Software Defect Prediction Using Call Graph Based Ranking (CGBR) Framework

Burak Turhan, Gozde Kocak, Ayse Bener
Dept. of Computer Engineering, Bogazici University, Turkey
{turhanb, gozde.kocak, bener}@boun.edu.tr

Abstract

Recent research on static code attribute (SCA) based defect prediction suggests that a performance ceiling has been achieved and this barrier can be exceeded by increasing the information content in data [18]. In this research we propose static call graph based ranking (CGBR) framework, which can be applied to any defect prediction model based on SCA. In this framework, we model both intra module properties and inter module relations. Our results show that defect predictors using CGBR framework can detect the same number of defective modules, while yielding significantly lower false alarm rates. On industrial public data, we also show that using CGBR framework can improve testing efforts by 23%.

1. Introduction

Defect rate is a key indicator of software product quality. High defect rates increase the cost of development, maintenance and lead to customer dissatisfaction. The aim is to keep the defect rate as low as possible and catch them as early as possible. The statistics show that in successful projects the testing phase takes up to 50% of the total project cost [9, 25]. Therefore an effective testing strategy would shorten the testing period, the minimize number of defects and improve software quality.

Recent research shows that, companies can benefit from defect prediction tools to effectively manage their testing phases. Using test resources effectively would significantly lower the costs of the company. Numerous software metrics and statistical models have been developed in order to predict defects in software [7, 10, 13, 14, 15, 16, 20, 28]. There are several approaches to tackle the problem. Some researchers use size and complexity metrics that are collected at different phases of development, some use process and human centric metrics to measure the quality of the development process, and some others prefer combining multiple approaches [7, 10, 13, 14, 15, 19].

Architectural characteristics of software also play an important role in predicting the defects. Design metrics are useful to realize the quality of a design. Architects use design metrics to decide which parts to rework, they evaluate aspects of a design and they can better focus on the problem areas such as testing and/ or re-engineering effort. For example, fan-in fan-out metrics give important clues over the complexity, code redundancy and maintainability of the software [12].

Although there are some doubts on the value of static code attributes (i.e. size and complexity metrics) [24], they are shown to be useful, easy to use and widely employed in several research [16, 17, 23, 26, 27, 29]. Basic static code attributes, like lines of code, Halstead attributes and McCabe attributes can be cheaply and automatically collected even for large systems. However, Menzies et.al., in a recent research, empirically show that static code attributes have limited information content i.e. their information can be quickly and completely discovered by even simple learners [18]. They also discuss that defect predictors based on static code attributes have reached a performance ceiling and further improvement would need increasing the information content of data either by incorporating metrics from different sources, i.e. requirements phase, or by defining human in the loop approaches.

Considering the results of Menzies et.al., we aim at increasing the information content of static code attributes by investigating the architectural structure of the code. More precisely, static code attributes assume the independence of software modules and measure individual module complexities without considering their interactions[1]. In this research we

[1] Fan-in fan-out metrics are frequency counts of module calls and do not represent the architectural structure of the overall software. Thus, they do not violate the module independence assumption.

978-0-7695-3276-9/08 $25.00 © 2008 IEEE
DOI 10.1109/SEAA.2008.52

adjust static code attributes by modeling their interconnection schema. Previous research mostly focused on intra module metrics [8, 21, 30]. In this research we combine both intra module and inter module metrics.

We claim that module interaction and structure play an important role in defect proneness of a given code. Therefore we propose a model to investigate the module interactions. We use the static call graph structures in our research. Then we calculate call graph based ranking (CGBR) values and assign ranks to each module. We have used the page rank algorithm (web link based ranking algorithm) in constructing our call graph based ranking algorithm [4, 6].

In our experiments we used data from a large-scale white-goods manufacturer [5]. We constructed call graphs for every project and then calculated CBGR values for all modules in each project. Then we adjusted our data attributes with CGBR values. Afterwards we used the Naïve Bayes model for defect prediction [16, 28]. Our results show that probability of false alarm is decreased significantly while recall is retained, thus the testing effort is improved.

