Early Estimation of Defect Density Using an In-Process Haskell Metrics Model

Mark Sherriff[1], Nachiappan Nagappan[2], Laurie Williams[1], Mladen Vouk[1]

[1] North Carolina State University, Raleigh, NC 27695
{mssherri, lawilli3, vouk}@ncsu.edu
[2] Microsoft Research, Redmond, WA 98052
nachin@microsoft.com

ABSTRACT

Early estimation of defect density of a product is an important step towards the remediation of the problem associated with affordably guiding corrective actions in the software development process. This paper presents a suite of in-process metrics that leverages the software testing effort to create a defect density prediction model for use throughout the software development process. A case study conducted with Galois Connections, Inc. in a Haskell programming environment indicates that the resulting defect density prediction is indicative of the actual system defect density.

Categories and Subject Descriptors

D.2.8 **[Software Engineering]:** Metrics - *Performance measures*, *Process metrics*, *Product metrics.*

General Terms

Measurement, Reliability.

Keywords

Empirical software engineering, multiple regression, software quality, Haskell.

1. INTRODUCTION

In industry, actual defect density of a software system cannot be measured until it has been released in the field and has been used extensively by the end user. Actual defect density information as found by the end users becomes available too late in the software lifecycle to affordably guide corrective actions to software quality. It is significantly more expensive to correct software defects once they have reached the end user compared with earlier in the development process [3].

Software developers can benefit from an early warning of defect density. This early warning can be built from a collection of internal, in-process metrics that are correlated with actual defect density, an external measure. The ISO/IEC standard [16] states that "internal metrics are of little value unless there is evidence that they are related to some externally visible quality." Some internal metrics, such as complexity metrics, have been shown to be useful as early indicators of externally-visible product quality [1] because they are related (in a statistically significant and stable way) to the field quality/reliability of the product. The validation of such internal metrics requires a convincing demonstration that (1) the metric measures what it purports to measure and (2) the metric is associated with an important external metric, such as field reliability, maintainability or fault-proneness [12].

*Our research **objective** is to construct and validate a set of easy-to-measure in-process metrics that can be used to create a prediction model of an external measure of system defect density.* To this end, we have created a metric suite we call the Software Testing and Reliability Early Warning metric (STREW) suite. Currently, there are two versions of STREW that have been developed to analyze an object-oriented language (STREW-Java or STREW-J) [20] and a functional programming language (STREW-Haskell or STREW-H) [23, 24].

In this paper, we present the results of an industrial case study designed to analyze the capabilities of the prediction model created by the STREW-H metrics suite. The project is an ASN.1 compiler created by Galois Connections, Inc., using the Haskell programming language. The remainder of the paper is organized as follows. Section 2 describes the background work, and Section 3 introduces the STREW metric suite. Section 4 discusses the industrial case study performed with the STREW-H metric suite. Section 5 presents our conclusions and future work.

2. BACKGROUND

In prior research, software metrics have been shown to be indicators of the quality of software products. Structural object-orientation (O-O) measurements, such as those in the Chidamber-Kemerer (CK) O-O metric suite [8], have been used to evaluate and predict fault-proneness [1, 5, 6]. These O-O metrics can be a useful early internal indicator of externally-visible product quality [1, 25, 26]. The CK metric suite consists of six metrics: weighted methods per class (WMC), coupling between objects (CBO), depth of inheritance tree (DIT), number of children (NOC), response for a class (RFC) and lack of cohesion among methods (LCOM).

Basili et al. [1] studied the fault-proneness in software programs using eight student projects. They observed that the WMC, CBO, DIT, NOC and RFC were correlated with defects while the LCOM was not correlated with defects. Further, Briand et al. [6] performed an industrial case study and observed the CBO, RFC, and LCOM to be associated with the fault-proneness of a class. A similar study done by Briand et al. [5] on eight student projects showed that classes with a higher WMC, CBO, DIT and RFC were more fault-prone while classes with more children (NOC) were less fault-prone. Tang et al. [26] studied three real time

A-MOST'05, May15–16, 2005, St. Louis, Missouri, USA.
Copyright 2005 ACM 1-59593-115-5/00/0004…$5.00.

systems for testing and maintenance defects. Higher WMC and RFC were found to be associated with fault-proneness.

