Preliminary Results On Using Static Analysis Tools For Software Inspection

Nachiappan Nagappan[1], Laurie Williams[1], John Hudepohl[2], Will Snipes[2], Mladen Vouk[1]
[1] Department of Computer Science, North Carolina State University, Raleigh, NC, USA
{nnagapp, lawilli3, vouk}@ncsu.edu
[2] Nortel Networks, Software Dependability Design (SWDD), Research Triangle Park, NC, USA
{hudepohl,wbsnipes}@nortelnetworks.com

Abstract

Software inspection has been shown to be an effective defect removal practice, leading to higher quality software with lower field failures. Automated software inspection tools are emerging for identifying a subset of defects in a less labor-intensive manner than manual inspection. This paper investigates the use of automated inspection for a large-scale industrial software system at Nortel Networks. We propose and utilize a defect classification scheme for enumerating the types of defects that can be identified by automated inspections. Additionally, we demonstrate that automated code inspection faults can be used as efficient predictors of field failures and are effective for identifying fault-prone modules.

1.0 Introduction

The use of software code inspections, design inspections, and requirements inspections, has been found to increase software quality and lower software development costs [9, 21]. Prior studies indicate that inspections can detect as little as 20% [4] to as much as 93% [8] of the total number of defects in an artifact. Based upon a literature survey, Briand et al. [2] report that, on average, software inspections find 57% of the defects in code and design documents. Inspections have been traditionally done manually with key members of the development and quality assurance teams.

Recently, commercial automated software inspection (ASI) tools and services have been emerging. ASI tools are also known as static analysis tools, coding standard analysis tools, and source code analysis tools. These tools produce error messages similar to those of a compiler. However, they identify additional faults, such as coding standard non-compliance, uncaught runtime exceptions, security vulnerabilities, redundant code, division by zero, and memory leaks. For illustration, Appendix A provides a comparison of compiler messages and of ASI messages (from the FlexeLint tool) for a short program. ASI tools are intended to identify faults which allow the software engineers to recode before they surface more publicly as manual inspections faults or as test and/or field failures. The creators of these tools profess that this early identification leads to more efficient defect removal.

However, ASI does not replace the manual inspection process. As will be discussed, manual inspections have been shown to identify a wider range of defect types than ASI. However, the removal of the defects found by the ASI can allow the labor-intensive manual reviews to be more efficient and to focus on more complex, functional and algorithmic defects.

In 2001, Nortel Networks began to use ASI tools and services in conjunction with manual inspection. While an economic analysis has not yet been completed, qualitatively Nortel believes that ASI is beneficial. As a result, Nortel has since inspected over 20 million lines of code with their manual and automated inspection best practice.

We analyzed the data from two successive releases of a large commercial software product which utilized ASI. *Our research objectives were to (1) develop and utilize an automated inspection fault classification scheme to identify what kind of defects can be surfaced via automated inspection; (2) assess the efficacy of automated inspection faults as a predictor of actual field failures; and (3) assess the efficacy of automated inspection faults in the identification of fault-prone modules.*

The paper is organized as follows. Section 2 discusses the background and related work. Section 3 outlines the implementation of ASI at Nortel. Section 4 illustrates the case study details, and Section 5 discusses the results. Section 6 presents a preliminary approach for validation. Finally, Section 7 and 8 present the conclusions and future work, respectively.

2.0 Background and related work

In this section, we discuss the prior work in software inspections, code churn, automated inspections and fault classification schemes.

2.1 Software inspections

Inspections are defined as a static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and

other problems [13]. Beginning with the landmark work of Michael Fagan in 1976 [9], structured review mechanisms, such as inspections, have been repeatedly shown to be effective means of finding work product defects early in the software development process [22]. Industrial data has shown that inspections are among the most effective of all verification and validation (V&V) activities, measured by the percentage of defects typically removed from a document via the technique [25]. Inspections also help to increase the development team's familiarity with the code. However, manual inspections are labor-power intensive, requiring preparation and the simultaneous attendance of all participants (or inspectors). The effectiveness of inspections is dependant upon satisfying many conditions, such as adequate preparation, readiness of the work product for review, high quality moderation, and cooperative interpersonal relationships [22]. The effectiveness of ASI is less dependant upon these human factors.

In prior research, manual software inspection defects have been used as predictors of the defects remaining in the product. Eick et al. [5] use the capture-recapture (CR) technique to estimate the number of faults remaining in design documents after formal review meetings. With CR, faults are intentionally injected into an artifact. The number of faults detected by each reviewer at a document review meeting is used as a measure of the effectiveness of the review. The three commonly-used statistical approaches to estimate the number of defects remaining in CR data is the maximum likelihood estimator (MLE) [27], jackknife estimator (JE), and the Chao's method [1]. However, the MLE and the JE have been shown to consistently underestimate the number of defects [5, 27, 28]. This is in disagreement with a study [24] in which these methods overestimates the number of defects by 10%. In Section 5, we propose a model for using ASI faults as a predictor of field failures.

