A Mahalanobis Distance-Based Integration of Suspicious Scores For Bug Localization

Masanao Asato*, Hirohisa Aman†, Sousuke Amasaki‡, Tomoyuki Yokogawa‡ and Minoru Kawahara†
*Graduate School of Sc. & Eng., Ehime University, Matsuyama, Japan
†Center for Information Technology, Ehime University, Matsuyama, Japan
‡Faculty of Computer Sc. & Systems Eng., Okayama Prefectural University, Soja, Japan

***Abstract*—Once a software bug is reported, it is crucial to locate the source file causing the bug and fix it as soon as possible. To this end, there have been various studies for locating bugs with the version control system and the bug reports. AmaLgam+ is one of the most promising methods for localizing bugs. This method quantifies the degree to which a source file causes the bug from different five perspectives (metrics), and combines those values (suspicious scores) into a single integrated score of the source file. However, the method has a challenge regarding the computation time because it uses the genetic algorithm (GA) to combine the above five metrics. This paper proposes an application of the Mahalanobis distance to the suspicious score integration to overcome the above challenge. The proposed method considers the above five metrics to be a five-dimensional vector. It computes the Mahalanobis distance of the vector from the origin as an alternative integrated suspicious score. The empirical study using six open source software projects proves that the proposed method has almost the same bug localization accuracy as AmaLgam+ and can reduce the computation time by up to 98%: e.g., while AmaLgam+ took about 3.5 hours, the proposed method did it about 4 minutes.**

***Index Terms*—Bug localization, mining software repositories, Mahalanobis distance, computation time**

I. Introduction

When a bug is reported to the software development organization, the developers need to localize the faulty (buggy) source file that causes the reported bug. However, it may not be easy because the bug report may not always present an informative clue to detect the buggy file. Furthermore, large-scale and complex software products tend to be plagued by a lot of bug reports [1], [2]. A quick resolution of a newly-reported bug is a crucial challenge in software maintenance.

The detection of buggy files is referred to as "bug localization," and various approaches have been studied [3]. In recent years, many studies focused on information retrieval (IR) techniques, and proposed IR-based bug localization methods that apply IR methods to the bug reports and the software repositories [4]–[12]. For instance, some methods analyzed the similarity among bug reports using the natural language processing techniques such as tf-idf method and picked already-resolved bug reports similar to the new (unresolved) bug report [4], [9], [10]. They then considered the source files that had been fixed to resolve those old bugs to be bug-prone ones. Other methods also focused on various data such as the bug-fixing history recorded in the code repository, the stack trace data given in the bug report, the bug report reporter.

Recently, Wang and Lo [7] proposed a promising bug localization method, AmaLgam+. For a newly-reported bug, this method evaluates the suspiciousness of a source file from five different points of view, i.e., it measures the source file using five different metrics quantifying the degree to which the source file causes the reported bug. Then, AmaLgam+ combines these five evaluation values of a source file into an integrated suspicious score by the genetic algorithm (GA). Wang and Lo empirically proved that AmaLgam+ could outperform the conventional bug localization methods [7].

Although AmaLgam+ is one of the most promising bug localization methods, it has a challenge regarding the computational cost. Because AmaLgam+ uses GA to tune the weight coefficients to combine different metric values, it may require a long tuning time. Moreover, it would be better to rerun the tuning whenever the metric data set is updated. To perform a better bug localization as quickly as possible, we need to overcome that problem of computation time. Thus, we propose another way of integrating different metric values in this paper. Our proposal applies the Mahalanobis distance [13] notion, which is commonly used in anomaly detection studies. Because it can reasonably integrate different-scale metrics with a low computational cost, we consider the proposed method can be a lightweight alternative to AmaLgam+.

The remainder of this paper is organized as follows. Section II describes AmaLgam+, and Section III presents our proposal, the Mahalanobis distance-based method. Then, Section IV reports the empirical study that we conducted to examine the usefulness of the proposed method. Finally, Section V gives the conclusion of this paper and our future work.

II. Conventional Method: AmaLgam+

In this section, we briefly describe the bug localization by a conventional method, AmaLgam+ [7]. Once a bug report is given, it computes a suspicious score for each source file; the suspicious score is computed by combining five metric values.

