International Journal of Software Engineering and Its Applications
Vol.7, No.5 (2013), pp.403-412
http://dx.doi.org/10.14257/ijseia.2013.7.5.35

Debugging in the Extreme: Spectrum-based Fault Localization with Limited Test Cases

Patrick Daniel[1] and Kwan Yong Sim[2]

[1]*Faculty of Engineering, Computing and Science*
Swinburne University of Technology
Kuching, Sarawak, Malaysia
[2]*Faculty of Engineering, Computing and Science*
Swinburne University of Technology
Kuching, Sarawak, Malaysia
[1]*pdaniel_koe@yahoo.com*, [2]*ksim@swinburne.edu.my*

Abstract

Spectrum-based Fault Localization (SBFL) is a well-known debugging technique that locates fault in program code by utilizing execution profiles (spectra) of pass and fail test cases. Hence, the performance of SBFL depends on the test cases executed and the test results. In the most extreme scenarios, the debugging process may have to be conducted with only one fail test case, one pass test case or no pass test case. These scenarios might occur due to extremely high or extremely low failure rates or when software testers decide to stop running more test cases due to time and resource constraints. However, limited test case execution profiles may reduce the accuracy of SBFL metrics. In view of this, we evaluate the performance of SBFL metrics in these extreme scenarios to identify the best performing SBFL metric for each of these scenarios. From the experiment results, we have further discovered the convergence in performance for SBFL metrics under these extreme scenarios.

Keywords: *Software Testing, Software Debugging, Spectrum-based Fault Localization, Limited Test Cases*

1. Introduction

Software testing and debugging are key activities in software development lifecycle to ensure the quality of the software developed. Software testing and debugging are not only the most expensive but also the most time consuming activity in software development life cycle [1]. Software testing researchers have proposed various strategies and fault localization techniques in the effort to reduce time and cost of testing and debugging. In practice, software testing aims to detect or reveal failures in the software developed through execution of test cases. On the other hand, software debugging is meant to locate and correct the faulty statement in the software which causes the failure to happen.

One of the effective approaches that can be applied in software debugging is Spectrum-based Fault Localization (referred to as SBFL) [2, 4, 8]. This approach works by utilizing statement execution information (spectra) obtained from the executed test cases. Based on the spectra, a ranking metric will be used to calculate the likeliness that the statement is faulty. Each statement is then ranked according to its likeliness to be faulty. The statement with the highest rank is most likely to be the faulty statement, but it is not guaranteed. In real life situation, this ranking will lead and guide the software debugger to locate the faulty statement. Therefore, the good SBFL technique will rank faulty statement on the top of the ranking.

ISSN: 1738-9984 IJSEIA

In software testing activity, the software testers are required to execute the software under test with test input (test case) in order to detect and reveal failures in the software. The number of test cases executed during the testing process might vary. It depends on the complexity of the software input domain, the test case selection strategy and the time and human resources available to run the tests. However, the number of test case executed during software testing affects the performance of SBFL metrics [5, 6]. Generally, more test case execution profiles will give more information about the location of faulty statement. This will in turn reduce the time consumed in the debugging process to locate the faulty statement. In contrary, fewer test case execution profiles will provide less information about the likely location of faulty statement, which in turn may increase the time consumed in the debugging process to locate the faulty statement. This tradeoff is illustrated the example in Figure 1. Initially, assume that the time consumed by software testing to execute test cases is equal to the time spent on software debugging. Next, by reducing the number of test cases, software tester could reduce the time consuming when testing process. However, fewer test cases could lead to less execution profiles or spectra and information about the likely location of the faulty statement. This, in turn, increases the time consumed in the debugging process to locate the faulty statement.

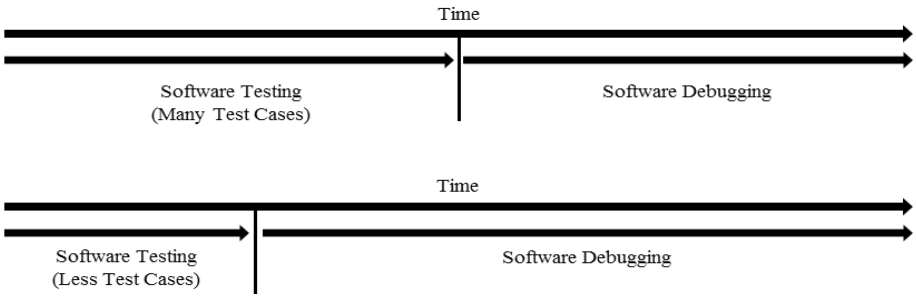

Figure 1. Tradeoff between the Time Consumed in Software Testing and Debugging Processes