The rest of the paper is organized as follows: In section 2 we review the literature, in section 3 we explain our data sets. Call graph based ranking algorithm is discussed section 4. Section 5 explains our experiment and results. The last section gives conclusion and future direction.

2. Related work

Various methods such as linear regression analysis, discriminant analysis, decision trees, neural networks and Naïve Bayes classification are used for defect prediction in previous research [11, 13, 14, 16, 17, 19, 20, 26, 27, 28, 29]. Menzies et.al. state that "*The current status of the research in learning defect predictors is curiously static*" [18]. They discuss that when complex algorithms no longer improve the prediction performance we should find ways to get a better insight for the training data. They suggest that improving the information content of the training data should replace investigating more complex learners [18].

Static code measures defined by McCabe and Halstead are module-based metrics like lines of code, complexity and number of unique symbols. While some researchers opposed using static code measures [24], recent research showed that using Naïve Bayes classifier with log-filtering static code measures produced significantly better results than rule based methods [16]. Additionally, various researches showed the usefulness of static code attributes [13, 14, 16, 17, 26, 27, 28, 29].

Since we aim at combining static code attributes with call graph information for defect prediction, we should carefully differentiate our research from other call graph related studies. In addition to the abovementioned studies on static code attributes (SCA), there are recent studies investigating call graph-defect relations [8, 21, 30]. While SCA related studies examine only intra module metrics, call graph related studies examine only inter module relations.

Our contribution in this research is to combine both approaches. We show that combining inter and intra module metrics not only increases the performance of defect predictors but also decreases the required testing effort for locating defects in the source code.

3. Data

In order to validate our approach, we have built our defect predictors from static code attributes adjusted by module interactions (i.e. CGBR values). To train these predictors, static code attributes are formed as a table, where columns describe software attributes such as lines of code, McCabe and Halstead metrics. Each row in the data table corresponds to data from software modules, where we define a module as the functions/ procedures in source code. We also include another column that indicates the corresponding modules' fault data (i.e. whether it is defective or defect free). The goal of defect predictor is to find the relations of attribute columns that predict values in defects column. If such relations are found, test team can use them to determine where to start testing [16].

We use data from a local company that develops embedded systems software [5]. This data source includes three different projects. These projects are embedded controller software for white-goods and named as AR3, AR4, and AR5. We collected 29 static code attributes, such as LOC, Halstead, and McCabe metrics from these projects. We also collected defect information for each module in AR3, AR4 and AR5, and manually matched these with the software modules. Table1 shows general properties of all three projects used in this study.

The projects are written in C programming language and we have access to source codes of the projects. Public data sets do not contain call graph information [22]. Therefore, we used the available source code to build the static call graphs. We created NxN matrix for building a call graph where N is the

number of modules. In this matrix, rows contain the information whether a module calls the others or not. We represent these by using 1 and 0 respectively. Columns contain how many times a module is called by other modules. Table 2 shows an example call graph matrix.

Table 1. Datasets used in this study

Project	Attributes	# Modules	Defect Rate
AR3	29	63	0.12
AR4	29	107	0.18
AR5	29	36	0.2

We built the call graphs for every project and calculated call graph based ranking values for all modules of the three projects. We adjusted static code attributes as described in the next section. Then we ran simulations as if each project were new. Thus, each time we learned defect predictors from two completed projects and tested the predictor on the new project.

Table 2. Sample call graph matrix

	Module A	Module B	Module C	...	Module X
Module A	0	0	1	...	1
Module B	1	0	0	...	0
Module C	1	1	0	...	1
....	...	...	...	...	...
Module X	0	0	0	...	0

4. Call graph based ranking algorithm

Call graphs can be used in tracing the software code module by module. Specifically, each node in the call graph represents a procedure and each edge (a, b) indicates that procedure "a" calls procedure "b". Call graphs can be used to better understand the program behavior, i.e. the data flow between procedures. The advantages of call graphs are simply locating procedures that are rarely or frequently called.

