2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing

Using Co-Change Histories to Improve Bug Localization Performance

Chakkrit Tantithamthavorn, Akinori Ihara, Ken-ichi Matsumoto
Software Engineering Laboratory
Graduate School of Information Science,
Nara Institute of Science and Technology, Japan.
E-mail: {chakkrit-t,akinori-i,matumoto}@is.naist.jp

***Abstract*—A large open source software (OSS) project receives many bug reports on a daily basis. Bug localization techniques automatically pinpoint source code fragments that are relevant to a bug report, thus enabling faster correction. Even though many bug localization methods have been introduced, their performance is still not efficient. In this research, we improved on existing bug localization methods by taking into account co-change histories. We conducted experiments on two OSS datasets, the Eclipse SWT 3.1 project and the Android ZXing project. We validated our approach by evaluating effectiveness compared to the state-of-the-art approach BugLocator. In the Eclipse SWT 3.1 project, our approach reliably identified source code that should be fixed for a bug in 72.46% of the total bugs, while BugLocator identified only 51.02%. In the Android ZXing project, our approach identified 85.71%, while BugLocator identified 60%.**

***Index Terms*—Software Maintenance; Co-Change Histories; Bug Localization; Information Retrieval;**

I. Introduction

In a large software project, software maintenance is a key activity focusing on the modification of the software product after release to correct bugs and to improve performance. On a daily basis, software projects receive many bug reports. Due to the increasing size and complexity of current software applications, finding a buggy file is a painstaking and time-consuming activity for developers. To address this problem, many automated software debugging systems based on static and dynamic program analyses have been developed to reduce human effort and software maintenance cost.

Bug localization is one of the popular automated software debugging approaches. It aims to automatically pinpoint which code fragments are relevant to a bug report allowing faster correction. Recently, Information Retrieval (IR) based techniques have been widely used to localize a bug, such as Latent Dirichlet Allocation (LDA) [1][2], Latent Semantic Indexing (LSI) [1]-[3], Vector Space Model (VSM) [2],[4]-[5], Cluster Based Document Model (CBDM) [2], and the Unigram Model (UM) [2]. Among these, Rao and Kak [2] reported that VSM is the most effective method. Currently, Zhou et al. [5] proposed BugLocator which improved on traditional VSM. Not only can this method effectively retrieve relevant buggy files given a bug report query, but it also utilizes information about similar bugs that have been fixed before to improve the ranking performance. Therefore, BugLocator is currently the best available bug localization method. However, the authors reported that the accuracy of BugLocator relies on the quality of the bug report. If a bug report does not provide enough information, or misleading information, the performance of BugLocator is adversely affected.

To improve on the performance of the existing bug localization techniques, we leveraged the following assumption: *if a buggy file was fixed, then the files that were changed together should be fixed together.* In this research, we introduce a novel bug localization method which not only considers the textual features in the same way as existing methods do, but also relies on the *co-change histories* that identify files which have been changed together before.

Our method consists of three steps. First, we calculate the co-change score by constructing a co-change matrix. Second, we create a list of all possible co-change files using the co-change score. Third, as a target method, we augment the results obtained from BugLocator as proposed by Zhou et al. [5]. These two techniques are complementary because BugLocator identifies buggy files based on textual features while co-change identifies buggy files based on what set of files are commonly changed together. We evaluate our approach on two OSS datasets, the Eclipse SWT 3.1 project and the Android ZXing project. In comparison with previous research, our work is different and contributes in these two ways:

- We introduce a novel bug localization method that takes advantage of the co-change assumption by introducing a co-change score, which is used to adjust the results obtained from BugLocator to increase performance.
- We introduce an in-depth evaluation approach to measure the effectiveness of our approach in comparison to previous research.

The organization of the paper is as follows. In Section II, we describe the background of this work, in terms of co-change histories, bug localization and the architecture of BugLocator. In Section III, we describe our proposed approach, augmenting the results of BugLocator with information from co-change histories. Section IV describes our experimental design, and Section V shows and discusses the experimental results and contributions. Section VI gives the threats to validity. We conclude the paper and underline future works in Section VII.

