Search-Based Fault Localization

Shaowei Wang, David Lo, Lingxiao Jiang, Lucia, and Hoong Chuin Lau
School of Information Systems
Singapore Management University
{shaoweiwang.2010, davidlo, lxjiang, lucia.2009, hclau}@smu.edu.sg

***Abstract*—Many spectrum-based fault localization measures have been proposed in the literature. However, no single fault localization measure *completely* outperforms others: a measure which is more accurate in localizing some bugs in some programs is less accurate in localizing other bugs in other programs. This paper proposes to compose existing spectrum-based fault localization measures into an improved measure. We model the composition of various measures as an optimization problem and present a search-based approach to explore the space of many possible compositions and output a heuristically near optimal composite measure. We employ two search-based strategies including genetic algorithm and simulated annealing to look for optimal solutions and compare the effectiveness of the resulting composite measures on benchmark software systems. Compared to individual spectrum-based fault localization techniques, our composite measures perform statistically significantly better.**

I. Introduction

Many approaches have been proposed to help in automating debugging process, especially in localizing faults [1]–[7]. One common family of approaches is spectrum-based fault localization, e.g., [2], [7], where program traces or abstractions of traces (called program spectra) of correct and failed executions are compared to identify likely program elements that causes failures and are reported as potential faults. Some well-known studies include Tarantula [2] and Ochiai [7]; Baah et al. [8] improve fault localization accuracy by using a linear model to estimate the causal effect of a program element on failures. Lucia et al. [9] have also investigated the effectiveness of more than 20 association measures, and report that *information gain* may on average perform the best for fault localization.

We observe that although one measure could be more accurate than other measures on some datasets, there are programs where a poorer measure performs better than a better one. We aim to leverage this fact by composing different measures into a composite one that could perform better than any individual constituent measure.

In this paper, we model the problem of finding an optimal composition of various measures as a search-based optimization problem. Our search-based fault localization engine works on two phases: training and deployment. In the training phase, the engine takes in a number of buggy programs along with actual fault locations to measure how good a particular composition is. A good composition should be able to locate many bugs in this training set with a high accuracy. Search heuristics (e.g. simulated annealing [10] and genetic algorithm [11]) would be employed to traverse the search space of all possible linear combinations of the more than 20 association measures to find near optimal ones that perform best on the training set. In the deployment phase, our engine takes a given buggy program and its spectra, and apply the composite measure to identify potential fault locations.

We evaluate our search-based fault localization engine on the Siemens test suite [12]. We compare the effectiveness of our proposed composite measure with its constituent measures, and show that our engine could locate relatively 51% and 22% more bugs than Tarantula and Ochiai respectively when only 10% of the program code is inspected, and that the average amount of code to be investigated to localize all bugs is reduced from 31.8% and 27.6% to 20.0% when compared to Tarantula and Ochiai respectively. With t-test, we also show that the improvements are statistically significant.

II. Related Work

Fault Localization. There are many studies on fault localization and automated debugging, which can be categorized in various ways. Based on the data used, fault localization techniques can be classified into *spectrum-based* and *model-based*. 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 [1]–[6] take as inputs two sets of spectra, one for successful executions and the other for failed executions, and report candidate locations where causes of program failures (i.e., faults) may reside. Given a failed program spectrum and a set of correct spectra, WHITHER [3] contrasts the failed execution to the nearest correct execution to find most suspicious locations. Delta debugging [1], [5], [13] uses binary-search-like algorithms to locate failure causes. Recently, Artzi et al. [14], [15] propose a directed test generation technique to generate appropriate test cases and adapt Ochiai and Tarantula for debugging web applications. They achieve significantly higher fault localization accuracy since their approach utilizes a mapping that could map different parts of an output to different parts of the program. This mapping is very useful for accurate fault localization when the output is of rich structures (e.g., HTML pages in web applications) and different parts of the output are generated by different parts of the program code. The mapping is unlikely to be useful for locating faulty code in a program that outputs only simple structures or even a single number—the code responsible for producing the output might be the entire program.

Other spectrum-based techniques, e.g., [16], only use failed executions as the input and systematically alter the program structure or program runtime states to locate faults.

Different from these studies, we focus on a compositional framework that combines any spectrum-based fault localization technique which can sort program elements by their suspicious scores, and we demonstrate that the framework with search and optimization algorithms and 22 existing fault localization measures can find better composite measures.

Search-based Algorithms in Software Engineering. Search-based techniques have been frequently used in testing. Various studies have proposed search-based test data generation approaches [17], [18]. In this work, similar to the above approaches, we also utilize several search algorithms. However, we address a different problem, and to our best knowledge, we are the first to use search algorithms to find optimal compositions from existing individual fault localization techniques.