In our research we have inspired from the search capability on the web. Most search engines on the web use the PageRank algorithm that is developed by Page and Brin [6]. The PageRank algorithm computes the most relevant results of a search by ranking web pages. We have adopted this ranking methodology to software modules (Figure 1). We hypothesize that more frequently used modules and less used modules should have different defect characteristics. For instance, if a module is frequently used and the developers/ testers are aware of that, they will be more careful in implementing/ testing that module. On the contrary, less used modules may not reveal their defective nature as they are not used frequently and existing defects can only be detected with thorough testing.

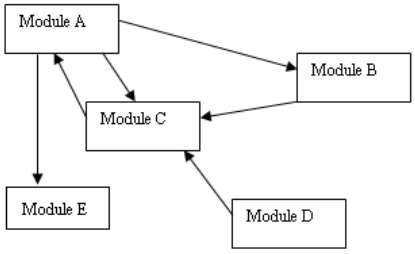

Figure 1. Simple Example

The PageRank theory holds that an imaginary surfer, who is randomly clicking on links, will continue surfing with a certain probability. This probability, at any step, that the person will continue is defined as a damping factor d. The damping factor is in the range $0 < d < 1$, and usually set to 0.85 [4, 6].

In our case, we do not necessarily take the damping factor as a constant, rather we dynamically calculate it for each project. The web surfer analogy corresponds to calling other software modules in our context. Therefore, we define the damping factor, d, of software with N modules, as the ratio of actual module calls to the all possible module calls.

We used the same algorithm of the PageRank. The modules resemble web pages, and calling relationships between the modules resemble the linking relationship between the web pages. We assign equal initial values (i.e. 1) to all modules and iteratively calculate module ranks. We named the results as the Call Graph Based Ranking (CGBR) values. CGBR framework gives ranks to all modules. The formula of the CGBR is given in Equation 1.

$$CGBR(A) = (1-d) + d * \sum_{i} \frac{CGBR(Ti)}{C(Ti)} \qquad (1)$$

,where CGBR(A) is the call graph based rank of module A, CGBR(Ti) is the call graph based rank of module Ti which call for module A, C(Ti) is the number of outbound calls of module Ti and d is the damping factor. Usually after 20 iterations the CGBR values converge [4]. The pseudo code of the CGBR

computation adopted from PageRank is given in Figure 2.

```
N = The number of all modules
damp = (The number of the called modules) / (N * N)
Load Call Graph Matrix
I = The number of the iteration
M = K = L = The size of the call graph matrix

for L times
        CGBR(L) = 1          # Assigned initial values
End

for I tiems
   for M times
      for K times
         if call graph matrix (M, K) = 1
            summation = summation + CGBR(K) / (summation of the call graph matrix column(K))
         end
      end
      CGBR(M) = (1 - damp) + damp * summation
      summation = 0
   end
end
```

Figure 2. The pseudo code of the CGBR computation.

Considering the simple example for call graph based ranking algorithm (recall Figure 1), there are 5 modules that interact with each other.

Table 3. Call graph matrix of the example

Caller \ Called	A	B	C	D	E
A	0	1	1	0	1
B	0	0	1	0	0
C	1	0	0	0	0
D	0	0	1	0	0
E	0	0	0	0	0

For example, module A calls three other modules, which are modules B, C and E. The first row in Table 3 shows the modules that are called by module A. These calls are represented with the numeric symbol 1. In addition, if we look at the first column in Table 3, we can say that module A is only called from module C. Also module D is dead code unless it is a main routine, since it is not called from any other modules. For this simple example damping factor is calculated as df = 6/(5*5), since there are a total of 6 module calls in this software. It can be seen in Figure 2 that, module C, which has the highest number of incoming calls, will have the greatest call graph based ranking value.

5. Experiments

5.1. Design

In our experiments, we left one of the projects as the test case and built predictors on the remaining two projects. We repeat this for all three datasets. We use a random 90% of the training set for the models and repeat this procedure 20 times in order to overcome ordering effects [16, 17].

Before we trained our model, we have applied log-filtering on the datasets and normalized our CGBR values. We classified our CGBR values in 10 bins and give each bin a weight value from 0.1 to 1. Then we have adjusted the static code attributes by multiplying each row in the data table with corresponding weights.