2.2 Static analysis tools used for automated software inspection (ASI)

ASI can remove a subset of defects prior to manual inspection or test. ASI has the potential to be economical. The main goals of using automated inspection are the following:

1. to identify syntax and interface errors;
2. to identify potential for reducing code volume via redundant or unused code;
3. to enforce architectural standards;
4. to enforce coding standards;
5. to eliminate potential sources of error and inefficiency; and
6. to eliminate actual errors.

Automated software inspection (ASI) tools are predominantly static analysis tools, error-checking tools, coding standard analysis tools, and source code analysis tools. Static analysis tools have been used for identifying defects in software systems [6, 7, 14]. These tools produce error messages similar to those of a compiler. However, they identify additional potential faults, such as coding standard non-compliance, uncaught runtime exceptions, security vulnerabilities, redundant code, inappropriate use of variables, division by zero, and memory leaks. ASI tools can, therefore, enable software engineers to fix faults before they surface more publicly in manual inspections or as test and/or field failures. The creators of these tools profess that this early identification leads to more efficient defect removal.

An important issue with the use of ASI tools is the inevitability of significant amounts of false positives. False positives are when the tool identifies a fault but a deeper analysis of the context shows no problem. There can be as many as 50 false positives for each true fault found by automated inspection tools [23]. Often, ASI tools can be customized and filters can be established so that certain classes of defects are not reported. Additionally, ASI pre-screening subcontract services examine the identified defects and the product and will remove as many false positives as possible.

The following ASI tools and services (which are used with C and C++) are being used at Nortel. These tools are representative of the commercial ASI tools and services.

FlexeLint[1] checks C/C++ source code to detect errors, inconsistencies, non-portable constructs, and redundant code. FlexeLint is a Unix-based tool (akin to the Window-based PC-lint). The tool has more than 800 error messages; some of the common defects identified are shown in Appendix B.

Reasoning[2]'s **Illuma** is an automated inspection service that finds defects in C/C++ applications. An organization sends their code to Reasoning who conducts an ASI, removes false positives, and produces a report. Illuma identifies reliability defects that cause application crashes and data-corruption. Examples of the C/C++ error classes include: NULL pointer dereference; out of bounds array access; memory leaks; bad de-allocation; and uninitialized variables.

Klocwork[3] has two ASI tools. **inForce** performs its automated inspection of source code to supply metrics for identifying potential defects, security flaws, and code optimizations. **GateKeeper** analyzes the source code architecture strengths and weaknesses and provides assessment details on code quality, hidden defects, and maintainability costs. Types of defects identified include actual relationships between modules (as compared to intended relationships), code clusters (cyclic

[1] http://www.gimpel.com/html/products.htm
[2] http://www.reasoning.com/
[3] http://www.klocwork.com/

relationships), high-risk code files and functions, potential logical errors, and areas for improvement.

2.3 Code churn

Code churn is a measure of the number of changed lines of code (LOC) between two (not necessarily consecutive) releases of the code. Similar to the LOC metric [17], code churn an also be used as an indicator of quality factors, such as fault-proneness [16]. We take code churn and deleted lines of code into consideration in our analysis. The untouched LOC across versions would be more likely to continue to function as they have in the past. Therefore, code churn can be used as a measure of change and majority of the new defects (including automatic inspection found) will be due to the new/changed (churned) or deleted code. [16]

2.4 Fault classification schemes

The goal of defect analysis is to identify the root cause of defects and to initiate actions so that the source of defects is eliminated [3]. Defect classification schemes are concerned with removing the subjectivity of the classifier and creating categories that are distinct, i.e. orthogonal [15]. The goal of IBM's Orthogonal Defect Classification (ODC) [3] scheme is to categorize defects such that each defect type is associated with a specific stage of development. Each defect type is intended to point to the part of the development process that needs attention. The ODC has eight defect types: Function, Interface, Checking, Assignment, Timing/Serialization, Build/Package/Merge, Documentation, and Algorithm [3]. In this paper, we use the ODC and a new classification scheme to categorize defects identified via ASI.

3.0 ASI at Nortel

The Software Dependability Design (SWDD) group of Nortel Networks has the mission of developing and disseminating key strategic technologies and best practices across the company for the improvement of the software dependability of each product. To achieve this mission, Nortel Networks has ten best practices for software dependability which they call the "Ten Point Program." The Ten Point Program is a collection of best practices, tools, methodology standards, and design guidelines developed within Nortel Networks. One of the best practices in the Ten Point Program is Manual and Automated Inspection (MAIN).

At Nortel Networks, the transition to using ASI is done in two phases: start up and in-process. In the start up phase, the SWDD group works closely with the transitioning team to establish a clean baseline of the product. For this, the most recently released product (considered the N-1 release) undergoes ASI. Because it is likely that this first use of ASI with this product will yield an excessive amount of false positive faults, the ASI output is pre-screened by a sub-contracting service. The SWDD and development teams receive a post-screening report (from the sub-contracting team) and fix the true faults that have been identified, beginning with the most severe defects first. Once the true ASI defects have been fixed (considered to be the N release), the product undergoes ASI again. The N release is the "clean baseline" and the team is considered to have implemented the ASI best practice. New functionality is then developed upon this baseline (the N+1 release). For the N+1 release and beyond, only the changes are inspected.