Nikora and Munson [22] have shown that structural metrics can be indicators of defects in software. Structural metrics include measurements such as number of executable statements, number of nodes and edges in a flow graph, and total number of cycles in a flow graph. Nikora and Munson showed that when the values of these specific structural metrics increase, more defects were likely to have been introduced into the system. Also, Harrison et al. [14] used structural metrics to estimate the quality of functional programs in order to compare the relative quality of systems created using functional languages with those created using object-oriented programs. Harrison et al. demonstrated a statistically significant correlation between the structural metrics and software quality.

El Emam et al. [13] studied the effect of class size on fault-proneness by using a large telecommunications application. Class size was found to confound the effect of all the metrics on fault-proneness. Finally, Chidamber et al. [7] analyzed project productivity, rework, and design effort of three financial services applications. High CBO and low LCOM were associated with lower productivity, greater rework, and greater design effort.

Vouk and Tai [27] showed that in-process metrics have strong correlation with field quality of industrial software products. They demonstrated the use of software metric estimators, such as the number of field failures, failure intensity (indicated by failures per test case), and drivers such as change level, component usage, and effort in order to quantify component quality in terms of the number of failures; identify fault-prone and failure-prone components; and guide the software testing process.

To summarize, there is a growing body of empirical evidence that supports the theoretical validity of the use of these internal metrics [1, 5] as predictors of fault-proneness. The consistency of these findings varies with the programming language [25]. Therefore, the metrics are still open to criticism. [9]

2.1 HUnit and Test-Driven Development

HUnit[1] is an open source unit-testing framework that has been created for Haskell systems. The capabilities of HUnit parallel those of the award-winning Java unit testing framework, JUnit[2] [15]. HUnit provides different developers working on the same project a standard framework for creating unit tests and allows them to run all sets of tests created by the development team. STREW-H contains metrics that are gathered from the HUnit test cases.

The basic construct in HUnit is an assertion. A HUnit test case is an executable unit of code that contains one or more assertions. Assertions are functions that check to see if an actual result matches an expected result using keywords such as `assertEqual`, `assertTrue`, `assertBool`, and `assertFailure`. For example, assume a function `foo` that takes an integer x and returns a tuple (1,x). The following HUnit code could be used to test a single case of this function:

```
test1 = TestCase (assertEqual "for (foo 2),"
(1,2) (foo 2))
```

[1] http://hunit.sourceforge.net/

[2] http://www.junit.org

This says that with test case name `"for (foo 2),"` verify that `(foo 2)` returns `(1,2)`. However, assume that this function has an defect in it and returns `(1,3)` instead. The following would be reported by the HUnit module:

```
### Failure in: 0:test1
  for (foo 2),
  expected: (1,2)
  but got: (1,3)
  Cases: 1  Tried: 1  Failures: 1
```

Suites of test cases can be created for various functions in a system and can be run in batch quickly and easily. Automated HUnit suites can be run often (e.g. at least once per day) as regression tests to check whether new functionality has broken previously-working functionality.

By creating and grouping together various sets of test cases, HUnit allows a developer to utilize a test-driven development (TDD) [2] practice. With TDD, before implementing production code, the developer writes automated unit test cases for the new functionality they are about to implement. After writing test cases, the developers produce code to pass these test cases. The process is essentially “opportunistic” in nature [11]. A developer writes a few test cases, implements the code, writes a few test cases, implements the code, and so on. The work is kept within the developer’s intellectual bounds because he or she is continuously making small design and implementation decisions and increasing the functionality at a manageable rate. New functionality is not considered properly implemented unless these new (unit) test cases, and every other unit test case written for the code base, run properly. Williams et al. performed a case study with a team at IBM that transitioned from ad hoc unit testing to TDD for a Java project [28]. The team experienced a 40% reduction in defect density of new/changed code once the method was adopted.

