Improving Spectrum Based Fault Localisation Techniques

Gulsher Laghari
AnSyMo
Universiteit Antwerpen België
gulsher.laghari@uantwerpen.be

Alessandro Murgia
AnSyMo
Universiteit Antwerpen België
alessandro.murgia@uantwerpen.be

Serge Demeyer
AnSyMo
Universiteit Antwerpen België
serge.demeyer@uantwerpen.be

***Abstract*—Spectrum based fault localisation techniques merely use the coverage of the program elements to localise the fault. These techniques ignore the dependency relationships between the elements, such as a combination of methods to be called together, and hence come at the cost of their diagnostic accuracy. In this paper we present a variation of spectrum based fault localisation techniques which leverages the sequences of method calls as coverage elements than merely individual method coverage. We compare our technique with traditional spectrum based fault localisation techniques, which use the coverage of individual methods, and demonstrate with a small set of faults that our technique outperforms these techniques.**

I. Introduction

During software maintenance, changes in the code may introduce regression faults, causing some of the previously passing test cases to fail [1]. Once such a regression fault appears, developers need to find the root cause of the fault in hundreds if not thousands of classes, a daunting task at best. To help developers locate the faults, there exist two broad categories of techniques (1) information retrieval based fault localisation techniques [2]–[5], and (2) spectrum based fault localisation techniques [6]–[9]. Both of these techniques produce a ranked list of program elements indicating their likelihood of containing the fault. While the former uses bug reports and source code files for analysis, the later uses program traces generated by executing failing and passing test cases. Since spectrum based fault localisation only require traces from test runs readily available after running the regression test suite— they are ideally suited for locating regression faults.

Spectrum based fault localisation [6]–[9], also known as *coverage based fault localisation* [10], is a lightweight automated technique to locate the faults. It localises the faults by comparing traces form failing and passing test cases. It discovers statistical correlation between test case failures and the execution of the program elements, also known as *units under test*. It collects the coverage information of units under test in the test cases as a test coverage matrix (TCM). A TCM is represented as a matrix, where rows correspond to units under tests and columns represent test cases [8], [9]. Each element in the matrix marks whether a given unit is covered (marked as 1) or not (marked as 0). A similarity coefficient [7] is used to assign a weight to each unit under test, which indicates the likelihood of the unit under test to be at the fault. Finally a ranking of the units under test is produced by sorting them with their weights in descending order. A unit under test executed more in failing tests and less in passing test gets a higher weight and appears at top position in the ranking.

Traditional spectrum based fault localisation (coverage based fault localisation) techniques merely use the coverage of units under test individually. These techniques ignore the dependency relationships between units under test, such as a combination of methods to be called together. Consequently, the top ranked unit under test may not be the root cause of failure. Additionally, the complex program executions cannot be reflected through coverage of individual units under test, hence it results into inaccurate outcome of these techniques [11]. For example, when the unit under test is a method, it can be executed in both passing tests as well as failing test. The similarity coefficient will rank the method lower. However, this method might be the potential candidate for the fault, if during its execution it followed a unique path (sequence of method calls) which led to the test failure. It turns out that the mere coverage of a program may not suffice to locate the fault, but the coverage of individual methods would do. It is also reported that method call sequences are better fault predictors than method coverage [12], [13].

In this paper we present a variation of spectrum based fault localisation which leverages the sequence of method calls as coverage elements, as opposed to a mere coverage of individual methods. More specifically, we collect the method call traces of each method during its execution, reduce these call traces of a method into call sequences and compare the call sequences between failing and passing tests using a similarity coefficient to assign a weight to each sequence. Next, each method gets a weight which is an average weight of its call sequences. Finally a ranking of the methods is produced.

We test the efficiency of our technique on two small, yet representative, JAVA projects (NanoXML and JMeter) with a limited number of faults, and demonstrate that our technique performs better than coverage based fault localisation techniques.

The remainder of this paper is organised as follows. Section II describes our technique, Section III describes experimental setup. Section IV reports the results along with their discussion and Section V concludes the paper.

TABLE I
A SAMPLE TRACE IN A TEST CASE

caller object id	caller id	callee id
o_1	m_0	m_1
o_1	m_0	m_1
o_1	m_0	m_2
o_1	m_0	m_3
o_1	m_0	m_2
o_2	m_0	m_1
o_2	m_0	m_1
o_2	m_0	m_1
o_1	m_1	m_5

II. OUR TECHNIQUE

Spectrum based fault localisation has been applied at different levels of granularity, including *statements* [6], *blocks* [7], *methods* [8], [9], and *classes* [12], [13]. In our technique, we choose the granularity level of *methods*, which is intermediate between low level *statements* and high level *classes*.