Once the team has undergone the start up phase, ASI is an additional defect removal filter in the development process, as shown in Figure 1. Since the code volume that undergoes ASI is more manageable once the clean baseline has been established, ASI is often run by the developers without the involvement of a pre-screening sub-contractor. Developers can tune the tool to create filters to reduce false positives. Nortel has also developed centralized, in-house expertise in the use of ASI tools and in the screening of the defects. These experts have anecdotally be able to reduce the false positive rate to approximately 1%.

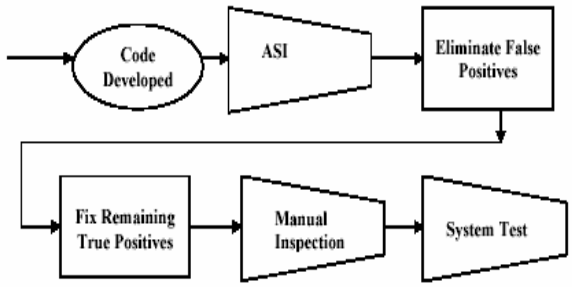

Figure 1: ASI in the development process

Depending upon the developer, the frequency of doing ASI varies. ASI can be run when a component is complete and is about to undergo manual inspection. Alternately, a developer can run the ASI tools more incrementally as code is being developed because the tools do not require a complete running product. In either case, manual inspection is also carried out after the true positives are fixed. In this way, the manual inspectors do not have to look for the defects that have already been identified by ASI.

4.0 Nortel Case study

An ASI case study was conducted on a large-scale telecommunications product developed at Nortel Networks. The 800 thousand lines of code (KLOC) product written in C, was chosen for analysis due to the component-specific data availability of the following:

- automated inspection faults from FlexeLint;
- actual field failure information;
- code churn; and
- deleted LOC information.

For this product, sub-contracting services were utilized to pre-screen the ASI defects. The defects that were analyzed were those that remained after the screening eliminate many false positives.

For the purpose of case study analysis, we used two consecutive software releases, henceforth called Release 1 and Release 2. The software system that was examined (both the releases) contains 61 components. The actual field failures were mapped to the component in which the defect was fixed. We use the following steps for analysis of the data elements (ASI faults, code churn, and the deleted lines of code):

- Classify the automated inspection faults to provide information on the types of defects that can be expected to be revealed by ASI.
- Examine relationships between the automated inspection faults, code churn, and the deleted lines of code with the actual field failures by computing a Spearman rank correlation.
- Build a multiple regression model to predict field failures using the data on automated inspection faults, code churn, and deleted lines of code from Release 1. Use this model with the results from Release 2 as a case study to examine the efficacy of the model as a predictor of field failures.
- Use discriminant analysis for the data from Release 1 to build a discriminant function for identifying fault-prone modules. Use this function with the results from Release 2 as a case study to examine its prediction accuracy for the identification of fault-prone modules.

Each of these will be discussed in Section 5.0.

5.0 Case study analysis and results

In Section 5.1 we present a classification scheme for automated inspection faults and the results of such a classification on Release 1 and Release 2. Section 5.2 provides the results of examining the relationship between the field failures and three metrics: automated inspection faults, code churn, and deleted lines of code. In Section 5.3, we present a two-fold approach toward building prediction models using automated inspection faults.

5.1 ASI defect classification

We classified the automated inspection faults from the FlexeLint tool found in both releases using the ODC [3], as shown in Figure 2. We mapped the FlexeLint defects that surfaced in the Nortel products to ODC; the details of the mapping between FlexeLint defects and ODC type are shown in Appendix B. The results indicate that ASI surfaces defects that fall into three ODC defect types: checking, assignment/ initialization, and interface. Manual code inspections have been shown to identify these three defect types and additionally function, documentation, and algorithm defects [3]. Therefore, the removal of ASI defects prior to manual code inspection could allow the inspectors to focus more on function, documentation, and algorithm defects.

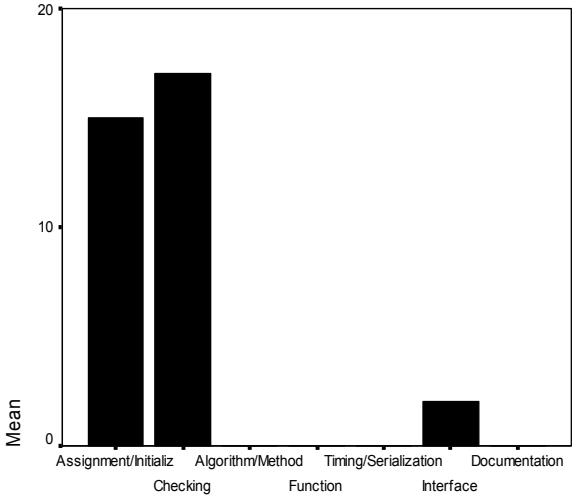

Figure 2: Mapping of FlexeLint defects to ODC defect types

We desired to further differentiate the defects identified by ASI. There has been research done on tailoring ODC to more specific requirements. For example, the ODC-CC for computational code was developed [15]. We present an automated inspection-based defect classification scheme that is based on six types of defects as shown in Table 1 developed at Nortel Networks.