SBFL relies on statement execution information (spectra) obtained from the executed test cases to rank statements in software code according to their likeliness to be faulty. In the most extreme scenarios, debugging may be started with only one failed test cases, one pass test cases or no pass test cases, which may have adverse effects on the performance of SBFL. This paper evaluates the performance of SBFL metrics under these extreme scenarios and identifies the best performing metrics.

The research questions of this paper are:

1. In these extreme situations, what is the performance of each SBFL metrics?
2. Which SBFL metrics have the best performance under the extreme scenarios?

Based on the results of our empirical study, we recommend SBFL metrics to be used on software debugging activity under each of these extremes scenarios. In these extreme debugging scenarios, we also found that SBFL metrics converge into groups of identical performances.

2. Preliminaries

In software testing, a test case is executed on the software under test and it output is compared with the expected output. The test case is categorized as a *pass test case* if the output

is the same as the expected output. On the other hand, if the output differ from the expected output, the test case is categorized as *fail test case*.

In SBFL, four spectra coefficients *aef*, *anf*, *aep*, and *anp* are computed for each statement in the software code based on execution profile of the pass and fail test cases. The coefficient *aef* represents the number of fail test cases that have executed the statement, whereas *anf* represents the number of fail test cases that have not executed the statement. Similarly, the coefficient *aep* represents the number of pass test cases that have executed the statement, whereas *anp* represents the number of pass test cases that have not executed the statement.

From *aef*, *anf*, *aep*, and *anp*, an SBFL metric is used to compute a ranking value for each statement in the software to rank its likeliness to be faulty. The higher an SBFL metric ranks the faulty statement, the better it is because less statement would need to be inspected before the faulty statement is successfully located in the debugging process. Therefore, the percentage of code to be inspected (*pci*) before the faulty statement can be located is commonly used to measure the performance of a spectra metric in SBFL.

2.1. Testing Objects

Siemens Test Suite has been selected as our testing objects as it is widely used to benchmark and evaluate the performance of SBFL metrics [12]. This Test Suite is available in public domain and can be downloaded from Software-artifact Infrastructure Repository maintained by University of Nebraska-Lincoln [11]. Siemen Test Suits contains seven programs with one correct version and multiple faulty versions each program. The test cases scripts are also included for each program. We use GCC version 4.6.1 and Gcov(GNU-GCC) on Ubuntu 11.10 to gather the spectra from Siemens test suite. Table 2 shows the total number of faulty versions, the line of code, the number of test cases and the description of each program in the Test Suite.

Table 1. Programs in Siemens Test Suite

Program	Faulty Versions	LOC	Number of Test Cases	Description	Versions excluded in experiments
print_tokens	7	563	4130	Lexical analyser	1, 2, 4, 6
print_tokens2	10	508	4115	Lexical analyser	10
replace	32	563	5542	Pattern recognition	12, 21, 32
schedule	9	410	2650	Priority scheduler	2, 7
schedule2	10	307	2710	Priority scheduler	9
tcas	41	173	1608	Altitude separation	10, 11 ,13, 14, 15, 31, 32, 33, 36, 38, 40
tot_info	23	406	1052	Information measure	6, 10, 19, 21

In our study, we execute all test cases for each program. Identical versions and faulty versions that produce identical outputs with the correct versions are excluded as SBFL cannot be conducted without fail test case. Furthermore, we focus our study on faulty versions with a single faulty statement. As a result, we have excluded *print_tokens* {v4, v6} because these versions are identical with the original version of the program. In addition, *print_tokens* {v1}, *replace* {v21}, *schedule* {v2, v7}, and *tcas* {v10, v11, v15, v31, v32, v33, v40} have also excluded because multiple faulty statements exist in program code. We have also excluded *print_tokens* {v2}, *replace* {v12}, *tcas* {v13, v14, v36, v38}, *tot_info* {v6, v10, v19, v21} because the faulty statement is a non-executable statement. Lastly, *print_tokens2* {v10}, *replace* {v32}, and *schedule2* {v9} have been excluded because these versions do not have any fail test case even though faulty statement existed in program code.