Our technique works in a scenario, where we have a failing test case and one or more passing test cases. We run these test cases and in each test case, we (1) collect the traces (Cf. Section II-A), slice the traces into individual method traces (Cf. Section II-B), reduce the sliced traces into call sequences for a method (Cf. Section II-C), and finally rank the methods (Cf. Section II-D) according to their likelihood of being at fault.

A. Collecting the trace

We collect in the trace for each method call (1) caller object id, (2) caller id, and (3) callee id. The 'caller object id' is the identifier of the caller object which calls the method, the 'caller' is the method from which the call is made and the 'callee' is the called method. In case a method is called from within the static method in the class context, there is no caller object, hence we mark the 'object caller id' as -1.

In each test case, we intercept the method call, collect the called method identifier, caller object identifier, the caller method identifier and record it in the trace. The method identifier is a unique integer associated with method name. A sample trace in a test case looks like as given in Table I

We only trace the methods belonging to the project and ignore methods other than project methods (such as JAVA library methods).

B. Slicing the trace

Once a trace for a test case is obtained, we slice the trace into individual method traces. Each sliced trace represents the trace for each executed method in the test case. It represents the execution profile of a method.

The sliced trace for a method m in a test case tc is as $T_m(tc) = \{t_1, t_2, ..., t_n\}$, where t_i represents the method calls invoked from method m through same caller object. If the method m is static, it may also be called as class context, in that case the 'caller id' is -1 and the calls appear in a single trace for 'caller id' -1.

Referring to Table I, $t_1 = \{m_1, m_1, m_2, m_3, m_2\}$ for the calls of method m_0 with object o_1 and $t_2 = \{m_1, m_1, m_1\}$ for the calls of method m_0 with object o_2. Therefore, the sliced trace $T_{m_0}(tc)$ for m_0 is as follows.

$$T_{m_0}(tc) = \left\{ \begin{matrix} \{m_1, m_1, m_2, m_3, m_2\}, \\ \{m_1, m_1, m_1\} \end{matrix} \right\} \quad (1)$$

C. Obtaining call sequences

As comparing the large sliced traces $T_m(tc)$ of methods between failing and passing test cases can be expensive, we reduce these call traces into a call sequence set for each method. To arrive at call sequence set $S_m(tc)$ for a method m, we adopt the *closed itemset mining algorithm* [14]. Given the sliced trace $T_m(tc)$ of method m in a test case tc, we define:

- X —*itemset*— a set of method calls.
- $\sigma(X)$ —support of X— the number of t_i in $T_m(tc)$ that contain this itemset X.
- $minsup$ —minimum support of X— a threshold used to tune the number of returned itemsets.
- *frequent itemset* — an itemset X is frequent when $\sigma(X) \geq minsup$.
- *closed itemset* — a frequent itemset X is closed if there exists no proper superset X' whose support is same as the support of X (i-e. $\sigma(X') = \sigma(X)$).

We refer a closed itemset X as a frequent call sequence or simply a call sequence. We fix $minsup$ to 1 to include call sequences for the methods called with one object only or for those called with a class context. Moreover, the mining algorithm also returns call sequences of length 1 comprising only one method call. But we exclude these from final set, a call sequence should comprise at least two methods. The call sequences set $S_{m_0}(tc)$ for method m_0 from sliced trace $T_{m_0}(tc)$ in Equation 1 is as follows.

$$S_{m_0}(tc) = \{\{m_1, m_3, m_2\}\} \quad (2)$$

D. Ranking methods

We rank the methods for their likelihood of being at fault using call sequences set $S_m(tc)$ of all methods from a test set. A test set contains one failing test case and one or more passing test cases. We obtain a call sequences set S_m (Equation 3) for each method m — which is the union of call sequences of the method in $S_m(tc_F)$ from failing test case and call sequences in $S_m(tc_{P_i})$ from passing test cases.

$$S_m = \{X | X \in S_m(tc_F)\} \cup \{X | X \in S_m(tc_{P_i}) \; and \; i \geq 1\} \quad (3)$$

To rank the methods, (1) we build a test coverage matrix (TCM) of call sequences for each method and, using a similarity coefficient [7], assign a weight to each sequence in the call sequences set S_m (Equation 3) of a method. Then, (2) we obtain the weight of a method by taking the average weight of its call sequences.