Table 1: Defect types for automated inspections

Defect Type	Example Categories
Syntax and interface errors	Errors particular to the specific language.
Code volume	Remove redundant code
	Remove unreachable code
Architectural standards	Identify violations in use relationships among components
Coding standards	Identify risky language constructs
	Evaluate compliance with local standards
Potential sources of error or inefficiency	Identify undeclared, uninitialized, or unused entities
	Identify risky program logic (for example, unchecked function return values)
	Identify memory leaks
Actual errors	Identify possible division by zero, array bound errors, overflow, and null pointer accesses

We then classified the automated inspection faults found in Release 1 and Release 2 into these six types. The results of this classification are shown below in Figure 3. The actual percentage information has not been presented due to the proprietary nature of the data.

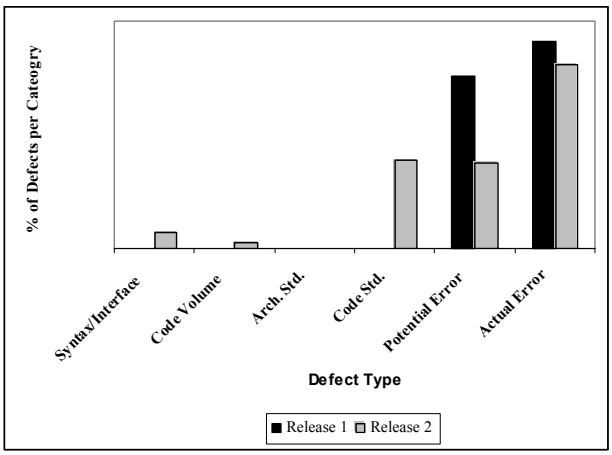

Figure 3: Automated defect classification

Our results indicate that the majority of the defects identified in the Nortel products were (1) potential sources of error or inefficiency and (2) actual errors. Release 2 also had coding standard defects identified.

5.2 Inter-Correlation between metrics

We then examine whether ASI results can be used as a predictor of field failures. The Spearman rank correlation is a commonly-used robust correlation technique [10] because it can be applied even when the association between elements is non-linear. The Pearson bivariate correlation requires the data to be distributed normally and the association between elements to be linear. The correlation results of the ASI faults with the actual field failures, churn, and deleted lines of code for Release 1 is shown in Table 2. The results indicate that the Spearman rank correlation coefficient between the automated inspection faults and the actual failures is statistically significant[4] indicating a relationship between the two. Further, a statistically significant correlation exists between actual field failures and the churned and deleted lines of code (LOC). As a result, we used the churned and deleted LOC information in the multiple regression model, as will be discussed in Section 5.3.

Table 2: Spearman rank correlation - Release 1

		Actual failures	Automated faults	Churn	Deleted
Actual failures	Correlation Coefficient	1.000	.404	.456	.424
	Sig. (2-tailed)	.	.001	.000	.001
Automated faults	Correlation Coefficient		1.000	.665	.623
	Sig. (2-tailed)		.	.000	.000
Churn	Correlation Coefficient			1.000	.893
	Sig. (2-tailed)			.	.000
Deleted	Correlation Coefficient				1.000
	Sig. (2-tailed)				.

To assess the repeatability of and to increase our confidence in the above results, we analyzed the correlations with the Release 2 data. The Release 2 results, as shown in Table 3, are similar to those of Release 1 demonstrating a correlation between actual failures and automated faults, churned LOC, and deleted LOC.

[4] Note: All statistical analysis was performed using SPSS®. SPSS does nor provide statistical significance beyond 3 decimal place. So (p=0.000) should be interpreted as (p<0.0005). Statistical significance is calculated at 95% of confidence.

Table 3: Spearman rank correlation - Release 2

		Actual failures	Automated faults	Churn	Deleted
Actual failures	Correlation Coefficient	1.000	.498	.559	.550
	Sig. (2-tailed)	.	.000	.000	.000
Automated faults	Correlation Coefficient		1.000	.553	.487
	Sig. (2-tailed)		.	.000	.000
Churn	Correlation Coefficient			1.000	.947
	Sig. (2-tailed)			.	.000
Deleted	Correlation Coefficient				1.000
	Sig. (2-tailed)				.

5.3 Prediction models

In this section, we describe the model building activities involved in our case study. Section 5.3.1 illustrates the use of multiple regression to predict field failures based upon the three metrics (automated inspection faults, code churn and deleted lines of code). Section 5.3.2 presents a discriminant analysis of fault-prone programs with respect to these three metrics. For both model building activities, Release 1 is used to build the model, and Release 2 is used to validate the model.

5.3.1 Multiple regression. Software metrics have previously been used with multiple linear regression analysis to predict quality [18, 20]. The multiple coefficient of determination, R^2, provides a quantification of how much variability in the quality can be explained by the regression model. Using multiple linear regression on Release 1 with the actual defects as the dependant variable and the three metrics (automated defects, churn and deleted lines of code) as the predictors, we form a standard multiple linear regression equation as shown in Equation 1.

$$\textbf{Estimated defects} = c + \alpha_1 * AD + \alpha_2 * CH + \alpha_3 * DL \qquad (1)$$

where c is the regression constant and α_1, α_2, α_3 are the regression coefficients, AD is automated inspection faults, CH is quantity of lines of churned code, and DL is quantity of deleted lines. Actual defects are the defects that are found in the field.