A. Metric(1): Version History-Based Metric

A version control system (VCS) records the source code change history made during development and maintenance. Such a change history may be beneficial to an efficient bug localization because a prior bug fix may cause another bug (fault) [14]. Rahman et al. [15] proposed a lightweight method for predicting faulty files by focusing on the bug-fixing

commits in a VCS. Their method has been utilized in large-scale software development projects [16]. AmaLgam+ uses such a metric to quantify the suspicious level (fault-proneness) of a file. We describe the metric below.

Suppose we have a new (not resolved) bug report and let R be the set of bug-fixing commits that may relate to the reported bug. A related bug-fixing commit $c(\in R)$ is a commit satisfying the following two conditions 1) and 2).

1) The commit log of c includes "fix" or "bug."
2) c was made within the last k days. Because an older commit would be less likely to cause a newly-reported bug, this metric focuses only on the recent bug-fixing commits. Wang and Lo [7] set k to 15 empirically.

Let t_c be the elapsed days after commit c. Now the suspicious score of a file f, $score_1(f)$, is defined as follows:

$$score_1(f) = \sum_{c \in R \wedge f \in c} \frac{1}{1 + e^{12(1 - \frac{k - t_c}{k})}} \ . \tag{1}$$

B. Metric(2): Bug Report Similarity-Based Metric

When two bug reports have similar contents, they may be associated with each other and require to fix the same files to resolve both of the bugs. We apply this notion to the bug localization. Once we have a new bug report, we pick already-resolved bug reports similar to the new one and focus on the files fixed to resolve the above bugs. Those files may also be related to the newly-reported bug.

AmaLgam+ quantifies the similarity between bug reports using the *tf-idf* method in the following three steps.

1) *Pre-processing of bug reports*: Extract all tokens from a bug report. If a token is a compound word written in the camel-case style or the snake-case style, split it into subwords following the compounding style; Then, perform the stemming and the stop-word elimination.
2) *Vectorization of bug reports*: Compute the tf-idf value of each word. Then, express each bug report by the multi-dimensional vector whose elements are the tf-idf values of the words appearing in the report.
3) *Computation of bug report similarity*: Let r_{new} and r_{old} be the new bug report and a resolved one, respectively. Obtain the similarity between them $sim(r_{new}, r_{old})$ as the cosine similarity between the corresponding vectors.

Next, the suspicious score of a file is computed using the above similarity. If r_{new} is more similar to r_{old} and a file f had been fixed to resolve r_{old}, f is more suspicious that it may also cause the newly-reported bug. Hence, the suspicious score of f is computed as follows[1]:

$$score_2(f) = \sum_{r_i \in \{r | f \in mod(r)\}} \frac{sim(r_{new}, r_i)^2}{|mod(r_i)|} \ , \tag{2}$$

where $mod(r)$ is the set of files fixed to resolve the bug report r, and $\{r \mid f \in mod(r)\}$ is the set of bug reports such that they had required to fix f in the past. Intuitively, the bug report similarity is the source of the suspicious score and is equally distributed to the corresponding files.

[1]Although the original AmaLgam+ [7] did not square the similarity, this paper adopts the squared version because Rath et al. [10] reported its superiority.

C. Metric(3): Structured Information Retrieval-Based Metric

There have been studies that link a bug report to a file by focusing on their contents. BLUiR [17], one of the most promising methods, performs a structured information retrieval for bug localization. It quantifies the suspicious score of a file using the textual similarity between the components of the bug report and the file. We describe its computation procedure below. Suppose we have a new bug report r_{new} and a file f.

1) *Component extraction from bug report*: Extract "summary" part and "description" one from r_{new} separately. Let r_1 and r_2 be the summary part and the description one of r_{new}, respectively.
2) *Component extraction from source file*: Pick out the class names, the method names, the variable names, and the comments appearing in f. Let f_1, f_2, f_3, and f_4 be these components of the file, respectively.
3) *Vectorization of components*: Vectorize each of the above six components (r_1, r_2, f_1, f_2, f_3, and f_4) by the same way as the above metric described in Sect. II-B.
4) *Suspicious score computation*: Compute the cosine similarity for each pair of a bug report's component and a file's one, and obtain the suspicious score of f:

$$score_3(f) = \sum_{i=1}^{2} \sum_{j=1}^{4} sim(r_i, f_j) \ . \tag{3}$$

D. Metric(4): Stack Trace-Based Metric

Schroter et al. [18] reported an empirical study that proved the worth of stack trace in locating the corresponding bugs. Therefore, when a bug report includes the stack trace, it is beneficial to pay attention to it. We describe the metric for quantifying the suspiciousness of a file below. Although the following description assumes the Java environment, we can define similar metrics for other languages without losing generality if their stack traces provide the file names.