978-0-7695-5005-3/13 $26.00 © 2013 IEEE
DOI 10.1109/SNPD.2013.92

II. Background

A. Co-Change Histories

A co-change event or change propagation consists of all classes whose changes have been committed at exactly the same time by exactly the same author. This concept was first introduced by Ball et al [6]. They used co-change information to visualize a graph of co-changed classes. Then they found clusters of classes that often changed together during the evolution of the system. This co-change information either can be present in the versioning system, or must be inferred by analysis. For example, Subversion marks co-changing files at commit time as belonging to the same *change set*, while in CVS, the logically coupled files must be inferred from the modification time of each file. Gal et al. [7] showed that the concept helps to derive useful insights about the system architecture. Other work has investigated the causes of change propagation [8]-[9], co-change prediction [10]-[11] and co-change visualization [12]-[13].

In terms of these different studies, this paper is positioned as follows: We propose a novel approach using co-change histories to improve existing bug localization techniques. We use the same co-change definition from previous studies as the event of all classes or files being changed at the same time by the same author [10][14]. We introduce a method to calculate a co-change score to construct a co-change matrix, then we adjust the results obtained from BugLocator by combining the results from our method.

B. Bug Localization based on Information Retrieval

Software bug localization is one of the most painstaking and time-consuming activities in program debugging. To overcome this problem, there is a high demand for automatic bug localization techniques that can guide programmers to the locations of bugs based on an initial bug report. Recently, Information Retrieval (IR) based techniques have been widely used to localize a bug, such as Latent Dirichlet Allocation (LDA) [1][2], Latent Semantic Indexing (LSI) [1]-[3], Vector Space Model (VSM) [2],[4]-[5], Cluster Based Document Model (CBDM) [2], and the Unigram Model (UM) [2]. Among these, Rao and Kak [2] reported that VSM was the most effective method. A common bug localization process which consists of the following three steps:

[Step 1: Corpus creation] This step extracts semantic words from source code and bug reports. Multiple-word identifiers are separated into single words. For example, *GetInitialValue()* will be split into three words: *Get*, *Initial*, and *Value*. Each remaining word will be normalized to lower case and be stemmed by the Porter Stemmer algorithm[1] to determine the root meaning. After that, some programming language keywords (e.g., int, double, char, etc), separators, operators, and common English words (e.g., a, an, the, etc) will be removed to reduce noise and retain the original meaning.

[Step 2: Indexing] The forward index stores a list of words for each document. Building an inverted index can quickly locate documents containing the words in a query and then rank these documents by relevance. Therefore, bug localization can directly access the index to find the documents associated with each word in the query and quickly retrieve the matching documents.

[Step 3: Retrieval & ranking] Bug localization treats the source code files as a document corpus and let the bug report as a query. Then, it calculates the relevant score between a document vector and a query vector by using various approaches such as the Vector Space Model.

[1]http://tartarus.org/martin/PorterStemmer/

C. The Architecture of BugLocator

In this research, as a target system, we extend BugLocator [5] to evaluate how using information about co-change histories improves existing bug localization methods. Figure 1 shows the architecture of BugLocator. It was introduced as an approach to ranking buggy files based on the similarity of source code files and the similarity of past bug reports.

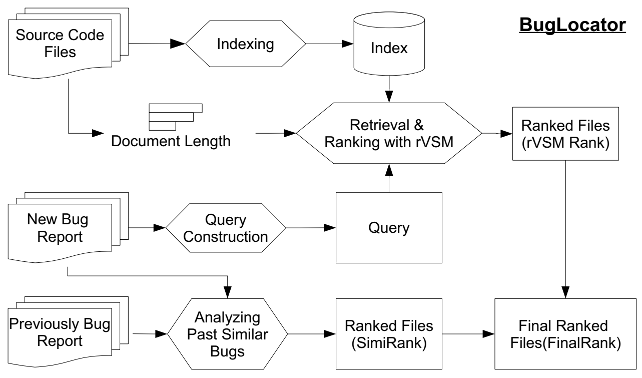

Fig. 1. The architecture of BugLocator [5]

