Fault Analysis in OSS Based on Program Slicing Metrics

Sue Black
Department of Computing
University of Westminster
Harrow Campus, UK
sueblack@gmail.com

Steve Counsell, Tracy Hall
Department of Computing
Brunel University
Uxbridge, UK
steve.counsell@brunel.ac.uk

David Bowes
Dept. of Computer Science
University of Hertfordshire
Hatfield, UK
d.h.bowes@herts.ac.uk

Abstract-**In this paper, we investigate the Barcode OSS using two of Weiser's original slice-based metrics (Tightness and Overlap) as a basis, complemented with fault data extracted from multiple versions of the same system. We compared the values of the metrics in functions with at least one reported fault with fault-free modules to determine a) whether significant differences in the two metrics would be observed and b) whether those metrics might allow prediction of faulty functions. Results revealed some interesting traits of the Tightness metric and, in particular, how low values of that metric seemed to indicate fault-prone functions. A significant difference was found between the Tightness metric values for faulty functions when compared to fault-free functions suggesting that Tightness is the 'better' of the two metrics in this sense. The Overlap metric seemed less sensitive to differences between the two types of function.**

Keywords-Slicing; faults; OSS; metrics

I. INTRODUCTION

Program slicing is an area that has recently attracted a range of research studies [20, 22, 23, 24, 25, 26, 30]. It is also an area that has developed its own set of metrics based on features of program 'slices'. However, while there are metrics devoted explicitly to measurement of program slicing, we still know very little about their behaviour and how useful they may be in illustrating features of the code they are extracted from. If we are to both appreciate and understand, for example, the relationship between program slicing and fault proneness, then we need to have a set of metrics that first quantitatively capture slice characteristics and second, can form the basis for sound statistical analysis (in this case with fault-proneness). Equally, if we want to build predictor models based on those metrics, then we need to understand the intricacies and vagaries of those metrics (in this case, of the two chosen slice-based metrics).

In this paper, we describe an empirical study of the Barcode Open Source System (OSS) and collected fault data from multiple versions of the same system. We explore whether two slice-based metrics (Tightness and Overlap [29]) can tell us anything about the propensity or otherwise of functions to be fault-prone or not. To inform our analysis, we collect fault-based data from multiple versions of the Barcode system and categorize a function as being either 'fault-prone' (i.e., contains at least one fault) or fault-free. Results suggest that of the two metrics, Tightness shows some promise in its ability to discriminate between fault-prone and fault-free functions. The same can not be said about the Overlap metric, however. The metrics also reveal insights into evolutionary features within the Barcode system and how functions might have behaved and deteriorated over time.

The remainder of the paper is organized as follows. In the next section we consider the motivation for the research and related work. We then describe details and definitions of the slicing metrics we collected (Tightness and Overlap) in Section 3. In Section 4, we describe an analysis of the research question posed and present data to support or refute that question. Finally we draw some conclusions (Section 5).

II. MOTIVATION AND RELATED WORK

The motivation for the work described in this paper stems from several sources. First, while a range of software metrics has been proposed for measuring cohesion [1, 4, 11, 13], very little research has been undertaken to determine the relationship between those metrics and the propensity for faults; work by Pan and Kim [26] used C language slicing metrics to compare the classification of faults with metrics for C++; yet we still know very little about software cohesion and even less about evolutionary trends in cohesion. Second, evolutionary studies have tended to focus on features of modules/classes (depending on the programming

978-0-7695-3784-9/09 $26.00 © 2009 IEEE
DOI 10.1109/SEAA.2009.94

paradigm being studied, i.e., procedural or OO). Very few studies have investigated the concept of cohesion from an evolutionary viewpoint. Third, vast numbers of software metrics have been proposed in the literature to measure different facets of software, but the majority of those metrics have not been validated in the sense that first, they have been shown to *actually* measure the software characteristic they purport to measure and second, are related to some dependent variable [19, 22]. In this paper, we try to form a relationship between two well-known and highly used slicing metrics and fault propensity. Finally, OSS is becoming an increasingly important part of software development integration policies for many organizations. We see an analysis of slicing metrics and faults in OSS as valid and useful as any corresponding analysis of faults in proprietary software. Tentatively, we may expect functions/classes with low cohesion to be more likely to be more fault-prone, since low cohesion often reflects a poorly written and/or poorly maintained artifact. The study presented here is based on that assumption.