1) *Extraction of file names from bug report*: Suppose the new bug report r_{new} includes a stack trace. Extract all file names appearing in it.
2) *Ranking of source files*: Rank the extracted files in descending order of appearance. The more times the file appears in the stack trace, the higher rank it gets.
3) *Suspicious score computation*: Compute the suspicious score of a file f as:

$$score_4(f) = \begin{cases} \dfrac{1}{rank(f, r_{new})}, & (f \text{ appears in } r_{new}), \\ 0, & (\text{otherwise}), \end{cases} \tag{4}$$

where $rank(f, r_{new})$ is the rank of file f, obtained in the above process.

E. Metric(5): Bug Reporter-Based Metric

A user sometimes pays attention to certain functionality. When the same user reported two or more bug reports, these bugs may be related to each other. They may be caused by the same file or the ones in the same package. We can compute the suspicious score of a file using the above tendency as follows.

1) *Search of related bug reports*: Suppose we have a new bug report r_{new} that is submitted by a user. Search all already-resolved bug reports that were also submitted by the same user; Let R_{su} be the set of these bug reports.
2) *Detection of related packages*: For each related bug report $r \in R_{su}$, pick all files that were fixed to resolve r, and let $P_a(r)$ be the set of the packages to which these files belong. Then, let P_{all} be the set of the related packages: $P_{all} = \bigcup_{r \in R_{su}} P_a(r)$.
3) *Suspicious score computation*: Compute the suspicious score of a file f as follows.

$$score_5(f) = \begin{cases} 1, & (f \text{ is in } P_{all}), \\ 0, & (\text{otherwise}) \ . \end{cases} \tag{5}$$

F. Metrics Integration in AmaLgam+

AmaLgam+ integrates the above five metrics to evaluate the suspiciousness of a file f. It sorts all files in descending order of the integrated score and suggests the highly-ranked files as the ones most likely buggy. The integrated suspicious score of a file f, $susp(f)$, is defined as follows [7].

$$susp(f) = \begin{cases} \sum_{i=1}^{5} w_i \cdot score_i(f), & (score_2(f) > 0 \\ & \quad \vee score_3(f) > 0) \\ 0, & (\text{otherwise}) \ , \end{cases} \tag{6}$$

where w_i is the weight coefficient representing the degree to which $score_i$ contributes to f's combined suspiciousness (for $i = 1, \ldots, 5$); These weights have to be tuned well.

AmaLgam+ uses the genetic algorithm (GA) [19] to tune the above weights w_1–w_5. GA is one of the most useful algorithms to decide parameters to maximize the objective function. We can perform it using the GA package[2] of R.

GA needs the objective function that evaluates bug localization's effectiveness to explore the optimal or suboptimal solution. AmaLgam+ considers the following two criteria to organize the objective function.

- Mean Average Precision (MAP): For a new bug report, suppose we have n files ranked by the bug localization method, and m files are truly buggy. Let $\mathrm{P}(k)$ be the precision at the top k files, i.e., $\mathrm{P}(k)$ = (number of buggy files ranked in top k)/k. The average precision, AP, is defined as follows.

$$\mathrm{AP} = \frac{1}{m} \sum_{k=1}^{n} \mathrm{P}(k) \cdot \mathrm{I}(k) \ ,$$

  where $\mathrm{I}(k) = 1$ when the file ranked at k is buggy; otherwise, $\mathrm{I}(k) = 0$.

  Notice that different bug reports may have different AP values. The mean AP (MAP) is obtained as:

$$\mathrm{MAP} = \frac{1}{N} \sum_{i=1}^{N} \mathrm{AP}_i \ , \tag{7}$$

  where N is the number of bug reports, and AP_i is AP value of the i-th bug report (for $i = 1, \ldots, N$).
- Mean Reciprocal Rank (MRR): MRR focuses on the highest rank among the buggy files and uses the reciprocal rank (MR) as an evaluation value. The higher rank a buggy file places, the higher the evaluation value is. The mean MR (MRR) is obtained as follows.

$$\mathrm{MRR} = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{rank_i} \ , \tag{8}$$

  where $rank_i$ is the highest rank of buggy file for the i-th bug report (for $i = 1, \ldots, N$).

