On the Value of Static Analysis for Fault Detection in Software

Jiang Zheng, *Student Member*, *IEEE*, Laurie Williams, *Member*, *IEEE*, Nachiappan Nagappan, *Member*, *IEEE*, Will Snipes, *Member*, *IEEE*, John P. Hudepohl, and Mladen A. Vouk, *Fellow*, *IEEE*

Abstract—No single software fault-detection technique is capable of addressing all fault-detection concerns. Similarly to software reviews and testing, static analysis tools (or automated static analysis) can be used to remove defects prior to release of a software product. To determine to what extent automated static analysis can help in the economic production of a high-quality product, we have analyzed static analysis faults and test and customer-reported failures for three large-scale industrial software systems developed at Nortel Networks. The data indicate that automated static analysis is an affordable means of software fault detection. Using the Orthogonal Defect Classification scheme, we found that automated static analysis is effective at identifying Assignment and Checking faults, allowing the later software production phases to focus on more complex, functional, and algorithmic faults. A majority of the defects found by automated static analysis appear to be produced by a few key types of programmer errors and some of these types have the potential to cause security vulnerabilities. Statistical analysis results indicate the number of automated static analysis faults can be effective for identifying problem modules. Our results indicate static analysis tools are complementary to other fault-detection techniques for the economic production of a high-quality software product.

Index Terms—Code inspections, walkthroughs.

1 Introduction

No single fault-detection technique is capable of addressing all fault-detection concerns [40]. One such fault-detection technique is static analysis, the process of evaluating a system or component based on its form, structure, content, or documentation [16], which does not require program execution. Inspections are an example of a classic static analysis technique that relies on the visual examination of development products to detect errors,[1] violations of development standards, and other problems [16]. Tools are increasingly being used to automate the identification of anomalies that can be removed via static analysis, such as coding standard noncompliance, uncaught runtime exceptions, redundant code, inappropriate use of variables, division by zero, and potential memory leaks. We term the use of static analysis tools *automated static analysis (ASA)*. Henceforth, the term *inspection* is used to refer to manual inspection. ASA may enable software engineers to fix faults before they surface more publicly in inspections or as test and/or customer-reported failures. In this paper, we report the results of a study into the value of ASA as a fault-detection technique in the software development process.

The study was a research partnership between North Carolina State University and Nortel Networks. Since 2001, Nortel has included inspection and ASA in its development process for over 33 million lines of code (LOC). In our research, we examined defect data from three large-scale Nortel software products (over three million LOC in total) written in C/C++ that underwent various combinations of inspection and ASA. We had so much available data that we used the Goal-Question-Metric (GQM) paradigm [2] to motivate and focus our data collection and analysis. GQM is a goal-driven method which can be used to keep metrics programs in alignment with business and technical goals; our study was in alignment with Nortel's goals.

The goal of the study was to determine whether automated static analysis can help an organization economically improve the quality of software products. This goal can be restated utilizing the GQM process goal template [9]:

> Analyze the **final product**
> for the purpose of **improvement**
> with respect to **final product quality and cost**
> from the viewpoint of the **organization**
> in the context of **Nortel Networks network service products**

1. A human error leads to insertion of a physical fault into a software product element (e.g., specifications, design, code, test-case, etc.), this fault may propagate (in the form of one or more defects) to the executable code. When such a defect (or combination of defects) is encountered during software execution, software system may enter an error-state. This error-state may or may not persist, and may or may not be masked. When this error state (or a combination of time-separated error-states) results in an observable anomaly, we say that a failure has occurred [16]. In this paper, we may use terms defect and fault interchangeably.

- *J. Zheng, L. Williams, and M.A. Vouk are with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695. E-mail: {jzheng4, lawilli3, vouk}@ncsu.edu.*
- *N. Nagappan is with Microsoft Research, Redmond, WA, 98052. E-mail: nachin@microsoft.com.*
- *W. Snipes and J.P. Hudepohl are with Nortel Networks, Software Dependability Design (SWDD), Research Triangle Park, NC 27709. E-mail: {wbsnipes, hudepohl}@nortelnetworks.com.*

Manuscript received 12 July 2005; revised 31 Jan. 2006; accepted 28 Feb. 2006; published online DD Apr. 2006.
Recommended for acceptance by D. Rombach.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0197-0705.