The research in this paper is informed by two areas – namely, program slicing and cohesion (metrics) [2, 17 ,20]. In terms of slicing literature, the paper from which the slicing metrics were analyzed and which is considered the seminal slicing text is Weiser [29]. Since then, the techniques of program slicing have been adapted by many disciplines and for a multitude of contexts [5, 6, 7, 8]. Ott and Thuss explored some of Weiser's original metrics [29] and also introduced several of their own. These metrics were then analyzed from a largely empirical viewpoint. Bieman and Ott [4] used program slicing in the context of tokens and 'glue' that held those tokens together and is considered an influential study in the area. Meyers and Binkley [23] undertook a large-scale empirical study of five slice-based metrics (largely those of Ott and Thuss) and provide baseline values for those metrics on a longitudinal basis; lowly-rated modules according to those baselines would be candidates for re-engineering. The research also showed that the same set of metrics could be used to analyze the decay of systems.

As a software engineering concept, cohesion was introduced as early as 1979 when Yourdon and Constantine introduced a seven point ordinal scale for component cohesion [31]. Stevens et al. started looking at inter-module metrics even earlier [28]. Many other studies of different aspects of cohesion have followed [3, 10, 14]. In terms of the OO paradigm, the best known and most researched cohesion metric is the Lack of Cohesion of Methods (LCOM) proposed by Chidamber and Kemerer [11]. LCOM measures the relationship of methods and variables of a class by counting the number of method pairs accessing different variables, minus the number of method pairs accessing the same variables. A high LCOM for a class is undesirable and indicates high complexity in that class. The CAMC of Bansiya et al. [1] uses a similar principle of the distribution of variables across a class but with minor variations in interpretation from the LCOM. The research in this paper builds upon previous work by the authors comparing cohesion metrics [9, 13, 14] where a comparative study of OO cohesion metrics has highlighted the strengths and weaknesses of each.

III. PRELIMINARIES

A. Metrics definition/collection

The two metrics which we explore in this paper were originally proposed by Weiser [29], namely 'Tightness' and 'Overlap' and we use the same definitions of the metrics. Before formally defining the two metrics, we first describe the formal underpinnings of a slice's components proposed by Ott and Thuss [25] which we adopt in this paper.

We denote a set of variables used by a function F as V_F and V_O as the subset of V_F representing output (return) variables. F represents a program 'function', defined as a unit under consideration with a 'Length' defined as number of executable lines of code. We further note that in the OO paradigm, this would equate to a class, the level at which OO cohesion metrics have tended to be applied in past studies [1, 11, 13]. We denote a slice SL_i as that obtained for $v_i \in V_O$ and SL_{int} as the intersection of SL_i over all $v_i \in V_O$. We use the same example function used in [22] for consistency. This function is shown in Appendix A, the purpose of which is to determine the smallest and largest of an array of integers. The slice of each variable and intersection are shown. The two metrics and also the basis of the longitudinal, empirical study by Meyers and Binkley [22] are as follows:

$$\text{Tightness(F)} = \frac{|SL_{int}|}{length(F)}$$

Tightness measures the number of statements that occur in every slice.

$$\text{Overlap(F)} = \frac{1}{|\text{Vo}|} \sum_{i=1}^{|\text{Vo}|} \frac{|SL_{\text{int}}|}{|\text{SLi}|}$$

Overlap measures 'how many statements in a slice are found only in that slice [22]'. From the definitions of Tightness and Overlap, we obtain the following values for the function in Appendix A:

$$\text{Tightness} = \frac{11}{19} = 0.58 \quad \text{Overlap} = \frac{1}{2}\left(\frac{11}{14} + \frac{11}{16}\right) = 0.74$$

The relatively high value of Overlap is due to high value of SL_{int} relative to the size of the two slices for 'smallest' and 'largest'. The value of Tightness reflects the fact that SL_{int} accounts for just over half the function length.