Note that we use sample weighting rather than attribute weighting. Attribute weighting corresponds to assigning weight to the columns of the data table, which we have investigated elsewhere [28]. Furthermore, it is a special case of feature subset selection, where informative features are assigned either 0 or 1 weight values. However, the purpose of this research is to locate software modules for assigning testing priority rather than deciding which attributes to use.

We have used a simple data miner that is Naïve Bayes [16, 17, 18]. Naive Bayes is implemented as described in Menzies et al. [16]. It is derived from Bayes' Theorem, where the prior probability and the evidence explain the posterior probability of an event.

$$P(H|E) = \frac{P(H)}{P(E)} \prod_i P(E_i|H) \qquad (2)$$

From Equation 2, we can denote the class label of a module, i.e. defective or defect-free as H. Evidences, Ei, are the attributes collected from the

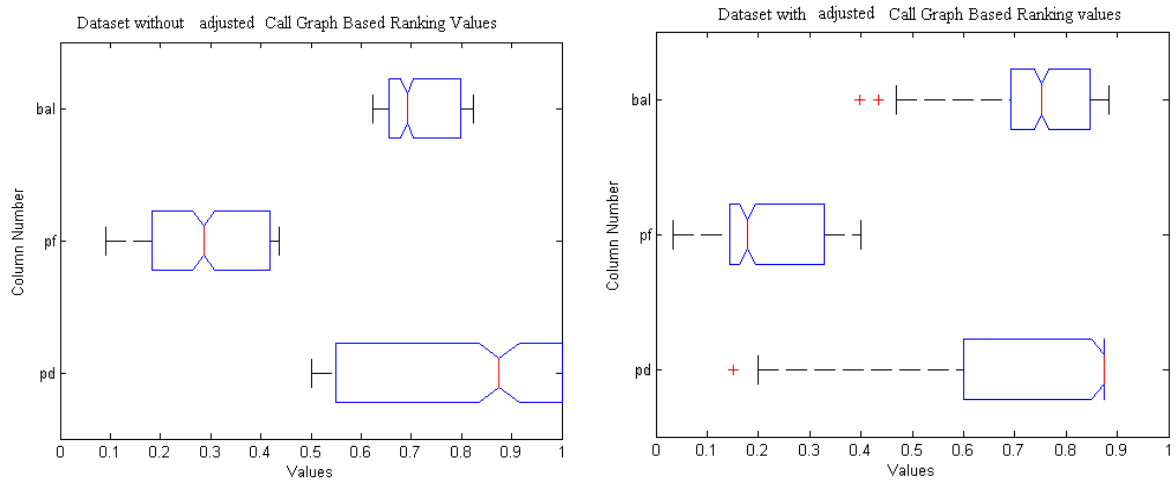

Figure 3. (pd, pf) and balance results with original and CGBR adjusted data

projects. In order to find the posterior probability of a new module given its attributes, we need to use the prior probabilities of two classes and the evidence from historical data. For numeric attributes with pairs of mean and standard deviation (μ, σ), the probability of each attribute given H is calculated with a Gaussian density function [1] (Equation 3).

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (3)$$

In order to make a comparison of our approach with previous research, we use three different performance measures, which are probability of detection (pd), the probability of false alarm, and balance (bal) as defined in Menzies et al. [16]. To compute pd and pf, the confusion matrix in Table 4 is needed. Performance measures are calculated using Equations 4, 5 and 6 respectively.

Table 4. Confusion Matrix

Real \ Estimated	Defective	Nondefective
Defective	A	C
Nondefective	B	D

$$pd = A / (A + C) \quad (4)$$

$$pf = B / (B + D) \quad (5)$$

$$balance = 1 - \frac{\sqrt{(0-pf)^2 + (1-pd)^2}}{\sqrt{2}} \quad (6)$$

In the perfect scenario, predictors should catch all the defects and never mark a defective-free module as defective. To provide this scenario pf should be a low value and pd should be a high value. Balance is a third performance measure and is used to choose the optimal (pd, pf) pairs that are closest to the perfect scenario.