Then, AmaLgam+ uses the following objective function:

$$ObjFunction = e^{\mathrm{MAP+MRR}} \ . \tag{9}$$

III. Proposed Method

AmaLgam+ uses the five metrics and integrates them using GA. Although AmaLgam+ performs well, it has a challenge regarding the computational cost. When we turned the weight coefficients of the metrics using an average personal computer, it took over three hours in the worst case. Such a long time motivated us to propose another way of integrating metrics.

All of the above five metrics evaluate the suspicious degree to which a file is buggy. Moreover, the higher value each of the metrics has, the more suspicious the file is. Now we express a file as the five-dimensional vector whose elements are the corresponding metric values, $\boldsymbol{x} = (score_1, \ldots, score_5)^T$, and consider its distance from the origin in the vector space. Intuitively, the farther file is more suspicious. That is, the distance can be yet another integrated suspicious score.

Although Euclidean distance is a common distance metric, it is not reasonable in our scenario because it does not consider the data dispersion and the correlations. Thus, we propose to use Mahalanobis distance [13] in this paper. To give an intuitive interpretation of it, let us consider a scholar data x whose mean value is 0, and the standard deviation is σ. Mahalanobis distance between x and 0 is $\sqrt{(x-0)^2}/\sigma$. That is the distance normalized by the data dispersion. By generalizing this notion to multi-dimensional vectors, Mahalanobis distance between $\boldsymbol{x}$ and $\mathbf{0}$, $d(\boldsymbol{x})$, is computed as:

$$d(\boldsymbol{x}) = \sqrt{\boldsymbol{x}^T S^{-1} \boldsymbol{x}} \ , \tag{10}$$

where S is the variance-covariance matrix, and S^{-1} is the inverse matrix of S.

The computation of Mahalanobis distance is just a matrix calculation[3] and does not require any parameter tuning. Hence, we can perform it much faster than GA. Suppose the proposed

[2]https://cran.r-project.org/web/packages/GA/

[3]We can perform it using stats package of R.

(Mahalanobis distance-based) method can perform at almost the same level of accuracy as GA-based AmaLgam+. In that case, it can play a lightweight alternative to AmaLgam+ and be a solution to its challenge regarding the computational cost.

IV. Empirical Study

In this section, we report the empirical study to examine our proposal. We describe the aim and the dataset in Sect. IV-A and explain the procedure in Sect. IV-B. Then, we present the results and our discussions in Sect. IV-C. Finally, we describe the threats to validity in Sect. IV-D.

A. Aim and Dataset

As we mentioned above, AmaLgam+ has a challenge regarding the computational cost, and we have proposed the Mahalanobis distance-based method to overcome the problem. In this study, we perform the bug localization by AmaLgam+ and the proposed method, comparing the results in terms of (1) accuracy and (2) computation time. If the proposed method can perform much faster than AmaLgam+ while having the bug localization accuracy at almost the same level as AmaLgam+, the proposed method can be a useful alternative method.

We use a part of the datasets provided by Rath and Patrick[4]: the bug reports (labeled as "bug" or "improvement") submitted to six open source software projects shown in Table I. Although the datasets do not include the source files, we obtained them from their repositories available at GitHub.

We carried out all of our data processing and computations on the personal computer whose CPU is Intel Core i5-9400, memory size is 32GB, and OS is Windows 10.

B. Procedure

We performed the following procedure for each project.

1) *Metrics computation*: For each bug report and source file, we compute the values of the above five metrics.
2) *Suspicious score computation*: We combine the metric values to get the integrated suspicious score for each file by AmaLgam+ and the proposed method independently. We also measure the elapsed time to finish all computations. In AmaLgam+, to tune the weight coefficients, we randomly sampled 5% of the bug reports and run GA with the parameters[5] `maxiter=200`, `popsize=50`, and `run=100`. We include the parameter tuning time in the computation time mentioned above.
3) *Accuracy evaluation*: For each bug report, we evaluate the bug localization accuracy by AmaLgam+ and the proposed method, respectively. We adopt Top@k measures [20] and use Top1, Top5, and Top10 accuracy.

TABLE I
Datasets used in this study

Project	Number of reports	KLOC	Data collection period
Railo	529	241	2008-11 — 2013-12
Izpack	402	89	2009-01 — 2016-01
Log4j2	676	72	2008-12 — 2017-04
Weld	657	63	2009-01 — 2017-03
Hornetq	481	171	2006-05 — 2015-06
Seam2	784	106	2005-08 — 2014-03

[4]https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/PDDZ4Q

[5]The values of `maxiter` and `popsize` are the same as the ones used in [7]; We set the value of `run` based on the tutorial of the package.