III. Search-Based Composition Engine

Our search-based fault localization process is divided into two phases: training and deployment. The two phases are illustrated in Figures 1 & 2.

A. Training Phase

In the training phase, we take as input all of 20 association measures and Tarantula and Ochiai investigated in [9], along with a training dataset. The training data is a set of program spectra along with actual bug locations, which could be obtained in practice from past fault localization efforts or known bugs and fixes in programs. Based on the 22 measures and the training data, the search algorithms investigate various composite measures and search for one that performs the best in localizing faults in the training dataset.

In the following paragraphs, we first define the search space of potential composite measures. Next, we describe how we adapt existing search algorithms for fault localization purpose. The output of the training phase is a heuristically near optimal composite measure that performs well for the training data.

1) Search Space: In this work, we employ one of the many possible composition strategies, which is a linear model represented as a weighted sum of individual fault localization measures. Given 22 measures named as $M_1, M_2, \ldots, M_{22}$, and a program element e, the suspiciousness score for e assigned by the composite measure is defined as follows:

$$M_{Composite}(e) = w_1 \times M_1(e) + w_2 \times M_2(e) + \ldots + w_{22} \times M_{22}(e)$$

The search space of all possible compositions corresponds to the various assignments of weights, $w_1, w_2, \ldots, w_{22}$, and each weight could be a real number from zero to one. However, many search algorithms only work on discrete search spaces, thus we discretize the search space by representing each weight by 7 bits. The function mapping the 7 bit signature to a weight is:

$$BitToWeight(b_1 b_2 b_3 b_4 b_5 b_6 b_7) = \frac{\Sigma(b_i \times 2^i)}{2^7 - 1}$$

2) Objective Functions: Search algorithms require an objective function to quantify the quality of potential solutions. We define the objective function f so that the lower the score is, the better quality a fault localization measure has, and the goal for search algorithms is to find candidate compositions (*i.e.*, assignments to the weights) that minimize f.

Since the quality of a fault localization technique is often defined as how much code is needed to be investigated to find certain numbers of bugs, we define f as the average proportion of program elements that need to be investigated to locate all bugs in a set of programs when lists of program elements sorted in the descending order according to their suspiciousness scores generated by the measure is traversed.

3) Adapting Search Algorithms: We consider two search algorithms in our compositional framework in this paper: simulated annealing [10] and genetic algorithm [11].

Simulated Annealing. With the search space modeled in Section III-A1 and the objective function defined f defined in Section III-A2 as the objective function, simulated annealing (SA) can be applied straightforwardly for our problem. Similar to previous work employing search-based techniques in software engineering tasks [19], we use some initial experiments to set the parameters for SA.

Genetic Algorithm. To use genetic algorithm (GA) for our problem, we further transform the search space modeled in Section III-A1 to chromosomes required by GA: we concatenate the 22 7-bit weights together to form 154-bit chromosomes. The search space is thus all possible binary chromosomes of length 154.

In addition, GA consists of the creation of an initial population, and the selection, crossover, and mutation operations. We created the initial population of chromosomes randomly. The crossover and mutation operations are standard [11]. We only define the selection step, and the intuition is to select the most fit chromosomes in each generation based on f. In this paper, we define the *fitness* of a chromosome X_i in a generation R containing $|R|$ chromosomes as follows:

$$Fitness(X_i) = sum_{j=1}^{|R|} e^{[10 \times (f(X_j) - f(X_i))]}$$

The definition implies that the smaller $f(X_i)$ is among all chromosomes in the same generation, the better fit X_i is. Since the difference between $f(X_j)$ and $f(X_i)$ can be very small, we take the exponentiation of this difference multiplied by ten.

With the fitness function, we consider two selections:

1) **Random.** In this strategy, each chromosome of the current generation has a chance to be selected to be a chromosome in the next generation proportional to its fitness score.
2) **Enhanced.** In this strategy, we set a threshold k. Then, if the objective function value of a chromosome is less