1) Weight per call sequence: Each call sequence $X \in S_m$ (Equation 3) gets a weight using a similarity coefficient. In principle, any similarity coefficient can be chosen from available in the literature, we used in our experiments three coefficients in order to see which performs better. The three similarity coefficients we used are the Jaccard (Equation 4), Tarantula (Equation 5) and Ochiai (Equation 6) as used in [7].

$$W_J(X) = \frac{a_{11}(X)}{a_{11}(X) + a_{01}(X) + a_{10}(X)} \tag{4}$$

$$W_T(X) = \frac{\frac{a_{11}(X)}{a_{11}(X)+a_{01}(X)}}{\frac{a_{11}(X)}{a_{11}(X)+a_{01}(X)} + \frac{a_{10}(X)}{a_{10}(X)+a_{00}(X)}} \tag{5}$$

$$W_O(X) = \frac{a_{11}(X)}{\sqrt{(a_{11}(X) + a_{01}(X)) * (a_{11}(X) + a_{10}(X))}} \tag{6}$$

Where:

- $a_{11}(X)$ = Number of failing tests in which $sequenceX$ is found.
- $a_{10}(X)$ = Number of passing tests in which $sequenceX$ is found.
- $a_{01}(X)$ = Number of failing tests in which $sequenceX$ is not found.
- $a_{00}(X)$ = Number of passing tests in which $sequenceX$ is not found.

2) Weight per method: Each method m gets a weight $W(m)$ as an average weight of its sequences (Equation 7). Those methods without call sequences set have weight 0.

$$W(m) = \frac{1}{|S_m|} \sum_{X \in S_m} W(X) \tag{7}$$

Finally, a ranking of all executed methods is produced using their weights W(m). The weight of the method indicates its likelihood of being faulty. Those methods with the highest weight appear top in the ranking.

III. Experimental setup

For our experiment we use the subject programs, also known as *probands* [8], [9], NanoXML and JMeter available from the Software-artifact Infrastructure Repository[1] [15]. The probands come with fault metrics and test cases. The particular details of the probands are reported in Table II

To obtain the rankings, we inject one fault at a time and trace each test case (Cf. Section II-A). Next, we calculate the ranking of methods for each failing test and related one or more passing tests. We ensure that we obtain the ranking with test cases which are related to each other. The test cases are related to each other if they activate the same feature of the program or they all belong to same JUnit test class.

For a comparison purpose, we also obtain the ranking with the traditional coverage based technique. We form a test set in which there is one failing test and one or more related passing tests and obtain a ranking with both our technique and the traditional coverage based technique. With this configuration, we have a total of 354 rankings for each technique.

[1]http://sir.unl.edu/portal/index.php

TABLE II
The Probands Used in Our Experiments

Proband	# of versions	UUTs*	# of faults
NanoXML	4	57	16
JMeter	2	81	3

*The number of average executed methods in the test cases

We measure the accuracy of our rankings as *wasted effort* [8], [9] — the number of methods (units under test) to inspect in vain before reaching the faulty method.

Limitation: We only trace the method calls that belong to the project itself, excluding JAVA library calls. Therefore, those faulty methods in our probands which do not call other project methods, have no call sequence (Cf. Section II-C). We had to drop these faults from each proband in our experiments. Consequently, we only report the exact number of faults in Table II that we used in our experiments.

IV. Results and discussion

As we compare our technique with the coverage based technique, we obtain two rankings with both techniques for each combination of a single failing test and multiple passing tests. Moreover, we obtain rankings with each of the three similarity coefficients mentioned in Section II-D. In total, we obtain 354 rankings.

In Table III, we report the average of wasted effort from all the rankings with our technique and coverage based technique for each of the similarity coefficient. In Table IV, we report the maximum wasted effort — maximum number of methods to search through — in 50% of the total rankings.

Discussion: In Table III, it can be observed that, on average, we have more than 50% improvement over the traditional coverage based technique. With our technique, using the Jaccard similarity coefficient, on average 8.6 methods need to be inspected in vain before finally reaching at the faulty method. On the contrary, 22.2 methods need to be inspected in vain with coverage based technique. Similarly, the 50% improvement holds using the Tarantula, Ochiai and Jaccard similarity coefficients.

Moreover, it can be observed from Table IV , as well as from Figure 1, that for 50% of rankings, a maximum of 4 methods (using Ochiai) need to be searched with our technique. Whereas, 16.5 methods need to be searched with coverage based technique, which is almost 4 times more.