1) Ranking Based on Source Code Files: BugLocator treat source code files as a document corpus, and a bug report as a query. Then, it calculates the relevancy score between a document vector and a query vector by using cosine similarity. To improve the performance of the classic VSM, they determine the term-frequency (tf) and inverse document frequency (idf) Equation as shown in Equation (1).

$$tf(t,d) = log(f_td) + 1, idf(t) = log\frac{\#docs}{n_t} \tag{1}$$

The larger the source code files, the higher the probability of containing a bug. They also use a logistic regression function in Equation (2) to give a higher score to larger documents during ranking where $N(x)$ is a normalization of the document length.

$$g(\#term) = \frac{1}{1+e^{-N(x)}}, N(x) = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{2}$$

In Equation (3), they introduced the $rVSMScore$ which is weighted by the document length score and optimized by the logarithm variant of the tf from Equation (1).

$$rVSMScore(q,d) = g(\#term) \times \frac{\vec{V}_q \bullet \vec{V}_d}{|\vec{V}_q||\vec{V}_d|} \quad (3)$$

2) Ranking Based on Similar Bugs: Past similar bug reports are analyzed under the hypothesis that similar bugs tend to require fixes to similar files. This similarity is computed by Equation (4).

$$SimiScore = \sum_{\forall S_i connect\_to F_j} (Similarity(B, S_i)/n_i) \quad (4)$$

3) Combining Score: The final score is calculated as a relevance score between a bug report to a relevant source code by combining the score between rVSMRank and SimiRank together as shown in Equation (5).

$$FinalScore = (1-\alpha) \times rVSMScore + \alpha \times SimiScore \quad (5)$$

, where α is a weighting factor between $0 \leq \alpha \leq 1$. We use $\alpha = 0.2$ for all experiments in this research. However, one of the limitations of BugLocator is that the accuracy relies on the quality of the bug report. If a bug report does not provide enough information, or provides misleading information, the performance of BugLocator is adversely affected.

III. The Proposed Approach

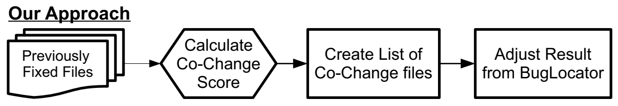
Our Approach
Previously Fixed Files
Calculate Co-Change Score
Create List of Co-Change files
Adjust Result from BugLocator

Fig. 2. The overall architecture of the proposed approach

Figure 2 shows the overall architecture of the proposed approach. First, we calculate the co-change score by constructing a co-change matrix. Then, we create a list of all possible co-change files using the co-change score. Finally, we adjust the results from BugLocator. The details of our approach are described below.

A. Analysis of Co-Change Histories

We construct a co-change matrix to calculate the co-change score. When N denotes the total number of classes, we define a diagonal co-change matrix C which has dimension $N \times N$ where $C_{i,j}$ is the number of times that each element has been modified concurrently with other elements [14]. We do not consider the $C_{i,j}$ value where $i = j$. To illustrate this, an entry $C_{i,j} = 5$ tells us that classes i and j have been modified 5 times together.

We define the $CoChangeScore$ by performing a normalization technique to scale the attribute data to fall within an appropriate range of 0 to 1 as shown in Equation (6) where C_{max} and C_{min} are the maximum and minimum value of vector $\vec{C_i}$ respectively. The higher the $CoChangeScore$ value, the stronger the relationship becomes.

$$CoChangeScore(C_{i,j}) = \frac{C_{i,j} - C_{min}}{C_{max} - C_{min}} \quad (6)$$

Then, we create a list of all possible co-change files related to the relevant result from BugLocator using the co-change score. We define $CoChangeSets(Bug_n)$ as a set of all possible co-change files related to the relevant result from BugLocator where Bug_n is a set of Top-N relevant source code files for each bug report given from the BugLocator result. We define F as the set of all source codes in a repository.