2.2. Spectrum-based Fault Localization Metrics

Software debugging researchers have proposed various SBFL metrics [2] in the attempts to accurately rank the statements in software code according to their likeliness to be faulty. Table 2 lists the SBFL metrics evaluated in our experiments.

Table 2. Spectrum-based Fault Localization Metrics

Name	Formula	Name	Formula
Naish1	$\begin{cases}-1 & if\ aef < F \\ P - aep & if\ aef = F\end{cases}$	Zoltar	$\frac{aef}{aef + anf + aep + \frac{10000anfaep}{aef}}$
Naish2	$aef - \frac{aep}{aep + anp + 1}$	Simple Matching	$\frac{aef + anp}{aef + anf + aep + anp}$
Jaccard	$\frac{aef}{aef + anf + aep}$	Sokal	$\frac{2(aef + anp)}{2(aef + anp) + anf + aep}$
Anderberg	$\frac{aef}{aef + 2(anf + aep)}$	Rogers & Tanimoto	$\frac{aef + anp}{aef + anp + 2(anf + aep)}$
Sorensen-Dice	$\frac{2aef}{2aef + anf + aep}$	Russel & Rao	$\frac{aef}{aef + anf + aep + anp}$
Dice	$\frac{2aef}{aef + anf + aep}$	AMPLE	$\left\lvert \frac{aef}{aef + anf} - \frac{aep}{aep + anp} \right\rvert$
qe	$\frac{aef}{aef + aep}$	Tarantula	$\frac{aef}{aef + anf} \Big/ \left(\frac{aef}{aef + anf} + \frac{aep}{aep + anp}\right)$
Wong1	aef	CBI Inc.	$\frac{aef}{aef + aep} - \frac{aef + anf}{aef + anf + aep + anp}$
Hamming etc.	$aef + anp$	Ochiai	$\frac{aef}{\sqrt{(aef + anf)(aef + aep)}}$
Binary	$\begin{cases}0 & if\ aef < F \\ 1 & if\ aef = F\end{cases}$	Euclid	$\sqrt{aef + anp}$
Kulczynski1	$\frac{aef}{anf + aep}$	AMPLE2	$\frac{aef}{aef + anf} - \frac{aep}{aep + anp}$
M1	$\frac{aef + anp}{anf + aep}$	M2	$\frac{aef}{aef + anp + 2(anf + aep)}$
Wong3	$aef - h, \quad where\ h = \begin{cases}aep & if\ aep \le 2 \\ 2 + 0.1(aep - 2) & if\ 2 < aep \le 10 \\ 2.8 + 0.001(aep - 10) & if\ aep > 10\end{cases}$		
Ochiai2	$\frac{aefanp}{\sqrt{(aef + aep)(anp + anf)(aef + anf)(aep + anp)}}$		
Arithmetic Mean	$\frac{2aefanp - 2anfaep}{(aef + aep)(anp + anf) + (aef + anf)(aep + anp)}$		
Geometric Mean	$\frac{aefanp - anfaep}{\sqrt{(aef + aep)(anp + anf)(aef + anf)(aep + anp)}}$		
Harmonic Mean	$\frac{(aefanp - anfaep)((aef + aep)(anp + anf) + (aef + anf)(aep + anp))}{(aef + aep)(anp + anf)(aef + anf)(aep + anp)}$		

Rogot2	$\frac{1}{4}\left(\frac{aef}{aef+aep}+\frac{aef}{aef+anf}+\frac{anp}{anp+aep}+\frac{anp}{anp+anf}\right)$
Cohen	$\frac{2aefanp-2anfaep}{(aef+aep)(anp+aep)+(aef+anf)(anf+anp)}$

3. Experiment Setup

In our experiment, we executed the test cases in Siemen Test Suite in normal scenario where the spectra of all pass and fail test cases were included to calculate the SBFL metric values to obtain the percentage of code inspected (pci) to locate the faulty statement. Subsequently, we repeated the experiment under three extreme scenarios with limited test cases scenario. The first scenario is "one fail all pass" where we used the spectra of one fail test case and all pass test cases to calculate the SBFL metric values and obtain the percentage of code inspected (pci) to locate the faulty statement. This experiment is repeated until every fail test case was selected to obtain the average pci for a SBFL metric under this extreme scenario. Second extreme scenario is "one pass all fail where we used the spectra of one pass test case and all fail test cases to calculate the SBFL metric values and obtain the percentage of code inspected (pci) to locate the faulty statement. This experiment is repeated until every pass test case was selected to obtain the average pci for a SBFL metric under this extreme scenario. The last scheme is "no pass" where we remove all pass test cases and retain only fail test case for the calculation of SBFL metric values.