V. Conclusion

Spectrum based fault localisation techniques are lightweight automated fault localisation techniques. These techniques localise the faults by comparing traces from failing and passing

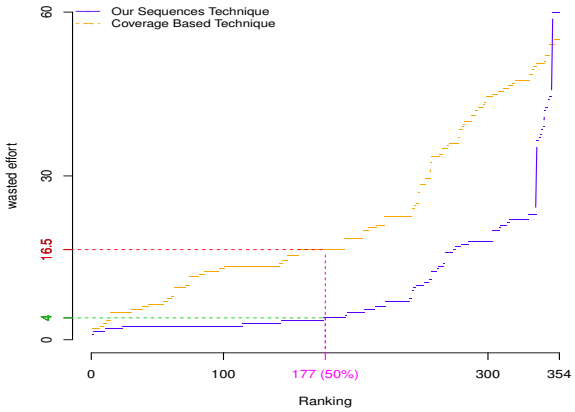

Fig. 1. The wasted effort in all the rankings calculated with Ochiai

TABLE III
THE AVERAGE ACCURACY MEASURED AS WASTED EFFORT

	Our Sequences Technique			Coverage Based Technique		
	Jaccard	Tarantula	Ochiai	Jaccard	Tarantula	Ochiai
μ^*	8.6	10.8	9.3	22.2	22.2	22.2
$\sigma^\#$	10.8	10.7	11.1	14.9	14.9	14.9

*Arithmetic mean

#Standard deviation

TABLE IV
THE MAXIMUMWASTED EFFORT IN 50% OF RANKINGS

Our Sequences Technique			Coverage Based Technique		
Jaccard	Tarantula	Ochiai	Jaccard	Tarantula	Ochiai
4.5	7.0	4.0	16.5	16.5	16.5

test cases, find statistical correlation between test case failures and the execution of the units under test.

As these techniques merely use the coverage of units under test, ignoring the dependency relationships between them, their diagnostic accuracy is limited and the results are often inaccurate.

In this paper we present an improved spectrum based fault localisation technique, which takes into account the call sequences of a method in a program. We demonstrate on a small number of faults that our technique outperforms the traditional coverage based technique. The diagnostic accuracy of our technique is 50% better than coverage based techniques.

ACKNOWLEDGMENTS

This work is sponsored by (i) the Higher Education Comission of Pakistan under a project titled "Strengthening of Univeristy of Sindh (Faculty Development Program)"; (ii) the Institute for the Promotion of Innovation through Science and Technology in Flanders through a project entitled "Change-centric Quality Assurance (CHAQ)" with number 120028.

REFERENCES

[1] S. H. Tan and A. Roychoudhury, "Relifix: Automated repair of software regressions," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 471–482. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818813

[2] 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 *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 14–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337226

[3] R. Saha, M. Lease, S. Khurshid, and D. Perry, "Improving bug localization using structured information retrieval," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 345–355.

[4] S. Rao, H. Medeiros, and A. Kak, "Comparing incremental latent semantic analysis algorithms for efficient retrieval from software libraries for bug localization," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 1, pp. 1–8, Feb. 2015. [Online]. Available: http://doi.acm.org/10.1145/2693208.2693222

[5] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 579–590. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786880

[6] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101949

[7] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2009.06.035

[8] F. Steimann and M. Frenkel, "Improving coverage-based localization of multiple faults using algorithms from integer linear programming," in *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, Nov 2012, pp. 121–130.

[9] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 314–324. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483767

[10] N. DiGiuseppe and J. A. Jones, "On the influence of multiple faults on coverage-based fault localization," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 210–220. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001446

[11] L. Zhao, L. Wang, Z. Xiong, and D. Gao, "Execution-aware fault localization based on the control flow analysis," in *Proceedings of the First International Conference on Information Computing and Applications*, ser. ICICA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 158–165. [Online]. Available: http://dl.acm.org/citation.cfm?id=1925974.1925998

[12] G. Laghari, A. Murgia, and S. Demeyer, "Localising faults in test execution traces," in *Proceedings of the 14th International Workshop on Principles of Software Evolution*, ser. IWPSE 2015. New York, NY, USA: ACM, 2015, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/2804360.2804361

[13] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for java," in *Proceedings of the 19th European Conference on Object-Oriented Programming*, ser. ECOOP'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 528–550. [Online]. Available: http://dx.doi.org/10.1007/11531142_23

[14] M. J. Zaki and C. J. Hsiao, "CHARM: an efficient algorithm for closed itemset mining," in *Proceedings of the Second SIAM International Conference on Data Mining, Arlington, VA, USA, April 11-13, 2002*, 2002, pp. 457–473. [Online]. Available: http://dx.doi.org/10.1137/1.9781611972726.27

[15] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10664-005-3861-2