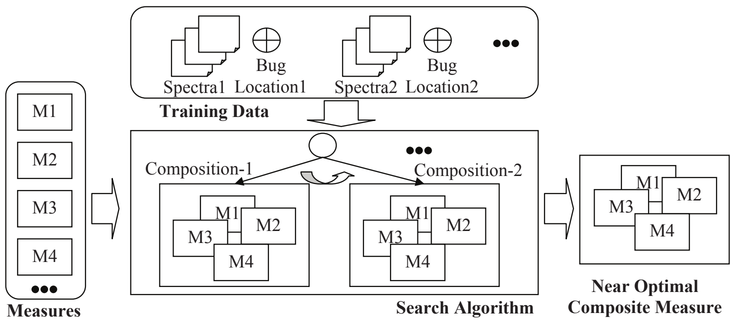

Fig. 1. Training Phase

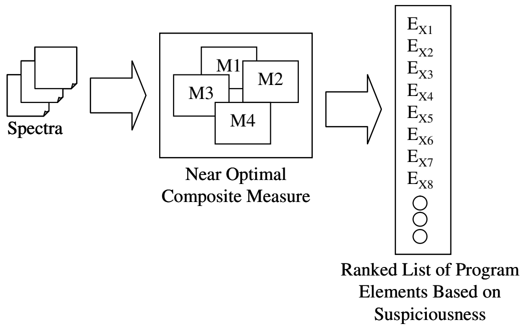
Spectra
M1
M2
M3
M4
Near Optimal Composite Measure
E_{X1}
E_{X2}
E_{X3}
E_{X4}
E_{X5}
E_{X6}
E_{X7}
E_{X8}
Ranked List of Program Elements Based on Suspiciousness

Fig. 2. Deployment Phase

than k away from that of the best chromosome in the current generation, it is put into the next generation; otherwise, the chromosome is selected as in the random strategy. The intuition for keeping better chromosomes for next generations is to achieve potentially better fault localization results within the same number of iterations.

We refer to GA with the *Random* selection as GA_{Random} and the other as $GA_{Enhanced}$. Similar to SA, we use some initial experiments to set the parameters of GA.

B. Deployment Phase

In the deployment phase, the composite measures learned in the training phase are used to locate faults. Given a program spectrum, a composite measure assigns a suspiciousness score to each program element, as individual measures do; the higher the score is for a program element, the more likely the element is faulty. These suspiciousness scores are used to sort all program elements in a list and presented to code inspectors. The process is illustrated in Figure 2.

IV. Empirical Evaluation

In this section, we describe our experimental settings and evaluation results.

A. Settings

Subject Programs. We analyze seven programs from Siemens Test Suite [12]. 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 http://www.cc.gatech.edu/aristotle/Tools/subjects/. The test suite contains seven programs, including print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info. Each program contains many different versions where each version has one bug. These bugs comprise a wide array of realistic bugs. The total number of buggy versions are 132. Among these versions, we use 120 versions.

Instrumentation. We instrument the programs and run them with the test cases to collect program block-hit spectra. Each block-hit spectrum identifies how many times each block is executed when a program is executed with a test case.

Cross Validation. We perform a three-fold cross validation (i.e., 2/3 of the dataset is used for training and 1/3 is used for testing) to evaluate the result of our search-based fault localization approach. We randomly split the 120 versions into 3 buckets. Each bucket would thus contain 40 versions. In three-fold cross validation, 3 separate iterations are performed. In each iteration, we keep one of the bucket as the test/validation set and use the other two buckets for training. The average results across the 3 iterations are reported.

TABLE I
Dataset Details

Dataset	LOC	Faulty versions	Test cases
print_token	472	7	4030
print_token2	399	10	4115
replace	512	32	5542
schedule	292	9	2650
schedule2	301	10	2710
tcas	141	41	1608
tot_info	440	23	1051

B. Evaluation Results

We measure the accuracies of the three variants of our proposed search-based approach, namely, SA, GA_{Random}, and $GA_{Enhanced}$, and compare them with those of Tarantula [2], Ochiai [20], and Information Gain which are among the best fault localization measures [9]. Here, the accuracy for each fault localization technique is measured in terms of number (or proportion) of bugs that could be localized by only examining a certain number (or propertion) of program elements (blocks in our case), which is also used in past studies [2], [5], [20].

We plot the accuracies of the various techniques in Figure 3. The x-axis describes the proportion of blocks inspected and y-axis describes the proportion of buggy versions that can be localized. Assuming an inspector can recognize a bug when a block is presented to him or her, the plot lines shall always reach 100% at the right ends. The more buggy versions localized at lower numbers of inspected blocks, the better a fault localization approach is.

We find that $GA_{Enhanced}$ and GA_{Random} perform better than all other techniques. When less than 10% of the blocks are inspected, $GA_{Enhanced}$ and GA_{Random} localize 52% of all buggy versions. At the same proportion of inspected blocks, SA localize 43% of the bugs respectively, while Ochiai, Tarantula and Information Gain localize 43%, 34%, and 37% respectively. The relative improvements of $GA_{Enhanced}$ and GA_{Random} over Tarantula, Ochiai, and Information Gain are 51%, 22%, and 41% respectively.

Furthermore, 65% of all bugs are localized by $GA_{Enhanced}$ with only up to 20% of blocks inspected. To achieve the same number of localized bugs, GA_{Random}, SA, and Ochiai require at least 20%, 28%, and 29% of block inspections respectively, while Tarantula requires at least 43%. Also, $GA_{Enhanced}$ and GA_{Random} can localize more than 90% of the bugs when up to 50% code elements are inspected. On the other hand, SA requires 60%, and Ochiai, Tarantula, and Information Gain require 67%, 75%, and 75% respectively.