The F-ratio to test the hypothesis that all regression coefficients are zero was statistically significant ($F = 5.721$; $p = 0.002$). Table 4 shows a summary of the built regression model. In order to protect proprietary information, the metric coefficients are not shown.

Table 4: Regression model summary

R	R Square	Adjusted R Square	Std. Error of the Estimate
.481	.231	.191	4.220

Figure 4 shows the normal probability plot of the regression standardized residual using the actual defects as the dependent variable.

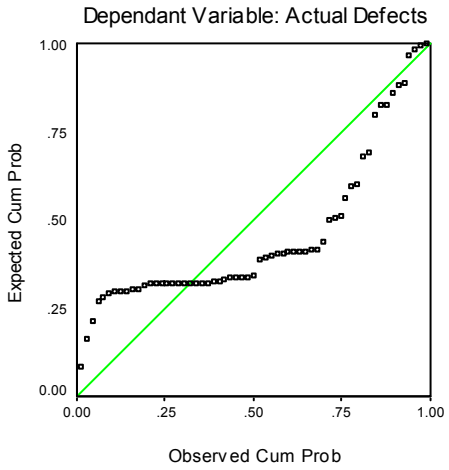

Figure 4: Normal probability plot of the regression standardized residual

This regression model, built using Release 1 data, was used as a predictor for the Release 2 components. The Spearman rank correlation shows that the model was effective for predicting actual faults, $\rho=0.526$ ($p<0.0005$) (linear correlation coefficient = 0.476, $p<0.0005$). Figure 5 shows the relationship between the estimated and the actual defects for the 61 Release 2 components. (The scale is not provided to protect proprietary information.) The clustering of the scatter plot shows that the estimated defects and actual defects are highly related (in a linear fashion).

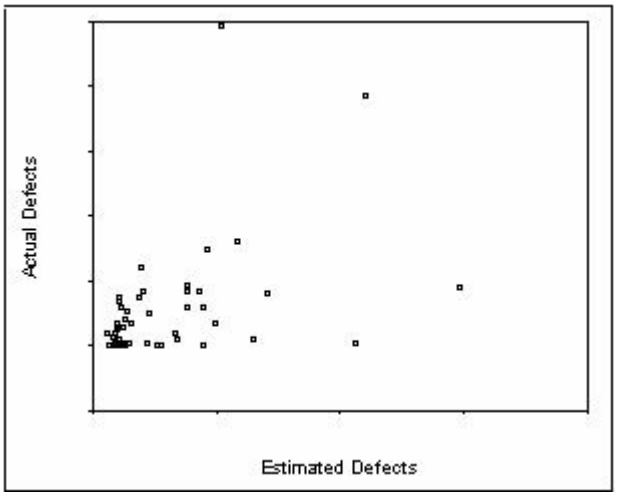

Figure 5: Estimated vs. actual number of defects

5.3.2 Discriminant analysis. Discriminant analysis, a statistical technique used to categorize programs into groups based on the metric values, has been used as a tool for the detection of fault-prone programs. Munson et al. [19] used discriminant analysis for classifying programs as fault-prone with a large medical-imaging software system. The classification resulting from Munson's use of discriminant analysis/data splitting was fairly accurate. There were 10% false positives among the high quality programs (incorrectly classified as fault-prone) and 13% false negatives (incorrectly classified as not fault-prone) among the fault-prone programs. Identifying fault-prone components enables developers to retest and rework the fault-prone modules more so that they are made more reliable.

We ran discriminant analysis using two models:

- Model 1: We use the ASI faults (only) from Release 1 to build the discriminant function and then use the resulting function to classify the fault prone components of Release 2.
- Model 2: We use the ASI faults, churn, and deleted LOC from Release 1 to build the discriminant function and then use it to assess the fault prone components of Release 2.

To classify a component as fault prone, we use the formula in Equation 2,

$$\textbf{LL (fault-proneness)} = \mu_x - z_{\omega/2} * \frac{\text{Std dev of actual failures/component}}{\sqrt{n}} \qquad (2)$$

where, LL is the lower confidence bound on an estimate, μ_x is the mean number of actual failures, and n is the number of components. $z_{\omega/2} = 1.96$ for a standard normal distribution for n>30. If the number of actual failures in a component is less than LL then the component is classified as not fault-prone otherwise it is classified as fault-prone. Table 5 shows that the results of the discriminant functions built using the two models.

Table 5: Model parameters of the discriminant functions

Model	Eigenvalue	False Positives	False Negatives
1: Automated faults	.047	11 %	19%
2: ASI faults, churn, deleted LOC	.259	13 %	13%

As shown in Table 5, Model 2 has a higher eigenvalue. The eigenvalue serves as a measure of the discriminative ability of the model, i.e. it is a measure of how good the discriminative function is with respect to the classification of the data. Model 2 is more effective overall at correctly classifying modules, having similar false positive and false negative rates to Munson's work discussed above.