B. Fault Extraction

We used the CodeSurfer tool [12] to extract the two metrics for every function from multiple versions of Barcode, an OSS written in C for processing barcode data and used in a previous study by Meyers and Binkley [23]. Nineteen versions of Barcode were studied as part of our analysis. The faults were extracted manually using the on-line report logs of Barcode. Henceforward, we distinguish between functions that contain at least one fault in any single version as 'fault-prone' and those that contained zero faults (in any single version) as 'fault-free'. The dataset was thus partitioned into two categories for the purpose of our analysis. Validating that CodeSurfer had extracted the 'correct' values of the two metrics was achieved through manual checking of the results (by one of the authors).

We note that the decision as to whether a function was fault-prone or fault-free, based on the report logs, was often made difficult because of the ambiguity or incompleteness in the reports. Consequently, 253 of the 775 functions in total had to be classed as 'don't knows' (i.e. we can not categorically say either way whether those functions contained a fault or not). We did not include these functions in the analysis presented on the basis that they might have posed a threat to study validity. One problem with the manual collection of faults is that it is often difficult to associate a reported fault with the function from which the fault was derived. In such a case, we can only say that we 'don't know' if that function actually contained a fault. We accept that leaving these functions out might have introduced a bias into our analysis. However, in our defense, inclusion of these functions would have clouded the clarity of the paper and perhaps posed an even greater bias than had we included them.

IV. DATA ANALYSIS

We explore the question of whether there were significant differences between the values of the Tightness and Overlap metrics with respect to the fault-proneness in Barcode functions. More specifically, the question we try to answer is: are either of the two metrics able to distinguish between functions with at least one fault and those without a fault? In this paper, we make the distinction between a 'module' and a 'function'; we define them in a one/many relationship where a single module may contain one of more functions.

A. Summary Data

Table I contains the summary data (number of functions (N) in each category, mean, maximum, standard deviation (SD) and median) for the two metrics for all functions in the two categories.

TABLE I. SUMMARY DATA FOR TIGHTNESS AND OVERLAP

	N	Mean	Max	SD	Median
Tightness (fault-prone)	372	*0.32*	0.99	0.32	*0.21*
Tightness (fault-free)	150	*0.38*	1.00	0.37	*0.28*
Overlap (fault-prone)	372	*0.59*	1.00	0.33	*0.63*
Overlap (fault-free)	150	*0.63*	1.00	0.38	*0.72*

A clear trend from Table I is the relatively higher values of Tightness and Overlap for fault-free functions in both the mean and median values (these values are italicized). This suggests, at face value, that the higher the cohesion (given by these two metrics) the lower the propensity of faults in the functions studied. It is also noticeable that the maximum value of Tightness for fault-prone functions was 0.99, but was 1.00 for fault-free functions (the two Overlap values are both 1.00). From the data in the table, it would appear that a salient characteristic of a fault-free function is a relatively high value of the Tightness and Overlap metrics. According to their formulas, one means of achieving this is to have a high $SL_{int.}$ In other words, a high slice intersection may be an indicator of a fault-free function. While this might seem a premature claim to make, we are mindful of the fact that the distribution, intersection and use of variables in an OO class is a key

feature of the LCOM and CAMC metrics of Chidamber and Kemerer [11] *and* Bansiya et al. [1], respectively. In both of those metrics, a high overlap is reflective of a cohesive class. It appears that the same principles might apply to C functions in Barcode as they do to OO classes for the LCOM and CAMC; that is, we can attain high cohesion through a high intersection of variables to prevent that function from becoming faulty. In a programming sense, this makes practical sense: we would expect a functionally cohesive 'unit' to optimize the extent of interactions between the declared variables and contain no redundant statements.

B. Tightness and Faults

Fig. 1 shows the values of the Tightness metric (*y*-axis) for fault-prone functions (*x*-axis) and Fig. 2 the Tightness metrics for fault-free functions. A prominent feature of Fig. 1 is the relatively large number of Tightness values 'on or around' the zero mark. Inspection of the source data revealed that these values belonged to just four functions in four different modules. The first was the function 'Barcode_ps_print' in the module 'ps.c', the second the 'main' function in the module 'sample.c', the third the function 'Barcode_Delete' in the 'library.c' module and the last in the function 'add_one' of the module 'code_39.c'. These four observations occurred in versions 12, 2, 2 and 8, respectively. The same feature (of relatively large numbers of small Tightness values) is evident in Fig. 2.