These three scenarios emulate real life situations where SBFL has to be conducted with only one fail test case, one pass test case or no pass test case due to extremely high or extremely low failure rates or when software testers decide to stop running more test cases due to time and resource constraints. The performance of each SBFL metric is evaluated based on its average *pci* for all faulty versions of the seven programs in Siemen Test Suite.

4. Experiment Result and Discussions

The results of our experiments are presented in Table 3. The first column of the table lists of SBFL metrics under study. The second column is the SBFL metrics performance (in pci) for the "normal scenario" where the spectra of all pass and fail test cases were included to calculate the SBFL metric. The third, fourth and fifth columns present the performance of SBFL metrics the three extreme debugging scenarios with limited test cases.

Based on the experiment results in Table 3, it could be observed that most of the SBFL metrics, except for Kulczynski1 and Ochiai2, performed worse under the extreme "one fail all pass" scenario compared to the "normal scenario". However, some SBFL metrics with moderate performance in "normal scenario" such as {Anderberg, Dice, Jaccard, Sorensen-Dice, qe, Tarantula, M2, Ochiai} converged to the best performance {Naish1, Naish2, Zoltar} with pci value 8.16 under the extreme "one fail all pass" scenario.

Under extreme "one pass all fail" scenario, it could be observed that SBFL metrics {Jaccard, Snderberg, Sorensen-Dice, Dice, Simple_Matching, Sokal, Rogers&Tanimoto, Hamming_etc, Euclid, Wong3 } performed better than the "normal scenario", whereas SBFL metrics {Wong1, Russel&Rao, Binary} maintained the same performance and the remaining of the SBFL metrics performed worse compared to the "normal scenario".

Lastly, under the extreme "no pass test" scenario, most of the SBFL metrics converge into the same performance groups and mostly worse performance compared to the "normal scenario".

The best performing SBFL metrics under each of the three extreme debugging scenarios has also been identified. Under the "one fail all pass" scenario, the best performing SBFL metrics are {Naish1, Anderberg, Dice, Jaccard, Sorensen-Dice, qe, Tarantula, M2, Ochiai and Zoltar}.

Table 3. The Performance of SBFL Metrics under Extreme Debugging Scenarios with Limited Test Cases

SBFL Metrics	NORMAL SCENARIO	1 Fail All Pass	1 Pass All Fail	No Pass Test
Naish1	4.89	8.16	6.70	9.44
Naish2	4.80	8.17	6.62	9.36
Jaccard	7.73	8.16	6.62	9.36
Anderberg	7.73	8.16	6.62	9.36
Sorensen-Dice	7.73	8.16	6.62	9.36
Dice	7.73	8.16	6.62	9.36
Tarantula	8.05	8.16	10.62	9.36
qe	8.05	8.16	10.62	13.77
CBI_Inc.	9.03	9.14	29.68	13.77
Simple_Matching	14.94	15.40	6.60	9.36
Sokal	14.94	15.40	6.60	9.36
Rogers&Tanimoto	14.94	15.40	6.60	9.36
Hamming_etc.	14.94	15.40	6.60	9.36
Euclid	14.94	15.40	6.60	9.36
Wong1	9.36	11.78	9.36	9.36
Russel & Rao	9.36	11.78	9.36	9.36
Binary	9.44	11.78	9.44	9.44
Ochiai	6.23	8.16	6.62	9.36
M2	5.62	8.16	6.62	9.36
AMPLE2	9.36	11.78	88.69	9.36
Wong3	10.90	67.51	6.72	9.36
Arithmetic_Mean	7.73	9.15	28.19	9.36
Cohen	8.96	9.15	28.36	9.36
Kulczynski1	13.99	13.72	33.19	55.13
M1	21.30	21.76	34.38	55.13
Ochiai2	9.97	9.69	27.87	55.13
Zoltar	4.81	8.16	6.63	9.36
Ample	11.42	13.38	32.99	9.36
Geometric_Mean	7.96	9.15	28.38	9.36
Harmonic_Mean	8.28	10.57	30.29	9.36
Rogot2	8.28	10.57	30.29	9.36