6.0 Preliminary metrics validation

Using a comprehensive metric validation scheme as proposed by Schneidewind [26], we examine our results of using the three metrics (ASI faults, code churn and deleted lines of code) for predicting field failures and for classifying fault-prone modules. Schneidewind's validation scheme has six criteria: association, consistency, discriminative power, tracking, predictability, and repeatability of the metrics. Assuming the indicator of quality is the number of defects (F) and the metric suite (M) is comprised of the ASI results, code churn and deleted lines, we test these validity criteria. Our analysis is shown in Table 6 that indicates that M satisfies the association, consistency, discriminative power, tracking, predictability, and repeatability criterion at statistically significant levels.

7.0 Conclusions

Three years ago, Nortel Network began to incorporate the use of ASI tools into their development process. In this paper, we share the results of analyzing defect data for one large 800 KLOC product that underwent ASI for two releases. The mapping of FlexeLint defects to ODC defect types indicated that ASI tools predominantly identify three ODC defect types: interface, checking, and assignment/initialization defects. Therefore, ASI can be used to identify three of the six ODC defect types often found in manual code inspection. Additionally, delineation of the identified defects beyond the ODC defect types revealed that these tools most often identified potential errors (such as unchecked function return values) and actual errors (such as array bounds errors).

The ASI faults, code churn, deleted lines of code, and actual defects of the product under study were analyzed. The main observations are as follows:

- A multiple regression approach for empirical analysis of actual field failures using ASI faults, code churn and deleted lines of code leads to statistically significant results, indicating these are effective in-process metric indicators of actual field quality; and
- Discriminant analysis performed using automated inspection faults, code churn and deleted lines of code is a feasible technique for detecting fault-prone programs

8.0 Future work

We will perform an economic analysis of adding ASI as a defect reduction filter in the development process. The economic analysis will consider the cost of purchasing the tool, the cost of pre-screening to reduce

false positives, and the cost of fixing defects. These costs will be compared with similar defect reduction costs of manual inspection, testing, and field escapes.

We will compare the ODC profile of products which have undergone both ASI and manual inspection and those that have only undergone manual inspection to further investigate the defect-removal potential of ASI. Additionally, we will validate our predictive and component classification models with other software products. Further, we plan to collect historical data in the development of standards so that new software undergoing development can identify fault-prone components earlier. Nortel had previously established the Enhanced Measurement for Early Risk Assessment of Latent Defects (Emerald) decision support system [11, 12]. Emerald was shown to be effective for improving software reliability and for facilitating accurate correction of field problems. We will examine the potential of incorporating ASI defect information in Emerald.

Table 6: Metrics validation summary

F = number of defects and M = the metric suite comprised of the ASI, code churn and deleted lines

Characteristic	Description	Evaluation Strategy	Results
Association	To test if there is a sufficient linear association between the F and M to warrant using M as an indirect measure of F	Using correlation analysis. The correlation coefficient at a statistically significant level indicates the association between these measures and the defects.	**YES.** The results were statistically significant for a linear correlation analysis between F and M ($p = .002$)
Consistency	To verify the consistency of the predictive ability of the metrics, the rank correlation coefficient r between F and M should exceed a certain threshold (developer defined as acceptable) to assess whether there is sufficient consistency between the ranks of F and M to warrant using M as an indirect measure of F	The rank correlation coefficient 'r' between F and M must statistically significant at acceptable levels of confidence (95 % in this case)	**YES.** The results were statistically significant ($p < .05$ for all) for a Spearman rank correlation analysis for M. (as shown in section 5.2)
Discriminative Power	To test if the metric(s) shall be able to discriminate between high quality and low quality software components.	Using discriminant analysis as outlined in Section 4.2.2	**YES.** The model has discriminative power. (as shown in section 5.3.2)
Tracking	This criterion serves to assess if M is capable of tracking changes in F, i.e. are changes in M reflected by appropriate changes in F.	This can be studied by using a scatter plot from which it can be observed if the changes in M are reflected by appropriate changes in the actual defects.	**YES.** Figure 5 indicates the tracking power of M.
Predictability	This criterion is used to study the predictability of F from M. The percentage of error between the predicted and actual values of F assesses the predictability criterion.	Using an OLS (Ordinary Least Square) regression we can find the best fit for a certain subset of data. By applying the remaining data to the regression equation, we can evaluate the predictability of the data set.	**YES.** Section 5.3.1 indicates the high correlation between the actual and the estimated values obtained using multiple regression.
Repeatability	This criterion assess whether the experiment is repeatable.	This is validated by carrying out the analysis to determine if the results are repeatable and comparable under similar conditions.	**YES.** The results were repeated for Release 1 and for Release 2.

Acknowledgements

We would like to thank Kiem Ngo of SWDD and the developers at Nortel Networks for their support of this research. We also thank the North Carolina State University (NCSU) Software Engineering reading group for their helpful suggestions on this paper. This work was funded by Nortel via an NCSU Center for Advanced Computing and Communications (CACC) enhancement grant.