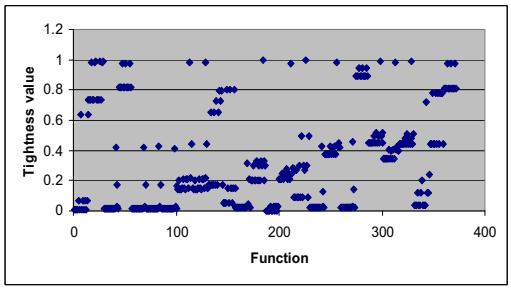

Figure 1. Tightness values for Barcode (fault-prone)

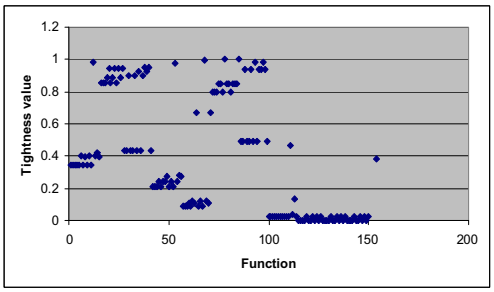

Figure 2. Tightness values for Barcode (fault-free)

Table II summarizes the frequencies of the Tightness metric values in fault-prone and fault-free functions. The high percentage of small Tightness values is evident from the table contents. Clearly, a low Tightness value (and, by implication, a low intersection of slices) seems to be a feature of a fault-prone function. Fig. 3 shows the extent of the difference between the fault-prone and fault-free functions; namely, analysis of the source data revealed that 70.2% of the fault-free functions were < 0.5, compared with 77.4% for fault-prone functions in the same range.

TABLE II. FREQUENCY OF TIGHTNESS VALUES

Range/ Category	0 – 0.199	0.2– 0.399	0.4– 0.599	0.6- 0.799	0.8- 1.00
Fault-prone	172	65	55	26	54
Fault-free	63	24	19	2	42

This observation supports the view that the incidence of faults in the Barcode system were for functions where there was a low value of SL_{int}. For fault-free functions therefore, a higher value of SL_{int} may be more desirable. Of course, function size does play a part in this, since the formula for Tightness implies that a small function also contributes to a high Tightness value; however, inspection of source data revealed no trend for fault-free functions to be smaller (in LOC) than the corresponding fault-prone functions.

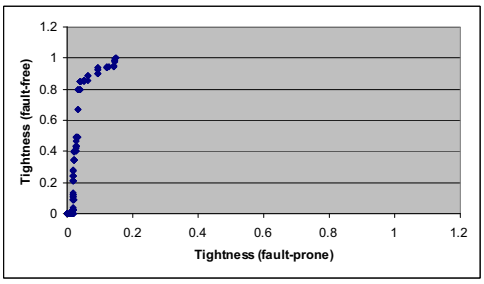

Figure 3. fault-prone vs. fault-free (Tightness)

Analysis of both fault-prone and fault-free functions revealed the majority in each to be close to the median for all functions (value 35 LOC) and neither large nor small in magnitude.

C. Overlap and Faults

Fig. 4 shows the data for the Overlap metric. In contrast to the Tightness metric, there are only a small set of zero values. Inspection of the source data revealed that these zero values featured exclusively to the 'main' function in the module 'sample.c'. No other zero values in any other modules/functions were reported. There is a simple explanation for this feature of main. A main function will differ from a 'regular' function in terms of the number of declarations and types of variables and, in particular, when considering the formula for Overlap – the lack of use of output (return) types may be a bias. Put another way, main is a special type of function and is unlikely to have the same type and distribution of variables and statements as a more 'regular' function.