On the other hand, under the "one pass all fail" scenario, the best performing SBFL metrics are {Euclid, Hamming_etc., Rogers&Tanimoto, Simple_Matching and Sokal}. Finally, under the "no pass test" scenario, the best performing SBFL metrics are {Ample, AMPLE2, Arithmetic_Mean, Cohen, Naish 2, Anderberg, Dice, Jaccard, Sorensen-Dice, Tarantula,

Euclid, Hamming_etc., Rogers&Tanimoto, Simple_Matching, Sokal, Russel & Rao, Wong1, Geometric_Mean, Harmonic_Mean, M2, Ochiai, Rogot2, Wong3 and Zoltar}. These findings can serve as a useful guide for SBFL practitioners to choose the best performing SBFL metrics to use under these extreme scenarios of limited test cases.

Table 4. The Best Performing SBFL Metrics under Extreme Debugging Scenarios with Limited Test Cases

SBFL Metrics	1 Fail All Pass	SBFL Metrics	1 Pass All Fail	SBFL Metrics	No Pass Test
Naish1	8.16	Euclid	6.60	Ample	9.36
Anderberg	8.16	Hamming_etc.	6.60	AMPLE2	9.36
Dice	8.16	Rogers&Tanimoto	6.60	Arithmetic_Mean	9.36
Jaccard	8.16	Simple_Matching	6.60	Cohen	9.36
Sorensen-Dice	8.16	Sokal	6.60	Naish2	9.36
qe	8.16	Naish2	6.62	Anderberg	9.36
Tarantula	8.16	Anderberg	6.62	Dice	9.36
M2	8.16	Dice	6.62	Jaccard	9.36
Ochiai	8.16	Jaccard	6.62	Sorensen-Dice	9.36
Zoltar	8.16	Sorensen-Dice	6.62	Tarantula	9.36
Naish2	8.17	M2	6.62	Euclid	9.36
CBI_Inc.	9.14	Ochiai	6.62	Hamming_etc.	9.36
Arithmetic_Mean	9.15	Zoltar	6.63	Rogers&Tanimoto	9.36
Cohen	9.15	Naish1	6.70	Simple_Matching	9.36
Geometric_Mean	9.15	Wong3	6.72	Sokal	9.36
Ochiai2	9.69	Russel & Rao	9.36	Russel & Rao	9.36
Harmonic_Mean	10.57	Wong1	9.36	Wong1	9.36
Rogot2	10.57	Binary	9.44	Geometric_Mean	9.36
AMPLE2	11.78	qe	10.62	Harmonic_Mean	9.36
Binary	11.78	Tarantula	10.62	M2	9.36
Russel & Rao	11.78	Ochiai2	27.87	Ochiai	9.36
Wong1	11.78	Arithmetic_Mean	28.19	Rogot2	9.36
Ample	13.38	Cohen	28.36	Wong3	9.36
Kulczynski1	13.72	Geometric_Mean	28.38	Zoltar	9.36
Euclid	15.40	CBI_Inc.	29.68	Naish1	9.44
Hamming_etc.	15.40	Harmonic_Mean	30.29	Binary	9.44
Rogers&Tanimoto	15.40	Rogot2	30.29	CBI_Inc.	13.77
Simple_Matching	15.40	Ample	32.99	qe	13.77
Sokal	15.40	Kulczynski1	33.19	Kulczynski1	55.13
M1	21.76	M1	34.38	M1	55.13
Wong3	67.51	AMPLE2	88.69	Ochiai2	55.13

Based on the results in Table 4, it could also be observed that the performance of SBFL metrics converge into groups of identical pci values. This is not surprising given that under the extreme scenario, the spectra coefficient will converge to a single value. For example, under the

"one pass all fail" scenario, the values of *aep* and *anp* can only be either 1 or 0. Similarly, under the "one fail all pass" scenario, the values of *aef* and *anf* can only be either 1 or 0. Lastly, under the "no pass test" scenario, *aep* and *anf* will have a value of 0. In these scenarios, groups of SBFL metrics will evaluate to the same value, hence resulting in groups of identical *pci*.