We broke this research goal into seven questions. Each of these questions, including the metrics collected and analyzed to formulate the answer, will be discussed in detail in Section 4 of this paper. In general, each of the seven questions is independent of the other questions and has its own unique set of metrics. The questions are as follows:

- Q1: Is ASA an economical means of software fault detection?
- Q2: Will my delivered product be of higher quality if ASA is part of the development process?
- Q3: How effective is ASA at detecting faults compared with inspection and testing?
- Q4: Can ASA be helpful for identifying problem modules?
- Q5: What classes of faults and failures are most often detected by ASA, by inspection, or by testing? What classes of defects escape to customers?
- Q6: What kinds of programmer errors are most often identified via ASA?
- Q7: Can ASA be used to find programming errors that have the potential to cause security vulnerabilities?

The rest of this paper is organized as follows: Section 2 provides background information. Section 3 discusses the implementation of ASA at Nortel, our data collection and analysis procedures, and limitations of our study. Section 4 reviews our findings on each of the seven research questions. Finally, Section 5 and Section 6 present the conclusions and future work, respectively.

2 Background

In this section, we provide an overview of ASA tools and provide background on our classification scheme choice to answer Q5.

2.1 Automated Static Analysis Tools

ASA can be used as an added fault-detection filter in the software development process. ASA tools automate the identification of certain types of anomalies, as discussed above, by scanning and parsing the source text of a program to look for a fixed set of patterns in the code. ASA utilizes control flow analysis, data flow analysis, interface analysis, information flow analysis, and path analysis of software code. There is a range of programmer errors which can be automatically detected by ASA, and there are some that can never be detected by ASA [33], [40]. Additionally, one study of ASA tools indicates that each tool seems to find different, sometimes nonoverlapping, bugs [37]. Although detected anomalies are not always due to actual faults, often they are an indication of an error.

An important benefit of ASA tools is that they do not necessitate execution of the subject program yet infer results applicable to all possible executions [33]. As such, ASA can complement the error-detection facilities provided by language compilers. ASA tools are particularly valuable for programming languages like C that do not have strict type rules; the checking the C compiler can do is limited.

There is a range of ASA tools and services deployed for C/C++ code. For example, FlexeLint[2] checks C/C++ source code to detect errors, inconsistencies, nonportable constructs, and redundant code. FlexeLint is a Unix-based tool (akin to the Window-based PC-lint). Reasoning's[3] Illuma is a static analysis-based service that finds defects in C/C++ applications. Organizations send their code to Reasoning. Reasoning performs the ASA, 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 detected by Illuma include: NULL pointer dereferencing, out of bounds array access, memory leaks, bad deallocation, and uninitialized variables. Klocwork[4] has two ASA tools, inForce and GateKeeper. inForce performs static analysis 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 among modules (as compared to intended relationships), code clusters (cyclic relationships), high-risk code files and functions, potential logical errors, and areas for improvement.

PREfix [28] analysis is based on the call graphs of a program which are symbolically executed. The PREfast [28] tool is a "fast" version of the PREfix tool where certain PREfast analyses are based on pattern matching in the abstract syntax tree of the C/C++ program to find simple programming mistakes. PREfix/PREfast are used to find defects, such as uninitialized use of variables, NULL pointer dereferences, the use of uninitialized memory, and double freeing of resources.

An important issue with the use of ASA tools is the inevitability of the tool reporting significant amounts of false positives or bugs that the program does not contain upon a deeper analysis of the context. There can be as many as 50 false positives for each true fault found by some static analysis tools [35]. The FindBugs tool [14] reports a low of only 50 percent false positives. Often, static analysis tools can be customized and filters can be established so that certain classes of faults are not reported, reducing the number of false positives. Some organizations, such as Nortel, contract a prescreening service to identify and remove false positives in the static analysis output prior to involvement by their own programming teams.

2.2 Fault Classification Schemes

To answer Q5, we classify the defects that can be detected via ASA using an established taxonomy. Each ASA tool defines its own unique defect types. However, these defect types are related to the lower-level types of faults that can be identified by ASA, and our goal in choosing a taxonomy is to place faults detected by ASA in the context of the entire development process.