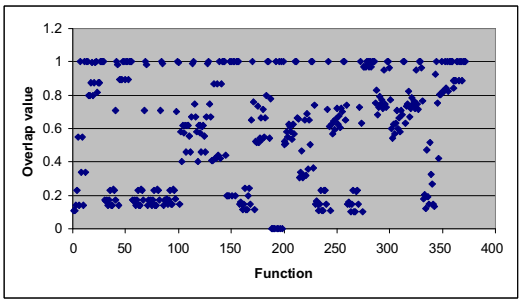

Figure 4. Overlap values (fault-prone)

Fig. 5 shows the values of the Overlap metric for fault-free functions. The values in Fig. 5 are far more polarized than those in Fig. 4. There are significantly more metric values close to, or with values of '1' relative to the proportion in Fig. 4. It is also interesting that the number of zero values in Fig. 5 is larger than that in Fig. 4. Scrutiny of the source data revealed that the zero values for Overlap shown in this figure are actually a 'superset' of the small set of zero values shown in Fig. 4, but only start to feature at a later version of the system. These zero values are exclusive to the 'main' function in the 'sample.c' module. From an evolutionary perspective, this is an interesting system characteristic. The values of the Overlap metric, in this case, indicate that significant maintenance effort may have been applied to the main function over the course of the versions studied. This claim is made purely on the basis that in earlier, fault-prone versions (i.e., versions 2&3) of Barcode, there were only 8 zero values; in later versions of fault-free functions (3, 4, 5&6) there were 24 zero values. In other words, as the Barcode system has evolved, the function main appears to have been modified significantly and that these modifications have been reflected in more zero Overlap values. The Overlap metric may therefore assist in identifying highly-changed functions as opposed to fault-prone functions.

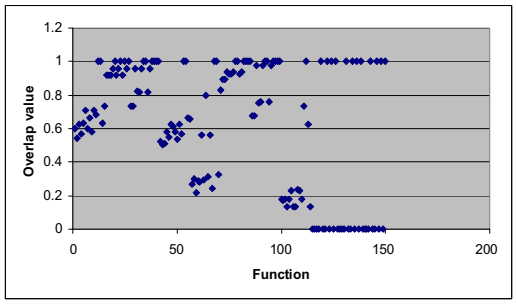

Figure 5. Overlap values (fault-free)

Table III shows the distribution of Overlap values for fault-prone and fault-free modules. The contrast between the profile in Table 3 and that in Table 2 (for Tightness) suggests that a high value of Overlap is found for *both* fault-prone and fault-free functions.

TABLE III. FREQUENCY OF OVERLAP VALUES

Range/Category	0 – 0.199	0.2 – 0.399	0.4 – 0599	0.6-0.799	0.8-1.00
Fault-prone	87	34	46	81	124
Fault-free	33	12	13	25	67

Fig. 6 shows the values of Overlap for fault-free versus fault-prone functions (c.f. Fig. 3 for the Tightness metric).

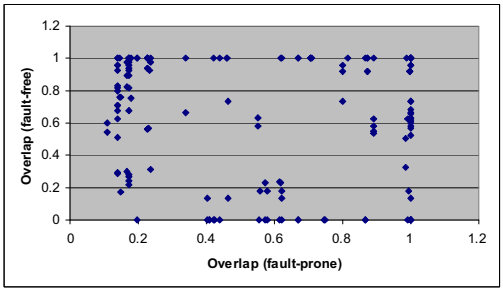

Figure 6. fault-prone vs. fault-free (Overlap)

The differences between the two sets of metrics are evident from the respective scatter plots. The fault-prone Overlap values appear to be either very high or very low. The values of the fault-free functions, on the other hand, are more evenly distributed. Further analysis of the data revealed that Overlap values for fault-prone functions tended to be higher in earlier versions of Barcode and then rapidly decreased thereafter; the values of Overlap for *fault-free* functions were far more evenly distributed from an evolutionary perspective. One conclusion that we draw from the prior analysis is that the 'smoother' profile for the fault-free functions, given by the Overlap metric values, may actually be a contributing factor to the extent of their lack of fault-proneness. This would also make sense from a maintenance point of view, since a function whose Overlap value fluctuates wildly over time is likely to have been the subject of significant variable modification, addition and deletion (or combinations of the three) in that period. Tightness seems to have more influence on the fault-proneness of a function – and Overlap on its changeability. In terms of the original research question, only one of the slicing metrics (Tightness) seems to display a correspondence with faults in a function and this is related to the distribution and interaction of variables given by the slice intersection.