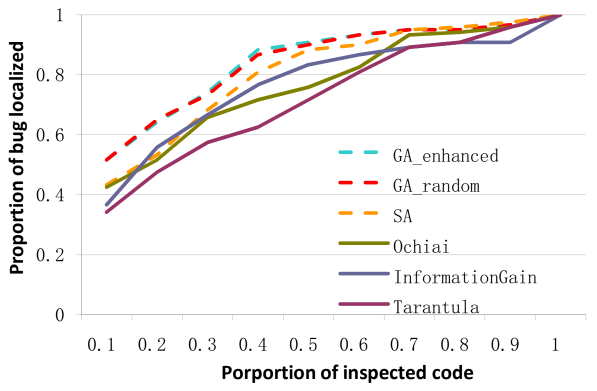

Fig. 3. Accuracies of Various Techniques

Looking from another angle where developers need to localize *all* faults, we examine the mean and standard deviation of the proportions of blocks that need to be inspected to localize all faults for each technique. The results are shown in Table II. $GA_{Enhanced}$ and GA_{Random} localizes all bugs with the lowest average proportion of blocks inspected (20.0%). The relative improvements of $GA_{Enhanced}$ over Tarantula, Ochiai, and Information Gain are 37%, 28%, and 28% respectively.

TABLE II
MEANS AND STANDARD DEVIATIONS OF PROPORTIONS OF INSPECTED BLOCKS (SMALLER THE BETTER)

Techniques	Mean	Standard Dev.
$GA_{Enhanced}$	20.0%	23.2%
GA_{Random}	20.0%	23.4%
SA	23.6%	24.2%
Ochiai	27.6%	27.9%
Information Gain	27.7%	28.6%
Tarantula	31.8%	29.1%

To check whether the differences among these means are statistically significant, we perform the commonly used t-test [21] for each pair of fault localization techniques with p-value set to 0.05. We notice that the improvements of $GA_{Enhanced}$ over Tarantula, Ochiai, and Information Gain are statistically significant.

V. CONCLUSION

There are many fault localization measures proposed in the literature. For some programs, bugs, and program spectra, some measures are better than the others. In this paper, we leverage this observation to build a solution that combines existing fault localization measures into an improved composite measure. We model the possible compositions as a search space in a linear model and adapt advances in the machine learning and metaheuristics community to compute a heuristically optimal composition based on a training set. We demonstrate that our approach can achieve better fault localization accuracy than existing spectrum-based fault localization techniques. Compared with Ochiai, Tarantula, and Information Gain, our approach with enhanced genetic algorithm can localize at least 22% more bugs when 10% of the programs is analyzed. Also, we can reduce the average amount of code investigated by at least 28%. Our t-tests also show that the improvements are statistically significant.

As future work, we plan to investigate more advanced genetic algorithm solutions. We also plan to increase the number of benchmark programs and perform a user study. In this work, we have only considered a linear model to compose individual fault localization measures; other composition models can also be explored in the future.

Acknowledgement. This work is partially supported by a research grant R-252-000-403-112 from Ministry of Education, Singapore.

REFERENCES

[1] A. Zeller, "Isolating cause-effect chains from computer programs," in *FSE*, 2002, pp. 1–10.
[2] J. Jones and M. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*, 2005.
[3] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," in *ASE*, 2003, pp. 141–154.
[4] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *PLDI*, 2003.
[5] H. Cleve and A. Zeller, "Locating causes of program failures," in *ICSE*, 2005.
[6] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff, "Sober: Statistical model-based bug localization," in *ESEC/FSE*, 2005.
[7] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, 2009.
[8] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *ISSTA*, 2010, pp. 73–84.
[9] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *ICSM*, 2010.
[10] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.
[11] J. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* Univ. of Michigan, 1975.
[12] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *ICSE*, 1994.
[13] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE TSE*, 2002.
[14] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *ISSTA*, 2010.
[15] ——, "Practical fault localization for dynamic web applications," in *ICSE*, 2010.
[16] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *ICSE*, 2006.
[17] P. McMinn, M. Harman, D. Binkley, and P. Tonella, "The species per path approach to search-based test data generation." in *ISSTA*, 2006.
[18] P. Tonella, "Evolutionary testing of classes." in *ISSTA*, 2004.
[19] Z. Li, M. Harman, and R. Hierons, "Search algorithms for regression test case prioritization," *IEEE TSE*, vol. 3, pp. 225–237, 2007.
[20] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *TAICPART-MUTATION*, 2007.
[21] D. Wackerly, W. Mendenhall, and R. Scheaffer, *Mathematical Statistics with Applications*, 7th ed. Duxbury Press, 2008.