References:

[1] Briand, L. C., El Emam, K., Freimut, B., Laitenberger, O., "Quantitative Evaluation of Capture-Recapture Models to Control Software Inspections," presented at International Symposium on Software Reliability Engineering, 1998.pp. 234-244

[2] Briand, L. C., El Emam, K., Laitenberger, O., Fussbroich,T., "Using Simulation to Build Inspection Efficiency Benchmarks for Development Projects," presented at International Conference on Software Engineering, 1998.pp. 340-449

[3] Chillarege, R., Bhandari, I.S., Chaar, J., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M-Y.,, "Orthogonal Defect Classification - A Concept for In-Process Measurements," *IEEE Transactions on Software Engineering*, vol. 18, pp. 943-956, 1992.

[4] Dow, H. E., Murphy, J.S., "Detailed Product Knowledge is not Required for a Sucessful Formal Software Inspection," presented at Seventh Software Engineering Process Group Meeting, Boston, MA, 1994.pp.

[5] Eick, S. G., Loader, C.R., Long, M.D., Votta, L.G., Weil S.V., "Estimating Software Fault Content before Coding," presented at International Conference on Software Engineering, 1992.pp. 59-65

[6] Engler, D., Chelf, B., Chou, A., Hallem, S., "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions," presented at OSDI 2000, 2000.

[7] Evans, D., Guttag, J., Horning, J., Tan, Y., M.,, "LCLint: A Tool for Using Cpecifications to Check Code," presented at ACM-SIGSOFT Foundations in Software Engineering, 1994.pp. 87-96

[8] Fagan, M. E., "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, vol. 12, pp. 744-751, 1986.

[9] Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, vol. 15, pp. 182-211, 1976.

[10] Fenton, N. E., Pfleeger, S.L., *Software Metrics*. Boston, MA: International Thompson Publishing, 1997.

[11] Hudepohl, J., S. J. Aud, T. Khoshgoftaar, E. B. Allen, and J. Mayrand, "Emerald: Software Metrics and Models on the Desktop," in *IEEE Software*, vol. 13, September 1996, pp. 56-59.

[12] Hudepohl, J., W. Jones, and B. Lague, "EMERALD: A Case Study in Enhancing Software Reliability," in *Eighth International Symposium on Software Reliability Engineering (Case Studies)*. Albuquerque, New Mexico, 1997, pp. 85-91.

[13] IEEE, "IEEE 1028-1988: IEEE Standard for Software Reviews and Audits," 1988.

[14] Johnson, S. C., "Lint, a C Program Checker," AT & T Bell Laboratories Unix programmers manual, computer science tech report 65, 1978.

[15] Kelly, D., Shepard, T., "A Case Study in the use of Defect Classification in Inspections," presented at IBM Centre for Advanced Studies Conference, 2001.pp. 26-39

[16] Khoshgoftaar, T. M., Allen, E.B., Goel, N., Nandi, A., McMullan, J., "Detection of Software Modules with high Debug Code Churn in a very large Legacy System," presented at International Symposium on Software Reliability Engineering, 1996.pp. 364-371

[17] Khoshgoftaar, T. M., Munson, J.C., "The Lines of Code Metric as a Predictor of Program Faults : A Critical Analysis," presented at Computer Software and Applications Conference (COMPSAC), 1990.pp. 408-413

[18] 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.pp. 72-79

[19] Munson, J. C., Khoshgoftaar, T.M., "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering*, vol. 18, pp. 423-433, 1992.

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

[21] O'Neill, D., "Issues in Software Inspection," in *IEEE Software*, 1997, pp. 18-19.

[22] Porter, A. A., Johnson, P.M., "Assessing Software Review Meetings : Results of a Comparative Analysis of two Experimental Studies," *IEEE Transactions on Software Engineering*, vol. 23, pp. 129-145, 1997.

[23] Reasoning Inc., "Automated Software Inspection: A New Approach to Increase Software Quality and Productivity."

[24] Runeson, P., Wohlin, C., "An Experimental Evaluation of an Experience-Based Capture-Recapture Method in Software Code Inspections," *Journal of Empirical Software Engineering*, pp. 381-406, 1998.

[25] Rus, I., Shull, F., Donzelli, P., "Decision Support for Using Software Inspections," presented at 28th Annual NASA Goddard Software Engineering Workshop, 2003.pp. 3 - 11

[26] Schneidewind, N. F., "Methodology for Validating Software Metrics," *IEEE Transactions on Software Engineering*, vol. 18, pp. 410-422, 1992.

[27] Wiel, S. A. V., Votta, L.G., "Assessing Software Designs Using Capture-Recapture Methods," *IEEE Transactions on Software Engineering*, vol. 10, pp. 1045-1054, 1993.

[28] Wohlin, C., Runeson, P., Brantestam, J., "An Experimental Evaluation of Capture-Recapture in Software Inspections," *Software Testing, Verification and Reliability*, vol. 5, pp. 213-232, 1995.