Fault classification schemes (taxonomies) are intended to have categories that are distinct, i.e., orthogonal, and to remove the subjectivity of the classifier [20]. There are several fault classification schemes presented in the literature. Basili et al. [3] proposed a classification scheme in research on requirements defects. The scheme consists of

2. http://www.gimpel.com/html/products.htm.

3. http://www.reasoning.com.
4. http://www.klocwork.com.

TABLE 1
ODC Defect Types and Process Associations, Adapted from [8]

Process Association	Defect Type
Design	Function
Low Level Design	Interface, Checking, Timing/Serialization, Algorithm
Code	Checking, Assignment
Library Tools	Build/Package/Merge
Publications	Documentation

five defect classes: Omission, Incorrect Fact, Inconsistency, Ambiguity, and Extraneous Information. Travassos et al. [39] tailored the five defect classes to object-oriented design in 1999. Schneider et al. [38] researched the defect classification scheme for fault detection in user requirement document, and Ackerman et al. [1] developed another defect classification scheme for requirements defects.

IEEE also provides a classification scheme of anomalies found in software and its documentation [17]. The mandatory first level of IEEE classification is Logic problem, Computation problem, Interface/timing problem, Data handling problem, Data problem, Documentation problem, Document quality problem, and Enhancement. The first five of these classes may all be injected in the coding or low-level design phase. As a result, the IEEE classification scheme does not clearly point toward the phase of development in which the fault was injected. The Beizer taxonomy [4] offers a comprehensive means of classifying faults. However, we were interested in identifying when in the development process the fault was injected, and Beizer's classification required more detailed information than we were provided in our Nortel data.

The goal of IBM's Orthogonal Defect Classification (ODC) [8] scheme is to categorize defects such that each defect type is associated with a specific stage of development. El Emam et al. [10] investigated the defect classification scheme that has been applied in ODC, indicating the use of this defect classification scheme is, in general, repeatable in many areas of software engineering. ODC has eight defect types. Each defect type is intended to point to the part of the development process that needs attention. The relationships between these defect types and process associations are shown in Table 1, which is adapted from [8]. Therefore, the ODC scheme can be used to indicate the development phase in which a defect was injected into the system. In our research, we analyzed software faults and failures and assigned each an ODC category.

3 Case Study Details

In this section, we describe how ASA was used at Nortel and the products included in the data analysis and the limitations of the case study.

3.1 ASA at Nortel

Nortel provides communications products to commercial customers. Beginning in 2001, the Software Dependability Design (SWDD) group at Nortel began to work with development teams to include ASA in their development process prior to releasing products to customers. Depending upon the team, ASA could be done prior to inspection, after inspection but prior to system test, or during system test. For most product groups, the transition to using ASA 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. The most recently released product (considered Release N-1) undergoes ASA. Because the initial static analysis run for a product is likely to yield an excessive amount of false positives, the total list of warnings is sent to a prescreening service. The SWDD has an extended, usually contracted, core team of prescreeners. Nortel has also developed centralized, in-house expertise in the use of ASA tools and in the screening of the faults. Similarly to inspections, the efficacy of static analysis prescreening is dependent upon the screeners' skills and experiences. However, the skill and experience of the prescreeners can be programming language-centric rather than domain-specific.

The prescreeners scrutinize the raw warnings and read code to analyze why the warnings are generated. Additionally, we noticed that the screeners recorded some obvious errors in the code that could not have been detected by the ASA tools. For example, faults with type "Logic Error and Typo" and "Wrong Output Message" were noted. These types of faults could not be caught by a tool and must have been identified by the screeners manually examining the code. However, only a few such faults were logged in the final report, and they have little impact on the overall analysis. According to data of whether the ASA-identified faults are fixed or not, these prescreeners were able to reduce the false positive rate to approximately 1 percent. The prescreening also may have eliminated some real faults. However, these false negatives are difficult to identify.

The SWDD and development teams receive the post-screening report and fix the true faults that have been identified, beginning with the most severe faults. Some defects are left to be fixed in maintenance releases or later releases because they do not impact customer-observable behavior or critical functions. Once the higher priority true ASA faults have been fixed (considered to be the release N), the product undergoes ASA again (on release N) to make sure the defects were fixed. The release N is then considered to be the "clean baseline." This start-up phase typically takes between two and six months, depending on the team's release cycles.