5.2. Results from the experiments

Figure 3 shows (pd, pf, bal) results for the original attributes (i.e. without CGBR adjustment) and the dataset with CGBR adjustment respectively. These figures show the box-plots of the (pd, pf) and balance results. The leftmost and the rightmost lines in the boxes correspond to the 25% and 75% quartiles and the line in the middle of the box is the median. The notches around the median correspond to the 95% confidence interval of the median for paired t-test. The dashed lines in the box-plots indicate 1.5 times of the inter-quartile range, i.e. the distance between the 25% and 75% quartiles. Data points outside these lines are considered as outliers.

Figure 3 shows that using CGBR adjusted data slightly increases the median probability of detection, and significantly decreases the median probability of false alarms. Moreover, CGBR's pd results are spread in a narrower interval, from which we can conclude that CGBR produces more stable results than standard approach. Note that 50% of the results yield detection rates over 88% and false alarms below 18%. Finally the balance performance measure significantly improves.

We conducted another experiment to reflect the practical usage of the model. In the new experiment we use all available data in the training projects and test the performance of the model on the third project. Table 5 shows averaged values over all projects for the

median (pd, pf) rates of the two models and the required testing effort in terms of LOC [24]. It also includes the estimated LOC for random strategy depending on the pd rates 79% and 77% respectively. In order to detect 79% of the actual defects in the code, the estimated LOC for inspection with random strategy is 13,866 and the estimated LOC for original data is 12,513. Therefore, using Naïve Bayes predictor with original data achieves an improvement of 10% in testing effort.

Conversely for 77% defect detection rate of the CGBR framework, the estimated LOC for random strategy is 13,515 and the required LOC for CGBR adjusted data is 10439. The false alarm rate has also changed from 30% to 20%. Hence CGBR framework decreases the probability of false alarms and testing effort, where the improvement is 23%.

Table 5. Analysis of the second experiment.

	Naive Bayes with Original Data	Naive Bayes with CGBR Adjusted Data
pd	79%	77%
pf	30%	20%
Estimated LOC for inspection	12513	10439
Required LOC for inspection (Random approach)	13866	13515
Improvement on testing effort	10%	23%

The results for individual projects also validate our findings. Table 6 shows the (pd, pf) rates with CGBR adjusted data and with the original data. For all projects we observe the same pattern: pd rates do not change significantly and pf rates decrease for all projects. For AR5, pf rate is decreased with the cost of missing some defective modules. However, AR5 is a small dataset with only 8 defective modules out of a total of 36 projects. Therefore CGBR adjusted data model miss only a single defective module compared to the original data model.

We explain our observations as follows: Adjusting dataset with CGBR values provide additional information content compared to pure static code attributes. The adjustment incorporates inter module information by using architectural structure, in addition to the intra module information available in static code attributes.

Table 6. (pd, pf) rates for all projects with CGBR adjusted and original data

		AR3	AR4	AR5	Avg.
CGBR	pd	88	55	88	77
	pf	33	10	18	20
Orig.	pd	86	50	100	79
	pf	42	18	29	30

5.3. Discussion of the experiments

The benefits of using datasets adjusted with CGBR values are very important and practical. Low pf rates make the predictors practical to use. The reason is that low pf rates keep the amount of code to be manually inspected at a minimum rate. As pf rates increase, the predictors become impractical, even not much better than random testing strategies. Our proposed approach allows decreasing the pf rates, thus it is practical to use.

Note that CGBR adjusted model locates equivalent number of defects with the standard model, while giving fewer false alarms. Our results show that, if testers were to use the predictions from the CGBR adjusted model, they would spend 13% less testing effort than the original Naïve Bayes model and 23% less than random testing.

In addition to the discussion of results, we investigated the correlation between the CGBR values and the defect content of the modules. We used Spearman's correlation coefficient that indicates a negative correlation with -0.149. This means that there are relatively fewer defects when the modules' CGBR values are high. That is what we have speculated in the first place. If a module is called frequently from other modules, any defect in this module is recognized and corrected in a short time by the developers. On the other hand, the modules which are not called by other modules will be more defective because these types of modules are not used frequently. Thus, developers are not as much aware as they do for frequently used modules. As a result the defects can easily hide in less used modules.