2.2 QuickCheck

Much like HUnit, QuickCheck[3] is a testing tool created specifically for Haskell systems. STREW-H also contains metrics that are gathered from the QuickCheck test cases. The purpose of QuickCheck is to find user-defined properties within a Haskell program and generate multiple random test cases for each property. QuickCheck was created to directly exploit an advantage that functional programs possess [10]. Most of the Haskell code in a system consists of pure functions as opposed to functions that produce side effects. This property enables testing without concern for the state of the program, allowing developers to test individual functions easily. In QuickCheck, a developer can define a property, which indicates a truism about a function. For example, the function `(reverse [x] = [x])` indicates that if a list of one element is passed to the function `reverse`, the list is returned unchanged. A property for this function can be defined as:

```
(prop_reverse x = reverse [x] == [x])
```

This property shows that for all values of x passed to the function reverse, the left side should always equal the right side for a list of one element.

[3] http://www.cs.chalmers.se/~rjmh/QuickCheck/

Also consider the example function from Section 2.1. This HUnit code easily tested one case of the function `foo`. However, this function would be a good candidate for testing under QuickCheck, since it is likely that a developer would want to test this function for numerous test cases simultaneously. For this function, a property can be defined as:

```
(prop_foo x = (foo x) == (1,x))
```

When activated, QuickCheck will scan through source code looking for these defined properties and will generate random values for the parameters (x, in this case) to test the function. The default is to run 100 values for x (random, not necessarily unique) per each property. However, QuickCheck provides a great deal of control over the testing as well, allowing developers to define the number of test cases to be run, along with ranges for proper values for random testing.

3. STREW BACKGROUND

The STREW metric suite is a set of internal, in-process software metrics that are leveraged to make an early estimation of defect density and its associated confidence interval. Prior studies [1, 5-7, 13, 14, 22, 25-27] have leveraged the structural aspects of the code, but not metrics associated with the testing effort, to make an estimate of defect density.

The STREW metric suites consist of measures of the thoroughness of white-box testing and of some structural aspects of the implementation code. The metrics are intended to cross-check each other and to triangulate upon a defect density estimate. For example, one developer may write fewer test cases, each with multiple assertions checking various conditions. Another developer might test the same conditions by writing many more test cases, each with only one assertion. We intend for our metric suite to provide useful guidance to each of these developers without prescribing the style of writing the system code or test cases.

The use of the STREW metrics is predicated on the existence of an extensive suite of automated unit test cases being created as development proceeds, such as is done with HUnit and QuickCheck. STREW leverages the utility of automated test suites by providing a defect density estimate. The defect density estimate relative to historical data is calculated using multiple linear regression analysis which is used to model the relationship between software quality and selected software metrics [17, 19].

Current research involves the refinement of the language-dependant STREW-J [20] and STREW-H metric suites. Metrics will be added and deleted from the suites based on case studies and validation efforts using various analysis techniques, such as multiple linear regression analysis, Bayesian analysis, and principal component analysis (PCA). Ultimately, the metric suites will be comprised of a minimal set of metrics necessary to provide an accurate defect density estimate while not imposing undue overhead.

4. STREW-H

We utilized the STREW-J [20] metric suite as our starting point to create the STREW-H metric suite. Metrics were eliminated that were not applicable for functional languages and additions were made based upon a review of the literature and upon expert opinion, as will be discussed. The metrics taken directly from the STREW-J, however, were changed somewhat to be used with the Haskell language. For example, assertions take on a slightly different meaning in Haskell. The STREW-J's assertions are calculated by counting the assert keywords in JUnit testing cases. The assertion/testing metric was changed to include the number of QuickCheck properties and HUnit asserts that are checked. This "number of test cases" metric effectively gives the same measure of individual testing statements for STREW-H as the JUnit asserts does for STREW-J.