D. Tightness versus Overlap

The range of the values found for Tightness and Overlap is also a notable feature of the analysis. Scrutiny of the source data and Figs. 1 to 6 shows that Overlap, in most cases, is always approximately 0.2 in excess of the value of Tightness. This explains why in Fig. 1 there are a high number of zero (or close to zero) values which are not evident in Fig. 4, for example. This also explains the relatively large number of '1s' for the Overlap metric in Figs. 4&5 when compared with Figs. 1&2. The data in the study by Meyers and Binkley [23] (which also used Barcode) showed this relationship. Fig. 7 Tightness and Overlap metrics on the same line graph and illustrates the extent of the correlation between the two metrics.

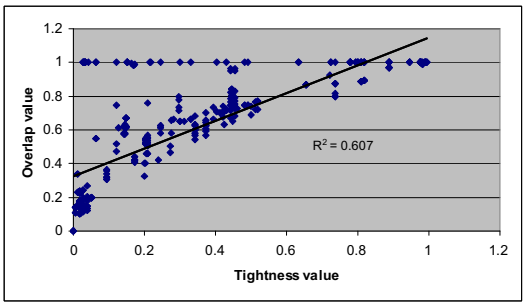

Figure 7. Tightness versus Overlap (fault-prone)

As well as the number of zero or close to zero values for the two metrics, the frequency of '1s' in each of Figs. 1-6 is also worth exploring. From the definition of the Tightness metric, a value of one is obtained when the value of SL_{int} equals the size of the module. For fault-prone modules, we found that no functions had a Tightness value of '1'. For the same set of data, there were 76 values of '1' in the Overlap metrics. For fault-free modules, there were just 2 Tightness values of '1' and 41 values of 1 for Overlap. This clearly shows the bias of Overlap in terms of generally producing higher values than those of Tightness. A further characteristic of Figs. 1-6 is the occurrence of small 'clusters' of metric values in the same region of the graph. For example, visual inspection of Fig. 1 shows one grouping of values on the 0.4 boundary and another grouping just above the 0.4 boundary. Equally, Fig. 2 shows one large grouping which extends just below and above the 0.2 boundary. This was an unexpected feature to emerge from our analysis and is a feature of both fault-prone and fault-free functions. Scrutiny of the source data revealed that these clusters were due to single functions whose values for Tightness and Overlap changed very little over the course of the versions studied. These small clusters might provide opportunities for reengineering or even refactoring [17] if the metric values are considered too small. For a fault-prone function exhibiting these characteristics, it might be worth targeting these clusters, especially if, as we hypothesized earlier, low Tightness values might be a contributing feature to a fault-prone function.

E. Statistical Analysis

Finally, to determine if there were statistically significant differences between the two groups (i.e., Tightness (fault-prone vs. fault-free) and Overlap (fault-prone vs. fault-free)), we ran Wilcoxon's signed rank test [27] and found the Tightness categories (fault-

prone vs. fault-free) to be significant at the 5% level (Z= -2.09). The same test for the Overlap categories, however, revealed significance at the 10% level only (Z=-1.78). In other words, there is significant difference between the two categories for Tightness, but not as strongly for Overlap. Spearman's and Kendall's non-parametric tests showed a significant, negative correlation for Tightness at the 1% level; for Overlap, no significant correlation was observed. Both of these results lend further credibility to the claim that the Tightness (but not the Overlap) metric produces significantly different values for fault-prone functions than for fault-free functions and that, as a result, it might offer some predictive capability.

F. Fault-prone functions (evolution)

One criticism that could be leveled at the study is that while we have studied the two metrics and their values, this tells us nothing about the propensity for fault-prone functions as the Barcode system evolved. In theory, we might expect a system to become more fault-prone as it evolves and as code 'decay' becomes a feature of the system due to continued maintenance. Fig. 8 shows the trend in fault-prone functions over the 19 versions of barcode studied where at least one fault-prone function occurred (N.b. 6 versions of Barcode had zero identified fault-prone functions. One theory for the 'peak and trough' distribution evident in Fig. 8 is that after an initial 'flurry' of a relatively large number of fault-prone functions in early versions due to changes in requirements, the system then recovers (given by a fall in the number of fault-prone functions) before faults begin to re-appear. Effort is applied again to address the second wave of faults before the system starts to stabilize.

