoning in medicine. *Mathematical Biosciences* 1975; **23**:351–379.
55. Aggarwal C, Yu PS. A new framework for itemset generation. *Symposium on Principles of Database Systems (PODS'98)* 1998; 18–24.
56. Hand DJ, Mannila H, Smyth P. *Principles of Data Mining*. MIT Press, 2001.
57. Klosgen W. Explora: a multipattern and multistrategy discovery assistant. *Advances in Knowledge Discovery and Data Mining*. American Association for Artificial Intelligence. 1996; 249–271.
58. Quinlan J. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
59. Cheng H, Lo D, Zhou Y, Wang X, Yan X. Identifying bug signatures using discriminative graph mining. *Proc. of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)* 2009; 141–152.
60. Ohsaki M, Kitaguchi S, Okamoto K, Yokoi H, Yamaguchi T. Evaluation of Rule Interestingness Measures with a Clinical Dataset on Hepatitis. *Proc. of the 8th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD'04)*, 2004; 362–373.
61. Goodman L, Kruskal W. Measures of association for cross classifications. *Journal of the American Statistical Association* 1954; **49**:732–764.
62. Dice LR. Measures of the amount of ecologic association between species. *Ecology* 1945; **26**(3):297–302.
63. Sorensen T. A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on danish commons. *Vidensk Selsk Biol Skr* 1948; **5**(4):1–34.
64. Anderberg M. *Clustering Analysis for Applications*. Academic Press: London, 1973.
65. Sokal RR, Michener C. A statistical method for evaluating systematic relationships. *University of Kansas science bulletin* 1958; **38**(6):1409–1438.
66. Rogers J, Tanimoto TT. A computer program for classying plants. *Science* 1960; **132**(3434):1115–1118.
67. Ochiai A. Zoogeographic studies on the soleoid fishes found in japan and its neighbouring regions. *Bulletin of the Japanese Society of Scientific Fisheries* 1957; **22**:26–530.
68. Hutchins M, Foster H, Goradia T, Ostrand T. Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. *Proc. of the 16th International Conference on Software Engineering (ICSE'94)*, 1994; 191–200.
69. Wilcoxon F. Individual comparisons by ranking methods. *Biometrics Bulletin* 1945; **1**:80–83. (Available from: http://www.jstor.org/stable/3001968)

DOI: 10.1002/smr

AUTHORS' BIOGRAPHIES

Lucia is a currently PhD student in the School of Information System in Singapore Management University. She started her PhD program since 2009. Previously, she received her master degree in Information and Communication Technology from University of Wollongong, Australia in 2005. Her research is in software system and data mining area; particularly, she is interested in software fault localization, software testing, and frequent pattern mining.

David Lo received his PhD degree from the School of Computing, National University of Singapore in 2008. He is currently an assistant professor in the School of Information Systems, Singapore Management University. He works in the areas of software engineering and data mining. He is particularly interested in specification mining, debugging, software text analytics, frequent pattern mining, and social network mining. He is a member of the Institute of Electrical and Electronics Engineers and Association for Computing Machinery.

Lingxiao Jiang is currently an assistant professor in the School of Information Systems, Singapore Management University. He was from Zhejiang Province, China. He completed his PhD with Prof. Zhendong Su in the Department of Computer Science at University of California, Davis (2003–2009). He worked as a test strategist at Nvidia for half a year before he joined the faculty of School of Information Systems at Singapore Management University in November 2009. He is broadly interested in the area of software engineering, programming languages, and data mining. He has been spending substantial efforts on code analysis and comprehension, aiming to provide practical techniques and tools for improving software reliability, increasing development productivity, and reducing maintenance cost.

Ferdian Thung is currently a PhD student in the School of Information System, Singapore Management University. He started her PhD program in 2013. Previously, he received his bachelor degree in the School of Electrical Engineering and Informatics from Bandung Institute of Technology, Indonesia in 2011. He has worked as a research engineer for more than 1 year in the School of Information Systems, Singapore Management University. His research interest is in software system and data mining area. He has been working on automated prediction, recommendation, and empirical study in software systems.

Aditya Budi was a research engineer in the School of Information Systems, Singapore Management University, Singapore. He completed his Master of Science in Information System to Wee Kim Wee Schools of Information at Nanyang Technological University, Singapore in July 2011. He also had worked as a system analyst for 8 years. He is interested in software maintenance and architectures.

DOI: 10.1002/smr