Interviews were conducted with 12 Haskell researchers at Galois Connections, Inc.[4] and with members of the Programatica team[5] at The OGI School of Science & Engineering at OHSU (OGI/OHSU) to elicit suggestions for the initial version of the STREW-H metric suite. The suggestions included metrics that measured structural and testing concepts that are unique to functional languages.

One of these structural and testing concepts that is unique to functional programming is monadic code. Monadic code is effectively a written "building block" of code. Programmers can put these "building blocks" together in an ordered way to create sequences of computations. Haskell programs incorporate monads to implement imperative functions [21]. Monads allow a system written in a functional language, such as Haskell, to perform tasks that require external information, such as user input. Monads also allow Haskell developers to use algorithms that work more efficiently in an imperative environment. Anecdotal evidence from Galois and OGI/OHSU suggests that once the system switches over into monadic code, there is a higher likelihood of programmer error.

Another of the proposed metrics is concerned with warnings that are reported by the Glasgow Haskell Compiler [18], one of the most commonly used Haskell compilers. These warnings are indicators of potential defects in the system. For instance, an "incomplete pattern" warning indicates that there are base cases not covered by a recursive function and could possibly produce a program failure. These warnings are not necessarily defects in the system, but could be an indication of programmer error [18].

A STREW-H feasibility study [23] was performed using a large (200+ KLOC), open-source Haskell project. This study used the initial version of STREW-H, Version 0.1. This study showed a strong relationship between the STREW-H metric suite and system defects and motivated further study of STREW-H in-process metrics. Our most recent findings from an industrial case study regarding the STREW-H suite is presented in this paper and build upon concepts from our feasibility study.

From this feasibility study, we analyzed what metrics contributed the most to the prediction model and how some metrics were related to each other. This analysis prompted us to refine our metric suite to five metrics that cover both testing information and coding standards specific to the Haskell programming language.

We thus propose the following five candidate metrics for the STREW-H Version 0.2:

- *test lines of code / source lines of code (M1)* includes HUnit, QuickCheck, and ad host tests and shows the

[4] http://www.galois.com

[5] http://www.cse.ogi.edu/PacSoft/projects/programatica/

general testing effort with respect to the size of the system;

- *number of type signatures / number of methods in the system (M2)* shows the ratio of methods that utilize type signatures, which is a good programming practice in Haskell;
- *number of test cases / number of requirements (M3)* includes HUnit, QuickCheck, and ad host tests and shows the general testing effort with regard to the scope of the system;
- *pattern warnings / KLOC (M4)* shows some of the most common errors in Haskell programming and includes common errors in the creation of recursive methods, such as incomplete patterns and overlapping patterns;
- *monadic instances of code / KLOC (M5)*, identified as a source of problems in Haskell programs, provides information on likely problem areas;

The STREW-H metric suite will be refined through additional research.

4.1 Case Study Description

We worked with Galois during the seven-month development of an ASN.1 compiler system. The project consisted of developing a proof-of-concept ASN.1 compiler (about 20 KLOC) that could show that high-assurance, high-reliability software could be created using a functional language.

During the course of the project, 20 in-process snapshots of the system were taken at one- or two-week intervals over the seven-month period. Also, logs were kept on the code base, indicating when changes needed to be made to the code to rectify defects that were discovered during the development process.

Galois' development methodology could best be considered a waterfall process, in which individual components of the ASN.1 compiler were designed, built, tested, and then integrated over the course of the project. The team was small (three developers), so all were involved in all aspects of design, development, and testing and were knowledgeable about the code base. Due to the aspects of this development methodology, all developers were actively discovering and correcting defects during the coding and integration process.

Two main tools were used to gather the various metrics on the in-process snapshots of the ASN.1 compiler. Regular expression tools, such as PowerGREP[6], were used to identify and count keywords in both code and compile logs to gather most of the metrics. SLOCCounter[7] was used to count source and test lines of code.