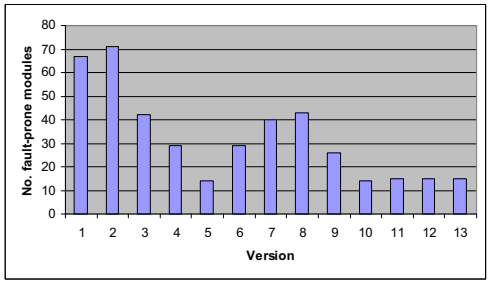

Figure 8. Fault-prone functions (evolutionary)

Consistent application of re-engineering or refactoring [17] over versions can often be the cause of such a fluctuating trend.

V. CONCLUSIONS

In this paper, we described an empirical study of the relationship between two metrics for program slicing and the faults generated by the functions of Barcode. We explored, using two metrics (Tightness and Overlap) and fault data extracted from the logs of the Barcode system, whether the two slicing metrics could illuminate features of either fault-prone or fault-free functions. We found that the low values of the Tightness metric showed some promise in terms of its ability to highlight fault-prone functions. The Overlap metric on the other hand was found to be useful for highlighting maintenance activity. Statistical support showed that there was a significant difference between the Tightness metrics for fault-prone and fault-free functions. One avenue of immediate future research would be to compare the results found in this paper with a closed-source, proprietary system. We accept that the analysis and results are presented for one system only and this presents a threat to the validity of the study. We therefore encourage replication and further studies in this area; to that end the data used in this study is available upon request of the authors.

ACKNOWLEDGEMENT

The research is supported by the UK Engineering and Physical Sciences Research Council (EPSRC) (EP/E055141/1).

REFERENCES

[1] Bansiya, J., Etzkorn, L., Davis, C., and Li, W. A class cohesion metric for object-oriented designs. Journal of Object-Oriented Programming 11(8), pp. 47-52, 1999.

[2] Basili, V., Briand, L., and Melo, W. A validation of object-oriented design metrics as quality indicators, IEEE Trans. on Software Engineering 22(10), 751-761, 1996.

[3] Bieman, J., and Kang, B.-K. Cohesion and reuse in an object-oriented system. Proceedings of ACM Symposium on Software Reusability, Seattle, Wash., pp. 259-262, 1995.

[4] Bieman, J., and Ott, L. Measuring functional cohesion. IEEE Trans. on Software Eng. 20, 8 (1994), pp. 644-657.

[5] Binkley, D. Gold, N. and Harman, M. An empirical study of static program slice size. ACM Trans. Software Engineering Methodology (TOSEM) 16(2):1-32, 2007.

[6] Binkley, D., Harman, M., and Krinke, J., Empirical study of optimization techniques for massive slicing. ACM Trans. Program. Lang. Syst. 30(1): (2007)

[7] Binkley D and Harman M., Locating dependence clusters and dependence pollution, IEEE International Conference on Software Maintenance, Budapest, Sept. 2005 pages 177-186.

[8] Binkley, D., Harman, M., Raszewski, I., and Smith, C. An empirical study of amorphous slicing as a program comprehension tool. Proc. of the Intl. Workshop on Program Comprehension, Limerick, Ireland, pp. 161-170, 2000.

[9] Bowes, D., Counsell, S and Hall, T., Calibrating program slicing metrics for practical use, Proceedings of TAIC PART, Windsor, UK, 2008, Computer Society Press.

[10] Briand, L., Daly, J., and Wust, J. A unified framework for cohesion measurement in object-oriented systems. Empirical Software Engineering Journal 3(1), 65-117, 1998.

[11] Chidamber, S., and Kemerer, C. A metrics suite for object oriented design. IEEE Trans. on Soft. Eng. 20(6) (1994), 467-493.