$$\begin{aligned} & CoChangeSets(Bug_n) \quad (7) \\ = & \bigcup_{\forall b \in B_n, \forall f \in F} \{f | CoChangeScore(C_{b,f}) > \delta\} \\ & , where; b \neq f \end{aligned}$$

The intuitive meaning behind Equation (7) is that $CoChangeSets(Bug_n)$ provides a union set of all source code files f that are a member of F such that $CoChangeScore$ between file b and file f has a score higher than a threshold where b is a member of the set of Top-N relevant source code files obtained from the BugLocator result and δ is a threshold of $CoChangeScore$. In this research, we use $\delta = 0.85$ for all experiments.

To illustrate, given a list of all source code files $F = \{1, 2, 3, 4, 5\}$ where the list of buggy files in Top-1 obtained from BugLocator is $B = \{1\}$.

$$C_{m,n} = \begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{pmatrix} 0 & 0.89 & 0.95 & 0.01 & 0.33 \\ 0.89 & 0 & 0.84 & 0.93 & 0.91 \\ 0.95 & 0.84 & 0 & 0.82 & 0.51 \\ 0.01 & 0.93 & 0.82 & 0 & 0.29 \\ 0.33 & 0.91 & 0.51 & 0.29 & 0 \end{pmatrix} \end{array} \quad (8)$$

We show an example of the co-change matrix after performing the normalization technique in Equation (8). Therefore, the final co-change files $CoChangeSets(B)$ is $\{2, 3\}$.

B. Augmented result

We adjust the result obtained from BugLocator by combining the results between the set $CoChangeSets(Bug_n)$ and the set B_n together as shown in the following Equation (9) because we did not consider the $CoChangeScore(i, j)$ where $i = j$ as mentioned above.

$$PredictedBuggyFile = CoChangeSets(B_n) \cup B_n \quad (9)$$

Finally, we have a new relevant source code files which is related to a bug report. In this example, we now consider $PredictedBuggyFile = \{1, 2, 3\}$

IV. Experimental Setup

To evaluate the effectiveness of our approach, we conducted a series of experiments with two OSS projects. First, we used the Eclipse SWT 3.1 project (an open source widget toolkit for Java) which contained 98 bug reports. Second, we used the Android ZXing project (a barcode image processing library for Android application) which contained 20 bug reports. These two datasets were obtained from BugLocator's dataset[2] provided by Zhou et al. and are not the complete sets of bug reports from these projects. We performed experiments on the following research questions to validate our approach.

RQ1: Does our model improve the existing bug localization performance?

We performed experiments on these two datasets. We separated the bug reports into two chunks ordered by time. The first chunk is 30% of the bug reports used for a training set, while the other 70% is used to evaluate the system. We used the first chunk to build co-change histories. We measured the effectiveness of our approach by using the performance metrics shown in Equation (10). This metric measures the percentage of the number of successfully localized bug reports. To measure this metric, we checked the ranks of predicted buggy files by our approach in testing set. If the files are ranked in the Top-1, Top-5 or Top-10 of actual buggy files, we considered that the report was effectively localized. To answer this question, we compared our results with traditional VSM and BugLocator to measure the effectiveness of our approach.

$$Performance(\%) = \frac{\#SuccessfullyLocalizedBugs}{\#TotalBugs} \tag{10}$$

Furthermore, we also studied the impact of co-change histories on bug localization with various sizes of dataset. We measured the performance with the Eclipse SWT 3.1 data set using various sizes of training and testing data sets to validate the impact of co-change histories. We examined the ratio of training data sets from 0.1 to 0.9. For example, ratio 0.3 is the combination with the first 30% of the bug reports for training and the remaining 70% of the bug reports for testing.

RQ2: How many buggy files does this model cover?

There is much research measuring the performance of bug localization method using Equation (10). In this research, if at least one actual buggy file was correctly predicted, the bug is considered successfully localized. In our empirical analysis, we found that there are often many buggy files that were fixed based on one bug report. In this research, we performed an in-depth analysis to validate the ability to predict buggy files by using the coverage ratio. We calculated the $CoverageRatio$ in each bug report using Equation (11). This metric measures the percentage of the number of buggy files that this model covers.