5. Conclusion

In order to save the time and cost in software testing phase, software testers could reduce the number of test cases. These savings at the testing phase may come at the expense of the performance of SBFL metrics in the debugging process. In the extreme scenarios, the debugging process may be started with only one fail test case, one pass test case or no pass test case. In addition to constraints in time and cost of testing, these scenarios also occur due to extremely high or extremely low failure rates.

However, limited test case execution profiles may reduce the accuracy of SBFL metrics. In view of this, we evaluated the performance of SBFL metrics in these extreme scenarios to identify the best performing SBFL metric for each of these scenarios. From the experiment results, we have further discovered the convergence in performance for SBFL metrics under these extreme scenarios.

In view of the better performance of some SBFL metrics these under extreme scenarios, we plan to further develop test case selection scheme to select a small set of test cases that may deliver better performance for SBFL metrics as our future work. In addition, we are conducting theoretical analysis on the performance of SBFL metrics under these scenarios.

Acknowledgement

This work is supported via Malaysian Government MOHE Fundamental Research Grant Scheme (FRGS/2/2010/TK/SWIN/02/03).

References

[1] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification", IBM Systems Journal, vol. 40, no. 1, **(2002)**.
[2] H. J. Lee, L. Naish and K. Ramamohanarao, "A model for spectra-based software diagnosis", ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 20, no. 3, **(2011)**, pp. 11.
[3] H. J. Lee, L. Naish and K. Ramamohanarao, "The Effectiveness of Using Non redundant Test Cases with Program Spectra for Bug Localization", Proceedings of the 2nd IEEE international Conference, Computer Science and Information Technology, ICCSIT, **(2009)** August 8-11, pp. 127-134.
[4] X. Y. Xie, T. Y. Chen and B. W. Xu, "Isolating Suspiciousness from Spectrum-Based Fault Localization Techniques", Proceedings of the 10th International Conference on Quality Software, **(2010)**, pp. 385-392.
[5] H. Lou, F. C. Kuo and T. Y. Chen, "Comparison of adaptive random testing and random testing under various testing and debugging scenarios", Software-Practice and Experience, **(2012)**, pp. 1055-1074.
[6] C. Gong, Z. Zheng, W. Li and P. Hao, "Effects of Class Imbalance in Test Suites: An Empirical Study of Spectrum-Based Fault Localization", Proceedings of the IEEE 36th International Conference on Computer Software and Applications Workshops, **(2012)**, pp. 470-475.
[7] R. Abreu, P. Zoeteweij and A. van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization", Proceedings of the 12th PRDC, **(2006)**, pp. 39-46.
[8] R. Abreu, P. Zoeteweij and A. van Gemund, "On the Accuracy of Spectrum-based Fault Localization", TAICPARTMUTATION, **(2007)**, pp. 89-98.
[9] R. Abreu, W. Mayer, M. Stumptner and A. van Gemund, "Refining Spectrum-based Fault Localization Rankings", SAC, Honolulu, Hawai, U.S.A, **(2009)** March 8-12, pp. 409-414.
[10] G. J. Myers, "The Art of Software Testing", 2nd edn. John Wiley and Sons, Revised and updated by T. Badgett and T. M. Thomas with C. Sandler: Hoboken, **(2004)**.
[11] 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, **(2005)**, pp. 405-435.

[12] M. Hutchins, H. Foster, T. Goradia and T. Ostrand, "Experiments on the effectiveness of dataflow- and control flow-based test adequacy criteria", Proceedings of the 16th International Conference on Software Engineering, **(1994)** May, pp. 191-200.

Authors

Patrick Daniel received his Bachelor of Science (CSSE) from Swinburne University of Technology in 2011. He is currently a Master of Science candidate at Swinburne University of Technology, Sarawak Campus, Malaysia. His research interest is software testing and debugging.

Kwan Yong Sim received his BEng (Hons) from the National University of Malaysia in 1999 and Masters of Computer Science from University of Malaya, Malaysia in 2001. He is currently a Senior Lecturer and the Associate Dean for Curriculum Enhancement and Accreditation at the Faculty of Engineering, Computing and Science, Swinburne University of Technology, Sarawak Campus, Malaysia. His research interests include software testing and for embedded system testing. Mr. Sim is a member of IEEE and IEEE Computer Society.