C. Results and Discussions

Table II shows the bug localization accuracy. Although the proposed method does not outperform AmaLgam+, it shows almost the same accuracy: While AmaLgam+ outperformed the proposed method in 12 cases (e.g., Top1 in Log4j2), the proposed method did in 5 cases (e.g., Top5 in Ralio); the remaining one case (Top10 in Hornetq) was a tie.

Table III presents the comparison of the computation times between AmaLgam+ and the proposed method. While AmaLgam+ took $12,782$ seconds (about 3.55 hours) in the worst case (Ralio project), the proposed finished within 246 seconds (about 4 minutes). That is, the proposed method reduced the computation time by about 98%. The average rate of time reduction is about 97%. Hence, the proposed method successfully performs the bug localization much faster than AmaLgam+ while not seriously losing the accuracy.

AmaLgam+ tunes the weight coefficients of the five suspicious metric values to maximize the predefined objective function. On the other hand, the proposed method simply computes the distance from the origin as the combined suspicious score while considering the metric data dispersion and the correlations between metrics. In this sense, GA-based AmaLgam+ is a supervised learning method, and the Mahalanobis distance-based method is an unsupervised learning one. Because the proposed method does not perform any parameter tuning, it has a risk of producing a useless integrated score if one of the metrics does not work well for locating bugs. In other words, GA can minimize such a risk by giving a smaller weight to the useless metric through tuning. Such a difference might

TABLE II
Top1, Top5, and Top10 accuracy (%)

Project	AmaLgam+			Proposed method		
	Top1	Top5	Top10	Top1	Top5	Top10
Railo	17.61	35.98	45.45	16.48	39.39	48.67
Izpack	29.03	52.85	61.54	24.07	48.64	59.31
Log4j2	30.18	56.95	65.38	27.37	58.28	67.90
Weld	16.44	35.46	44.44	16.74	31.51	40.79
Hornetq	18.26	33.82	36.93	15.15	31.95	36.93
Seam2	12.75	26.10	32.84	10.54	22.06	27.21

TABLE III
Comparison of computing times

Project	elapsed time (sec)		$\frac{(a)-(b)}{(a)}$
	(a) AmaLgam+	(b) proposed method	
Railo	$12,782$	246	98.08%
Izpack	$1,090$	24	97.80%
Log4j2	$2,709$	68	97.49%
Weld	$1,024$	31	96.97%
Hornetq	920	36	96.09%
Seam2	$1,724$	109	93.68%

cause the loss of bug localization accuracy shown in Table II. Nonetheless, the empirical results did not show severe deterioration in accuracy. That is, it successfully maintains almost the same accuracy and reduces the computation time dramatically (reduced by about 97% on average).

We used the notion of Mahalanobis distance to integrate different metrics for bug localization reasonably and overcomes AmaLgam+'s challenge regarding computational cost. However, our method is not only for AmaLgam+; i.e., it can be applied to other bug localization methods. There have been various bug localization methods using IR, spectrum analysis, or program analysis techniques [4], [21]–[27]. Because many methods utilize two or more metrics for bug localization, we can apply our integration method to those bug localization methods without any changes. A further comparative study is our significant future work.

D. Threats to Validity

Construct validity: The parameter setting of GA would affect the computational results. If we run the GA computation with a different parameter, we may obtain a different result. Although we used the same parameter as the previous work [7], it would be better to tune the settings as well.

External validity: Because we used only six Java projects, the proposed method may not work well for another project. To mitigate this threat, we would also need to perform a further study using various projects written in other than Java.

V. Conclusion and Future Work

AmaLgam+ is a promising bug localization method, which combines five metrics into a suspicious score of a source file. However, because AmaLgam+ uses GA to decide the weight coefficients of metrics, it tends to need a long time for tuning the weights. To reasonably integrate these metrics with a low cost, we proposed a Mahalanobis distance-based method. Then, we empirically proved that the proposed method could reduce the computation time by about 97% on average without severe loss of the bug localization accuracy. Thus, it would be a lightweight alternative to AmaLgam+.

Our future work is to enhance the accuracy of the proposed method, i.e., narrow the accuracy gap with AmaLgam+. We plan to further study the contribution of each metric and incorporate the study results into the computation of Mahalanobis distance toward a better and fast bug localization. A further comparative study using various methods other than AmaLgam+ is also our significant future work.