4.2 Case Study Limitations

We counted in-process defects by manually reading Galois's logs, identifying instances where code had been checked into the code base, run, and at some point determined to be incorrect. Due to the nature of the logs, we could thus only count when defects were discovered and corrected, not when defects were introduced into the system. Also, it is impossible to classify the types of defects that were logged, because the descriptions in the log were often too vague to gather any information other than the fact a bug was discovered and corrected.

Another limitation of the study is that the snapshots were taken relatively close together. That is, there is not a great deal of time between the snapshots that we analyzed. Due to the nature of the project and the length of time involved (seven months), it was decided that more data points were desired despite their proximity in time. This is a possible explanation for why the data does not change greatly from snapshot to snapshot. Additionally, the changes in the projects across the snapshots may not have been not large enough to avoid multi-collinearity. We tried to negate this effect to an extent by employing a data splitting technique as detailed below.

Finally, we propose that this method can and should be performed in-process in order to allow developers to utilize feedback from the method to help guide their development practices. However, for the purposes of validating the study, our analysis of the ASN.1 project was performed post-mortem in order to ascertain the method's ability to predict defect density with as little bias introduced as possible. The development team was aware that the code would be used in this study, but did not know how or to what extent.

4.3 Experimental Results

A multiple regression prediction model[8] was created with the five metrics of the STREW-H and the number of in-process defects that were corrected and logged in the versioning system by Galois. We created the multiple regression model with 14 random in-process snapshots of the system and used this model to predict the remaining six snapshots' defect densities. This data splitting was done five separate times with a different randomly-chosen set of 14 snapshots to help remove any bias. The analysis showed that future defect densities in the system could be predicted based on this historical data of the project with respect to the STREW-H metric suite.

The R^2 values from the five models were as follows:

- 0.943 (F=26.304, $p<0.0005$);
- 0.930 (F=21.232, $p<0.0005$);
- 0.962 (F=25.206, $p<0.0005$);
- 0.949, (F=29.974, $p<0.0005$); and
- 0.967 (F=42.237, $p<0.0005$).

We conservatively used the model with the lowest R^2 value for the prediction model in this case study. The results of the regression model showed that the STREW-H metrics are associated with the number of defects that were discovered and corrected during the development process. Using the generated regression model, the defect density of the remaining six snapshots was predicted to be as shown in Figure 1. The graph in Figure 1 shows the closeness of the fit of the regression equation. This regression model predicted the number of defects within an acceptable range that should be discovered in relation to the STREW-H metric values of a system. Five of the six predictions were within .3

[6] http://www.powergrep.com/

[7] http://www.dwheeler.com/sloccount/

[8] SPSS was used for the purpose of statistical analysis. SPSS does not provide statistical significance beyond three decimal places. p=0.000 is interpreted as $p<0.0005$.

defects/KLOC, but one prediction was .8 defects/KLOC away from the actual value. We believe this occurred for this particular snapshot because it occurred during a time when new code development dropped and test code development increased. This change in their general weekly development methodology contributed to this outlier.