Appendix A. Sample Compiler/Automated inspection Output Comparison

```
 1  #include <stdio.h>
 2  #include <strings.h>
 3  #include <stdlib.h>
 4  #define MAX_BUF 1024
 5  int main() {
 6     char buf[MAX_BUF], sbuf[128];
 7     float fVal, fMax = 100.0;
 8     int i, nTest, nFoo, nCount = 0;
 9     for (i = 0; i <= MAX_BUF; i++)
10        buf[i] = 0;                              // access of out-of-bounds pointer
11     nTest = 0;
12     fVal = fMax/nCount;                         // division by 0
13     sprintf(sbuf, "the answer is: %s for %d", fVal, buf);
14     strcat(sbuf, getenv("MY_HOME"));            // possible use of null pointer
15     switch (nTest)
16     {
17        case 1: nFoo = 1; break;
18        case 2: nFoo = 21;                       // control flows into case/default
19        case 3: nFoo = 321; break;
20     }
21     nTest = nFoo * 2;
22     for(;;) {
23        nTest++;
24        if (nTest > nFoo) {
25           nTest = 0;
26        }
27        goto done;
28     }
29     if (nTest = 0)
30        return 0;
31     done:
32     {
33         unsigned int nTest3 = 1;
34         while (1)
35              nTest3 = nTest3 << 1;
36     }
 37 }
```

Compiler Warnings

ctoa.c:5: warning: function declaration isn't a prototype
ctoa.c: In function `main':
ctoa.c:13: warning: format argument is not a pointer (arg 3)
ctoa.c:13: warning: int format, pointer arg (arg 4)
ctoa.c:29: warning: suggest parentheses around assignment used as truth value
ctoa.c:8: warning: `nFoo' might be used uninitialized in this function

Automated inspection Warnings (FlexeLint)

ctoa.c(10): Warning 661: Possible access of out-of-bounds pointer (1 beyond end
of data) by operator '[' [Reference: file ctoa.c: lines 9, 10]
ctoa.c(12): Warning 414: Possible division by 0 [Reference: file ctoa.c: line 8]
ctoa.c(13): Warning 560: argument no. 3 should be a pointer
ctoa.c(13): Warning 626: argument no. 4 inconsistent with format
ctoa.c(14): Warning 668: Possibly passing a null pointer to function 'strcat(sig
ned char *, const signed char *)', arg. no. 2 [Reference: file ctoa.c: line 14]
ctoa.c(19): Warning 616: control flows into case/default
ctoa.c(20): Info 744: switch statement has no default
ctoa.c(21): Warning 644: Variable 'nFoo' (line 8) may not have been initialized
ctoa.c(27): Info 801: Use of goto is deprecated
ctoa.c(29): Warning 527: Unreachable
ctoa.c 29 Info 720: Boolean test of assignment
ctoa.c 29 Warning 506: Constant value Boolean
ctoa.c 29 Info 774: Boolean within 'if' always evaluates to False [Reference: file ctoa.c: line 29]
ctoa.c(34): Info 716: while(1) ...

Appendix B. Classification of sample of FlexeLint automated inspection faults

Automated inspection	*ODC Classification*
Likely use of null pointer	Assignment/Initialization
Possible division by 0	Checking
Access of out-of-bounds pointer	Checking
Creation of out-of-bounds pointer	Assignment/Initialization
Passing null pointer to function (e.g. memcpy, strcpy…)	Checking
Apparent data overrun for function (e.g. memcpy, strcpy…)	Checking
Apparent access beyond array for function (e.g. memcpy, strcpy…)	Checking
Creation of memory leak in assignment to variable	Assignment/Initialization
Inappropriate de-allocation	Assignment/Initialization
Custodial pointer 'Symbol' has not been freed or returned	Assignment/Initialization
Boolean argument to relational	Checking
Unusual use of a Boolean	Checking
Unreachable	Checking
Static variable or static function was not referenced	Assignment/Initialization
A named variable was declared but not referenced in a function	Assignment/Initialization
An auto variable was used before it was initialized	Assignment/Initialization
Function 'Symbol' should (not) return a value	Checking
A construct of the form if(e); was found	Assignment/Initialization
Returning address of auto variable	Checking
Control flows into case/default	Checking
Possibly passing a null pointer to function	Checking
Possible memory leak in assignment	Assignment/Initialization
Returning address of auto through variable 'Symbol'	Checking
Boolean test of assignment	Checking
Suspicious use of ;	Checking
Attempting to 'delete' a non-pointer	Checking
Checking for assignment to this; for example, if (&arg == this)	Assignment/Initialization
Static variable 'Symbol' found within inline function in header file	Interface
Base class 'Name' absent from initializer list for copy constructor	Interface
Member 'Symbol' not assigned by assignment operator	Checking
Pointer member 'Symbol' neither freed nor zeroed by destructor	Assignment/Initialization
Member 'Symbol' possibly not initialized by constructor	Assignment/Initialization
Member 'Symbol' possibly not initialized	Assignment/Initialization
Value of variable 'Symbol' (Location) indeterminate (order of initialization)	Assignment/Initialization
Binary operator returning a reference (e.g. X &operator+ (X &, X &);)	Checking
Overloading special operator; for example, || or &&	Checking