[2]http://code.google.com/p/bugcenter/wiki/BugLocator

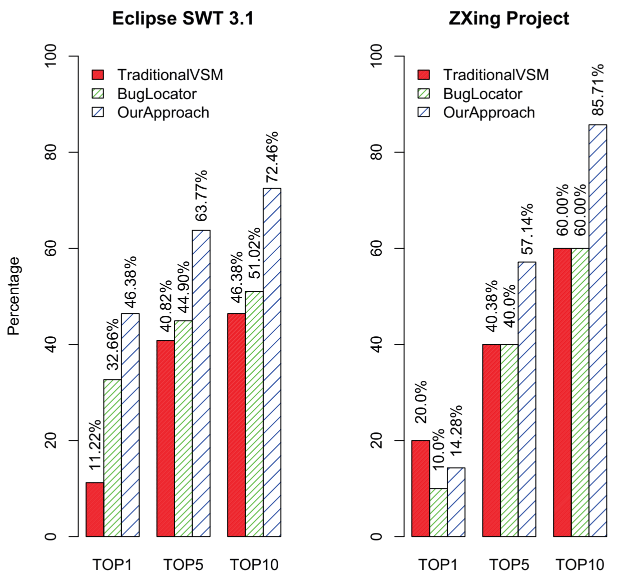

Fig. 3. The performance of our approach comparing to traditional VSM and BugLocator

$$CoverageRatio_(\%) = \frac{\#SuccessfullyPredictedBuggyFiles}{\#ActualBuggyFiles} \tag{11}$$

To answer this question, we measured the effectiveness of our approach using the $AverageCoverageRatio$ calculated by Equation (12). It is the average of the $CoverageRatio$ of all bug reports.

$$AverageCoverageRatio = \frac{1}{M} \times \sum_{i=1}^{M} CoverageRatio_i \tag{12}$$

V. Results and Contributions

RQ1: Does our model improve existing bug localization performance?

Figure 3 shows the performance of our approach on the two projects. In the Eclipse SWT 3.1 project, our approach successfully identified 46.38%, 63.77%, 72.46% of the bug reports in Top-1, Top-5, and Top-10 respectively. For the Android ZXing project, our approach successfully identified 14.28%, 57.14%, 85.71% of bug reports in Top-1, Top-5, and Top-10 respectively. For comparison, we used the same testing data set for all subject systems. We also compared our results to traditional VSM and BugLocator. The results show that our approach outperforms traditional VSM and BugLocator. We can conclude that our approach using co-change histories can localize a large percentage of bugs.

To measure the impact of co-change histories in bug localization, we also calculated the performance of our method

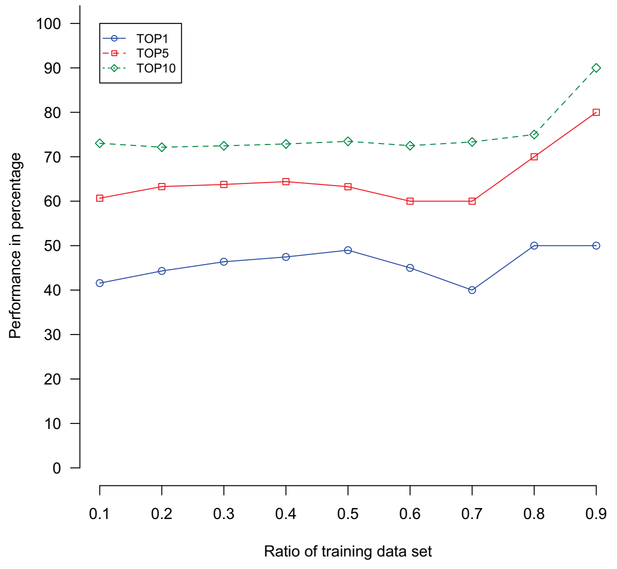

Fig. 4. The performance of Eclipse SWT 3.1 in different size of training data set