6. Threats to validity

Public data sets do not contain call graph matrices [22]. This is a challenge but we overcome this with using datasets of a company that we had source code access. Therefore, this study should be externally validated with other industrial and open source projects of different sizes. Although open source projects allow access to code, matching defects at the module level and obtaining call graphs are difficult tasks. For this research, we have implemented a tool for constructing

static call graphs from ANCI C source codes and manually matched defects to modules.

Furthermore, the datasets we have used are implemented in a procedural language, i.e. C, and we are well aware that the same study should be repeated with object oriented projects, where we suspect that these effects would be more visible.

7. Conclusion

In this research, we propose a novel combination of the static code attributes (SCA) and architectural structure of software to make predictions about defect content of software modules. We model intra module properties by using SCA. Further, we model inter module relations by weighting software modules, where the weights are derived from static call graphs using the well-known PageRank algorithm.

Our results are strongly consistent with recent discussions by Menzies et.al. They state that SCA have limited information content and further research direction should focus on increasing the content rather than using new models for defect prediction [18]. They combine requirement level metrics with SCA, whereas we combine inter module relations with SCA in order to increase the information content in software data. We achieve this by modeling inter module relations with call graph based ranking (CGBR) values. Our results show that data adjusted with CGBR values do not change the median probability of detection, and significantly decrease the median probability of false alarm.

Proposed CGBR framework has important practical aspects. Defect predictors are helpful when they guide testers to the software modules that should be manually inspected. Therefore, defect predictors should detect as much defect as possible, while firing low false alarms. False alarms cause waste of testing effort since they unnecessarily mark a defect free module as defective. CGBR framework's detection performance is not worse than state-of-the-art defect predictors. On the other hand, it has significantly lower false alarm rates.

We should also note that CGBR is a framework rather than a model. In this research we have used it with Naive Bayes data miner, however it can be safely applied to any model that is possibly being used in practice.

Another contribution of this research is easing the call graph analysis. Call graphs are visually inspected in general. For large systems, visual inspections of call graphs become difficult. CGBR framework allows automated analysis of identifying heavily or scarcely used modules.

Our future work consists of replicating this study on projects developed in object-oriented languages. We also plan to expand the number of data miners, to show the applicability of CGBR framework.

8. Acknowledgements

This research is supported by Boğaziçi University research fund under grant number BAP 06HA104 and the Turkish Scientific Research Council (TUBITAK) under grant number EEEAG 108E014.

9. References

[1] Alpaydin, E., *Introduction to Machine Learning*, The MIT Press, 2004.

[2] Arisholm, E. and Briand, C. L., "Predicting Fault-prone Components in a Java Legacy System", *ISESE*'06, 2006.

[3] Basili, V. R., Briand, L. C., and Melo, W. L., "A Validation of Object-Oriented Design Metrics as Quality Indicators", *Software Engineering*, vol. 22, 1996, pp. 751-761.

[4] Benincasa, C., Calden, A., Hanlon, E., Kindzerske, M., Law, K., Lam, E., Rhoades, J., Roy, I., Satz, M., Valentine, E., and Whitaker, N., "Page Rank Algorithm", Department of Mathematics and Statics, University of Massachusetts, Amherst, Research, May, 2006.

[5] Boetticher, G., Menzies, T. and Ostrand, T., PROMISE Repository of empirical software engineering data http://promisedata.org/repository, West Virginia University, Department of Computer Science, 2007.

[6] Brin, S. and Page, L., "The anatomy of a large-scale hypertextual search engine", *Computer Networks and ISDN Systems*, 1998,pp. 107-117.

[7] Fenton, N. E., "A Critique of Software Defect Prediction Models", *IEEE Transaction on Software Engineering* , vol.25, no. 5, 1999.

[8] Hall, M. W. and Kennedy, K., "Efficient Call Graph Analysis", *LOPLAS*, vol.1, issue 3, 1992, p. 227-242.

[9] Harrold, M. J. , "Testing: a roadmap", In Proceedings of the Conference on the Future of Software Engineering, ACM Press, New York, NY, 2000, pp. 61-72.