[12] www.grammatech.com/products/codesurfer/

[13] Counsell, S., Swift. S., and Crampton J. The Interpretation and Utility of Three Cohesion Metrics for Object-Oriented Design. ACM Trans. on Software Eng. and Methodology, 15(2):123 – 149, 2006.

[14] Counsell, S., Bowes D., and Hall T., Evolutionary Cohesion Metrics: The Empirical Contradiction. Proceedings of The Psychology of Prog.Interest Group (PPIG), Open University, January 2009.

[15] El Emam, K., Benlarbi, S., Goel, N., Rai, s., The Confounding Effect of Class Size on the Validity of OO Metrics. IEEE Trans. Soft Eng, 27(7):630-650 (2001).

[16] Fenton, N., Pfleeger, S. Software Metrics, A Rigorous and Practical Approach Thomson Intl. Comp. Press, (1996).

[17] Fowler, M. Refactoring (Improving the Design of Existing Code). Addison Wesley, 1999.

[18] Gold, N., Harman, M., Binkley, D., Hierons, R., Unifying program slicing and concept assignment for higher-level executable source code extraction. Softw., Pract. Exp. 35(10): 977-1006 (2005).

[19] Harrison, R, Counsell, S and Nithi, R. An Evaluation of the MOOD Set of Object-Oriented Software Metrics, IEEE Trans. on Soft. Engineering, vol. 24(6), pp. 491-496, 1998.

[20] Horwitz, S, Reps, T. and Binkley, D., Interprocedural Slicing Using Dependence Graphs. ACM Transactions on Programming Language and Systems, 12(1): 26-60, 1990.

[21] Kitchenham B., Pfleeger S. L., and Fenton, N., Towards a Framework for Software Measurement Validation, IEEE Trans. on Software Engineering, 21(12), pp. 929-944, 1995.

[22] Meyers, T and Binkley, D. Slice-based Cohesion Metrics and Software Intervention, Proceedings Working Conference on Reverse Engineering, Delft, Netherlands, pages 256-265.

[23] Meyers, T. and Binkley, D. An empirical study of slice-based cohesion and coupling metrics. ACM Trans. on Software Engineering and Methodology, 17(1), 2007.

[24] Ott L, Thuss J., (1993) Slice based metrics for estimating cohesion; Proc Software Metrics, 71–81, Baltimore, US.

[25] Ott L. and Thuss, J., The relationship between slices and module cohesion. ICSE Proceedings, Pittsburgh, US, 1989, pages 198-204.

[26] Pan, K., Kim, S., Bug Classification Using Program Slicing, Prcoeedings of IEEE Workshop on Source Code Analysis and Manipulation, Philadelphia, US, 2006, pages 31-42.

[27] Snedecor, G., and Cochran, W. Statistical Methods, 8th ed. Iowa State University Press, Ames, Iowa, 1989.

[28] Stevens, W., Myers, G., and Constantine, L. Structured design. IBM Systems Journal 13, 2 (1974), 115-139.

[29] Weiser, M. Program slicing. Proceedings Int. Conf on Soft Eng., San Diego, 1981. IEEE Press, pp. 439-449.

[30] Weiser M (1982) Programmers use slices when debugging, Comm. of the ACM, 25(7):446-452, July 1982

[31] Yourdon, E., and Constantine, L. Structured Design. Prentice Hall, Englewood Cliffs, New Jersey, 1979.

APPENDIX A - Function slices taken from [22]

Function	$SL_{smallest}$	$SL_{largest}$	SL_{int}
main()			
{			
int i;	|	|	|
int smallest;	|	|	|
int largest;		|	
int A[10];	|	|	|
for (i=0; i <10; i++)	|	|	|
{			
int num;	|	|	|
scanf("%d", &num);	|	|	|
A[i] = num;	|	|	|
}			
smallest = A[0];	|	|	|
largest=smallest;		|	
i=1;	|	|	|
while (i <10)	|	|	|
{			
if (smallest > A[i])	|		
smallest = A[i];	|		
if (largest < A[i])		|	
largest = A[i];		|	
i = i +1;	|	|	|
}			
printf("%d \n", smallest);	|		
printf("%d \n", largest);		|	
}			
Length =19	14	16	11