with the Eclipse SWT 3.1 project data set using different sizes of training data, as shown in Figure 4. The x axis is the ratio of training data set size and the y axis is the percentage of performance calculated by Equation (10). The result shows that the larger the training data set, the higher the accuracy of our approach is. However, because of the results of our approach are mainly based on the results obtained from BugLocator, the performance at the ratio 0.7 in Top-1 and Top-5 decreased slightly. If the result obtained from BugLocator is not a buggy file, the performance of our approach may decrease slightly.

RQ2: How many buggy files does this model cover?

We conducted an experiment to measure the effectiveness of our approach using the $AverageCoverageRatio$ described in the previous section. Table I shows the effectiveness on the two projects. The results show that our model can predict up to 58.33% of the actual buggy files in the Eclipse SWT 3.1 and 70.71% of the actual buggy files in the Android ZXing project. As a result, we can conclude that our approach using co-change histories is completely effective to localize a bug based on the initial bug report.

Dataset	**TOP1**	**TOP5**	**TOP10**
Eclipse SWT 3.1	34.07 %	48.65 %	58.33 %
Zxing	10.71 %	44.28 %	70.71 %

TABLE I
THE EFFECTIVENESS OF OUR APPROACH

VI. Threats to Validity

This section discusses potential threats to the validity of our experiments as following:

- The first threat to validity in this experiment is related to the data collection which was based on the BugLocator datasets. Even though BugLocator is the state-of-the-art algorithm, they did not use the completed set of bug reports in the Bug Tracking System.
- Second, we only analyzed two Java programs, so our results may not be generalizable to projects with other programs written in different languages. Also, the results of our approach may be different from results with other projects because our experiments were based on open source software projects.
- Third, the results of our approach are mainly based on the results obtained from BugLocator. When the result obtained from BugLocator is not a buggy file, the performance of our approach may decrease slightly.

VII. Conclusions and Future Work

In this research, we introduced a novel bug localization method which not only considered the textual features as existing approaches do, but also utilzed on the co-change histories, identifying class files that have been changed at the same time. Our approach consisted of three steps. First, we calculated the co-change score by constructing a co-change matrix. Second, using the co-change score, we created a list of all possible co-change files using the co-change score. Third, as a target method, we augmented the results obtained from BugLocator as proposed by Zhou et al. [5]. These two techniques are complementary because BugLocator identifies potential buggy files based on textual features while co-change identifies them in terms of sets of files are commonly changed together at the same time.

We based our experimental evaluation of our approach on two OSS datasets from the Eclipse SWT 3.1 project and the Android ZXing project. As described in Section V, in comparison to the state-of-the-art BugLocator approach, on the Eclipse SWT 3.1 project data, our approach reliably identified 72.46% of the total bugs, while BugLocator identified only 51.02%. Similarly, on the Android ZXing project data, our approach identified 85.71% of the buggy files, while BugLocator identified 60.00%. From these results, we conclude that our approach using co-change histories improves the performance of existing bug localization approaches, localizing a larger percentage of the reported bugs. It is our belief that research in these directions can help significantly reduce the human efforts and software maintenance cost.

The future research directions for our work can be summarized as follows:

- We plan to extend our experiments to other projects, including a large evolutionary software project.
- We plan to conduct experiments by using other bug localization target systems.

Acknowledgement

We would like to thank Prof. Mike Barker from Nara Institute of Science and Technology for valuable comments from early versions of this paper. We also thank the anonymous reviewers for their valuable comments. This research is being conducted as a part of Grant-in-aid for Young Scientists (B), 25730045, 2013 of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[1] S. K. Lukins, N. a. Kraft, and L. H. Etzkorn, "Bug localization using latent Dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, Sep. 2010.

[2] A. K. Shivani Rao, "Retrieval from Software Libraries for Bug Localization : A Comparative Study of Generic and Composite Text Models," in *Proceedings of the 8th IEEE Working Conference on Mining Software Repositories (MSR'11)*, 2011, pp. 43–52.