Acknowledgment

This work was supported by JSPS KAKENHI #18K11246 and #20H04184.

References

[1] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, vol. 1, May 2010, pp. 45–54.

[2] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful ... really?" in *In Proc. 24th IEEE Int. Conf. Softw. Maintenance*, Oct. 2008, pp. 337–345.

[3] W. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, pp. 707–740, Aug. 2016.

[4] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *Proc. 34th Int. Conf. Softw. Eng.*, June 2012, pp. 14–24.

[5] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proc. 22nd Int. Conf. Program Comprehension*, May 2014, pp. 53–63.

[6] Q. Wang, C. Parnin, and A. Orso, "Evaluating the usefulness of ir-based fault localization techniques," in *Proc. Int. Symp. Softw. Testing & Analysis*, July 2015, pp. 1–11.

[7] S. Wang and D. Lo, "Amalgam+: Composing rich information sources for accurate bug localization," *J. Softw.: Evol. & Process*, vol. 28, no. 10, pp. 921–942, Oct. 2016.

[8] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Inf. & Softw. Tech.*, vol. 52, no. 9, pp. 972–990, Sept. 2010.

[9] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Softw. Eng.*, Nov. 2014, pp. 689–699.

[10] M. Rath, D. Lo, and P. Mader, "Analyzing requirements and traceability information to improve bug localization," in *Proc. 15th IEEE/ACM Working Conf. Mining Softw. Rep.*, May 2018, pp. 442–453.

[11] M. Rath and P. Mader, "Influence of structured information in bug report descriptions on ir-based bug localization," in *Proc. 44th Euromicro Conf. Softw. Eng. & Advanced App.*, Aug. 2018, pp. 26–32.

[12] M. Rath and P. Mader, "Structured information in bug report descriptions—influence on IR-based bug localization and developers," *Softw. Quality J.*, vol. 27, pp. 1315–1337, May 2019.

[13] G. Taguchi, S. Chowdhury, and Y. Wu, *The Mahalanobis-Taguchi System*. New York: McGraw-Hill, 2001.

[14] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *Proc. 29th Int. Conf. Softw. Eng.*, May 2007, pp. 489–498.

[15] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, "Bugcache for inspections : Hit or miss?" in *Proc. 19th ACM SIFSOFT Symp. & 13th European Conf. Foundations Softw. Eng.*, Sept. 2011, pp. 322–331.

[16] C. Lewis and R. Ou, "Bug prediction at google," http://google-engtools.blogspot.com/2011/12/bug-prediction-at-google.html.

[17] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2013, pp. 345–355.

[18] A. Schroter, A. Schroter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *Proc. 7th IEEE Working Conf. Mining Softw. Rep.*, May 2010, pp. 118–121.

[19] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992.

[20] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge: Cambridge University Press, 2008.

[21] A. Rui, Z. Peter, G. Rob, and G. Arjan, "A practical evaluation of spectrum-based fault localization," *Softw. Quality J.*, vol. 82, pp. 1780–1792, Nov. 2009

[22] A. Rui, Z. Peter, G. Arjan, and J.C.van, "Spectrum-Based Multiple Fault Localization," in *Proc. 24th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2009, pp. 88–99.

[23] X. Xie, W.E. Wong, T.Y. Chen and B. Xu, "Spectrum-Based Fault Localization: Testing Oracles are No Longer Mandatory," in *Proc. 11th Int. Conf. Quality Softw.*, July 2011, pp. 1–10.

[24] L. Yun, S. Jun, X. Yinxing, L. Yang and D. Jinsong, "Feedback-Based Debugging," in *Proc. 39th Int. Conf. Softw. Eng.*, May 2017, pp. 393–403.

[25] L. Yun, S. Jun, T. Lyly, B. Guangdong, W. Haijun and D. Jinsong, "Break the Dead End of Dynamic Slicing: Localizing Data and Control Omission Bug," in *Proc. 33rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sept. 2018, pp. 509–519.

[26] W. Tao and R. Abhik, "Hierarchical Dynamic Slicing," in *Proc. Int. Symp. Softw. Testing & Analysis*, July 2007, pp. 228–238.

[27] M. Wen, R. Wu and S. Cheung, "Locus: Locating bugs from software changes," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sept. 2016, pp. 262–273.