[10] Henry S., and Kafura D., "Software structure metrics based on information flow", *IEEE Trans. on Software Engineering*, 7(5):510 518, September, 1981.

[11] Khoshgoftaar T. M., and Seliya N., "Fault Prediction Modeling for Software Quality Estimation: Comparing Commonly Used Techniques", *Empirical Software Engineering,* vol. 8, no. 3, 2003, pp. 255-283.

[12] Kitchenham, B. A., Pickard, L. M., and Linkman, S. J., "An evaluation of some design metrics", Software Engineering Journal archive, vol. 5, Issue 1, 1990, pp. 50-58.

[13] Koru, A. G. and Liu, H., "Building effective defect-prediction models in practice Software", *IEEE,* vol. 22, Issue 6, Nov.-Dec. 2005, pp. 23 – 29.

[14] Koru, A. G. and Liu, H., "An Investigation of the Effect of Module Size on Defect Prediction Using Static Measures." *Proceeding of PROMISE 2005*, St. Louis, Missouri, 2005, pp. 1-6.

[15] Malaiya, Y. K., and Denton, J., "Module Size Distribution and Defect Density", *ISSRE 2000*, 2000, pp. 62-71.

[16] Menzies, T., Greenwald, J. and Frank, A., "Data Mining Static Code Attributes to Learn Defect Predictors", *IEEE Transactions on Software Engineering*, vol. 33, no.1, 2007, pp. 2-13.

[17] Menzies, T., Turhan, B., Bener, A., and Distefano, J., "Cross- vs within-company defect prediction studies", Technical report, Computer Science, West Virginia University, 2007.

[18] Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B., Jiang, Y., "Implications of Ceiling Effects in Defect Predictors", to appear in PROMISE Workshop in ICSE 2008, Leipzig, Germany, May 2008

[19] Munson, J. and Khoshgoftaar, T. M., "Regression modelling of software quality: empirical investigation", *Journal of Electronic Materials*, vol. 19, no. 6, 1990, pp. 106-114.

[20] Munson, J. and Khoshgoftaar, T. M., "The Detection of Fault-Prone Programs", *IEEE Trans. on Software Eng.*, vol. 18, no. 5, 1992, pp. 423-433.

[21] Nagappan, N. and Ball, T., "Explaining Failures Using Software Dependencies and Churn Metrics", *In Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement,* Madrid, Spain.

[22] Nasa/Wvu IV&V Facility, Metrics Data Program, available from http://mdp.ivv.nasa.gov; Internet.

[23] Oral, A.D. and Bener, A., 2007, Defect Prediction for Embedded Software, *ISCIS 2007,* 7-9 November, Ankara, Turkey.

[24] Shepperd, M. and Ince, D., "A Critique of Three Metrics," *J. Systems and Software*, vol.26, no. 3, pp.197-210, Sept. 1994

[25] Tahat, B. V. , Korel, B. and Bader, A., "Requirement-Based Automated Black-Box Test Generation", In Proceedings of 25th Annual International Computer Software and Applications Conference, Chicago, Illinois, 2001, pp. 489-495.

[26] Tosun, A., Turhan, B., Bener, A., "Ensemble of Defect Predictors: An Industrial Application in Embedded Systems Domain", Technical Report, Computer Engineering, Bogazici University, 2008.

[27] Turhan, B., and Bener, A., "A multivariate analysis of static code attributes for defect prediction", *In Proceedings of the Seventh International Conference on Quality Software,* Los Alamitos, CA, USA, 11-12 October, IEEE Society, , 2007, pp. 231-237.

[28] Turhan, B., and Bener, A., "Software Defect Prediction: Heuristics for Weighted Naïve Bayes", *ICSOFT 2007*, Barcelona, Spain, 2007, pp. 22-25.

[29] Turhan, B., Bener, A., "Data Sampling for Cross Company Defect Predictors", Technical Report, Computer Engineering, Bogazici University, 2008.

[30] Zimmermann, T. and Nagappan, N., "Predicting Subsystem Failures using Dependency Graph Complexities," *The 18th IEEE International Symposium on Software Reliability (ISSRE '07),* 2007, pp. 227-236.