[3] B. D. Nichols, "Augmented Bug Localization Using Past Bug Information," in *Proceedings of the 48th Annual Southeast Regional Conference (ACM SE'10)*, 2010, pp. 61:1–61:6.

[4] E. Hill, "On the Use of Stemming for Concern Location and Bug Localization in Java," in *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation (SCAM'12)*, 2012, pp. 184–193.

[5] J. Zhou, H. Zhang, and D. Lo, "Where Should the Bugs Be Fixed ?" in *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012, pp. 14–24.

[6] T. Ball, J.-m. K. A. A. Porter, and H. P. Siy, "If Your Version Control System Could Talk ..." in *Proceedings of the ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997.

[7] H. Gall, "Change analysis with evolizer and changedistiller," *IEEE Software*, vol. 26, no. 1, pp. 26–33, 2009.

[8] G. Antoniol, V. F. Rollo, G. Venturi, and E. P. D. Montr, "Detecting groups of co-changing files in CVS repositories," in *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, 2005, pp. 23–32.

[9] T. Zimmermann, P. Weiß gerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," in *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, 2004, pp. 563–572.

[10] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *Proceedings of the 20th International Conference on Software Maintenance (ICSM'04)*, 2004, pp. 284–293.

[11] N. Tsantalis, A. Chatzigeorgiou, and I. C. Society, "Predicting the Probability of Change in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 601–614, 2005.

[12] M. D. Ambros, S. Member, M. Lanza, and I. C. Society, "Visualizing Co-Change Information with the Evolution Radar," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 720–735, 2009.

[13] a. Vanya, R. Premraj, and H. van Vliet, "Interactive Exploration of Co-evolving Software Entities," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR'10)*, Mar. 2010, pp. 260–263.

[14] M. M. G. Schweitzer and Frank, "The Link between Dependency and Co-Change : Empirical Evidence," *IEEE Transactions on Software Engineering*, vol. 38, no. 0098-5598, pp. 1432–1444, 2011.

[15] S. Lal and A. Sureka, "A Static Technique for Fault Localization Using Character N-Gram Based Information Retrieval Model Categories and Subject Descriptors," in *Proceedings of the 5th India Software Engineering Conference (ISEC'12)*, 2012, pp. 109–118.

[16] M. Revelle, M. Gethers, and D. Poshyvanyk, "Using structural and textual information to capture feature coupling in object-oriented software," *Empirical Software Engineering*, vol. 16, no. 6, pp. 773–811, Mar. 2011.

[17] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An Empirical Study on Inconsistent Changes to Code Clones at Release Level," in *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE'09)*, 2009, pp. 85–94.

[18] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel, "Using concept analysis to detect co-change patterns," in *Proceedings of the 9th international workshop on Principles of software evolution in conjunction with the 6th ESEC/FSE joint meeting (IWPSE'07)*. ACM Press, 2007, pp. 83–89.

[19] M. P. Robillard, "Recommending change clusters to support software investigation : an empirical," *Journal of Software Maintenance and Evolution*, vol. 22, no. 3, pp. 143–164, 2010.

[20] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE'08)*, 2008, p. 308.

[21] M. M. Geipel and F. Schweitzer, "Software Change Dynamics : Evidence from 35 Java Projects," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE'09)*, 2009, pp. 269–272.

[22] M. D'Ambros, M. Lanza, and R. Robbes, "On the Relationship Between Change Coupling and Software Defects," in *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE'09)*, 2009, pp. 135–144.

[23] T. Zimmermann, S. Kim, A. Zeller, and E. J. W. Jr, "Mining Version Archives for Co-changed Lines," in *Proceedings of the 2006 international workshop on Mining software repositories (MSR'06)*, 2006, pp. 72–75.

[24] D. Beyer, "Co-Change Visualization," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 89–92.

[25] C. C. Williams, J. K. Hollingsworth, and S. Member, "Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 466–480, 2005.