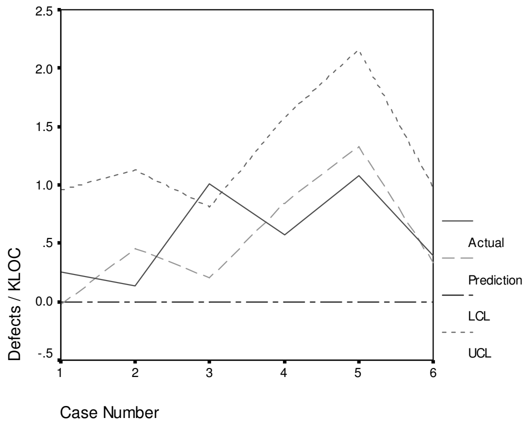

Figure 1: Actual vs. Estimated Defects/KLOC

One way to analyze which metrics are the most important would be a PCA. When PCA is performed, Kaiser-Meyer-Olkin coefficient (KMO) is used as a measure of sampling adequacy. KMO values of 0.6 or better are an indication that PCA is an acceptable technique to use with the data set [4]. The KMO measure of sampling adequacy for this data set, however, was not greater than the acceptable value of 0.6 at 0.522 to perform PCA. One possibility for this is that some of the measures, such as the number of requirements, remained constant through multiple snapshots. This statistical indication may show that while the combination of the metrics in the regression equation is indicative, the metrics individually cannot be proven to be direct indicators of defect density.

From this initial PCA analysis, however, we can gain some information as to which aspects of the STREW-H suite contribute the most to the prediction of defects. For example, pattern warnings/KLOC was identified as a primary contributor to the model in this experiment using factor analysis. Pattern errors include two common Haskell programming errors, overlapping patterns and incomplete patterns. An overlapping pattern is a condition where a certain case of a recursive function cannot be reached due to the fact that another function is called instead. This is a similar problem to a variable reference in local scope being called instead of one in the global scope – one superseded the other. Incomplete patterns occur when some base cases of a recursive function are not covered by a given function. This finding is consistent with what we discovered during our developer interviews during the creation of STREW-H.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have reported on a metric suite for providing an early warning regarding system defect density for Haskell programs. The STREW-H metric suite has been found to be a practical early indicator of defect density. A case study involving an industrial project indicates the promise of this approach. The metric suite is intended to be easy to gather in the development environment so that developers can receive an early indication of system defect density throughout development. This indication allows developers to take corrective actions earlier in the development process.

We will continue to refine the metric suite by add/deleting new metrics based on the results of further studies. Furthermore, we also will assess, in the context of our suite, the relationship between traditional software metrics, such as LOC and cyclomatic complexity and system reliability. Based upon these results, future versions of STREW-H may include these metrics. We will continue to validate the metric suite under different industrial and academic environments.

Future STREW-H case studies will involve the active use of the method during the development cycle. In this study, data was gathered post-mortem as to not introduce bias, but it is important to analyze how the STREW-H information can actively influence a development cycle. These future cases studies will involve teams actively coding and correcting bugs, calculating the STREW-H values, and then adjusting their development plans and techniques accordingly for the next development cycle.

During the validation process, we will also incorporate the STREW defect density estimation method into the Eclipse integrated development environment as a plug-in. Work has already begun on incorporating STREW-H into Eclipse [24]. This will enable the automatic gathering of the STREW metrics, thus enabling developers to utilize the STREW defect density estimation with little overhead for the software developer. The plug-in provides this estimation in the developer's programming environment when corrective action can be taken to correct defects when it is still economical.

6. AKNOWLEDGEMENTS

We would like to sincerely thank Galois Connections, Inc., and the Programatica Team at OGI/OHSU for their contributions to this work. We would also like to thank the NCSU Software Engineering Reading group for their comments on earlier drafts of this paper. This material is based upon work supported by the National Science Foundation under CAREER award Grant No. 0346903. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] Basili, V., Briand, L., Melo, W., "A Validation of Object Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 22, pp. 751 - 761, 1996.

[2] Beck, K., *Test Driven Development- by Example*. Boston: Addison-Wesley, 2003.

[3] Boehm, B. W., *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[4] Brace, N., Kemp, R., Snelgar, R., *SPSS for Psychologists*: Palgrave Macmillan, 2003.

[5] Briand, L. C., Wuest, J., Daly, J.W., Porter, D.V., "Exploring the Relationship between Design Measures and Software Quality in Object Oriented Systems," *Journal of Systems and Software*, vol. Vol. 51, pp. 245-273, 2000.

[6] Briand, L. C., Wuest, J., Ikonomovski, S., Lounis, H., "Investigating quality factors in object-oriented designs: an industrial case study," ICSE, 1999.

[7] Chidamber, S. R., Darcy, D.P., Kemerer, C.F., "Managerial Use of Metrics for Object Oriented Software: An Exploratory Analysis," *IEEE Transactions on Software Engineering*, pp. 629-639, 1998.

[8] Chidamber, S. R., Kemerer, C.F., "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. Vol. 20, pp. 476 - 493, 1994.

[9] Churcher, N. I. and Shepperd, M. J., "Comments on 'A Metrics Suite for Object-Oriented Design'," *IEEE Transactions on Software Engineering*, vol. 21, pp. 263-5, 1995.

[10] Classen, K. and Hughes, J., "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," International Conference on Functional Programming, Montreal, Canada, 2000.

[11] Curtis, B., "Three Problems Overcome with Behavioral Models of the Software Development Process (Panel)," International Conference on Software Engineering, Pittsburgh, PA, 1989.

[12] El Emam, K., "A Methodology for Validating Software Product Metrics," National Research Council of Canada, Ottawa, Ontario, Canada NCR/ERC-1076, June 2000 June 2000.

[13] El Emam, K., Benlarbi, S., Goel, N., Rai, S.N., "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Transactions on Software Engineering*, vol. Vol. 27, pp. 630 - 650, 2001.

[14] Harrison, R., Samaraweera, L. G., Dobie, M. R., and Lewis, P. H., "Estimating the quality of functional programs: an empirical investigation," *Information and Software Technology*, vol. 37, pp. 701-707, 1995.

[15] Herrington, D., "HUnit User's Guide 1.0." Available Online. http://hunit.sourceforge.net/HUnit-1.0/Guide.html. 2002.

[16] ISO/IEC, "DIS 14598-1 Information Technology - Software Product Evaluation," 1996.

[17] Khoshgoftaar, T. M., Munson, J.C., Lanning, D.L., "A Comparative Study of Predictive Models for Program Changes During System Testing and Maintenance," International Conference on Software Maintenance, 1993.

[18] Marlow, S., "The Glasgow Haskell Compiler." Available Online. http://www.haskell.org/ghc. 2004.

[19] Munson, J. C., Khoshgoftaar,T.M., "Regression Modelling of Software quality: Empirical Investigation," *Information and Software Technology*, pp. 106-114, 1990.

[20] Nagappan, N., "A Software Testing and Reliability Early Warning (STREW) Metric Suite," in *Department of Computer Science*, vol. PhD. Raleigh, NC: North Carolina State University, 2005.

[21] Newbern, J., "All About Monads." Available Online. Web Page. http://www.nomaware.com/monads/html/. Aug. 22, 2004.

[22] Nikora, A. P. and Munson, J. C., "Understanding the Nature of Software Evolution," IEEE International Conference on Software Maintenance, Amsterdam, The Netherlands, 2003.

[23] Sherriff, M., Williams, L., and Vouk, M. A., "Using In-Process Metrics to Predict Defect Density in Haskell Programs," Fast Abstract, International Symposium on Software Reliability Engineering, St. Malo, France, 2004.

[24] Sherriff, M., Williams, L., "Tool Support For Estimating Software Reliability in Haskell Programs," Student Paper, IEEE International Symposium on Software Reliability Engineering, St. Malo, France, 2004.

[25] Subramanyam, R., Krishnan, M.S., "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Transactions on Software Engineering*, vol. Vol. 29, pp. 297 - 310, 2003.

[26] Tang, M.-H., Kao, M-H., Chen, M-H., "An empirical study on object-oriented metrics," Sixth International Software Metrics Symposium, 1999.

[27] Vouk, M. A., Tai, K.C., "Multi-Phase Coverage- and Risk-Based Software Reliability Modeling," CASCON '93, 1993.

[28] Williams, L., Maximillian, E.M., Vouk, M.A., "Test-Driven Development as a Defect-Reduction Practice.," International Symposium on Software Reliability Engineering, Denver, CO, 2003.