Once the team has been through the start-up phase, ASA is an additional fault-detection filter in the development process and ASA is done *in-process*. Only new and changed code undergoes ASA from this point forth, and ASA is then often run by the developers without the involvement of prescreeners. Depending upon the developer, the frequency of doing ASA varies. ASA can be run when a component is complete, or a developer can run the ASA tools more incrementally as code is being developed.

Researchers and practitioners in Japan used a similar phased approach of introducing ASA prior to system test [27]. At first, a support group worked closely with development organizations, introducing ASA into their process, developing filters to reduce false positives, and prioritizing ASA faults

TABLE 2
Summary of the Data Analyzed

Product/Release		ASA	Inspection	CR
A		FlexeLint, Klocwork	Not performed	Yes
B		FlexeLint	No (No archive)	Yes
C	0	Not performed	Yes	Yes
	1	FlexeLint, Reasoning, Klocwork	Yes	Yes
	2	FlexeLint, Klocwork	Yes	Yes

to fix. Ultimately, development groups used ASA more autonomously. Through this process, they reduced static analysis-detectable faults from a high of 11.8 percent of system test failures to 0 percent [27].

3.2 Data Collection

We collected and analyzed fault data of three large-scale network service products. Data analysis consisted of faults reported by over 200 inspectors and testers, and by customers, for over three million LOC written in C/C++ developed at Nortel Networks. As will be discussed, each of these projects underwent a different combination of ASA, inspections, and testing. ASA or inspection may or may not have been conducted, and ASA could have been done prior to inspection, prior to test, or during test. FlexeLint, Reasoning's Illuma, and Klocwork's inForce and GateKeeper are some of the ASA tools and services used by Nortel. These static analysis tools are representative commercial tools which are used to detect errors in C/C++ source code [32]. In this study, the number of faults and their variety (in terms of types of faults) identified by Flexelint was about two times that identified by Klockwork, and about four times that identified by Reasoning's Illuma. Therefore, we based much of our analysis on the Flexelint faults.

The first two Nortel products we analyzed, henceforth called Product A and Product B, both underwent ASA. However, inspections were not performed on Product A. For Product B, the inspection faults were communicated via e-mail, not archived, and thus could not be analyzed. Data for several releases were available for the third product, which is referred to as Product C. We analyzed one release (C.0) that underwent inspection only because it was developed prior to instituting ASA. The following two releases (C.1 and C.2) underwent both ASA and inspections. In this case, release C.1 is considered as the release N-1 and C.2 is the release N and the "clean baseline." For Products A, B, C.1, and C.2, the ASA faults were sent to a prescreening service. The faults that were analyzed in our research were the true positives that remained after the prescreening. For each release, we scrutinized and classified a multitude of ASA faults, inspection records, and Change Request (CR) records. A CR was created for each test and customer-reported failure. The summary of the data that was analyzed for each product is shown in Table 2.

3.3 Limitations

There are certain limitations to our research. First, we classify faults into ODC categories based upon our subjective assessment of fault descriptions and of the ASA defect types. This categorization is most subjective in our analysis of customer-reported failure CR data. Frequently, causal analysis (or root-cause analysis) data was not available in the information provided for the CR failures. Therefore, most CRs were classified post hoc as Function failures due to the lack of information on exactly what was fixed and the cause of the initial fault. Second, the defects were classified without considering the severity or impact of the potential failure. Additionally, our results focused on the use of three ASA tools (FlexeLint, Klockwork, and Reasoning's Illuma), and therefore may not be representative of all ASA tools. These tools were chosen for analysis because of the availability of large amounts of fault data from Nortel. Finally, the results were obtained from Nortel Networks, and therefore relate to very large network service software systems written in C/C++. These results may not be representative of other types of software beyond those of Nortel Networks.

4 Results

In this section, we use the Nortel Networks data to provide insight into the seven questions posed in Section 1. The basic goal is *to determine whether ASA can help an organization economically improve the quality of a software product*. Sections 4.1 through 4.7 each address one of the seven questions. Using the GQM, in each section, the question is restated and the metrics that were collected and analyzed are listed. Metrics data are analyzed,and the implications of this data analysis to the posed question are discussed.

4.1 Economics of Fault Detection

- *Q1: Is ASA an economical means of software fault detection?*
- *Metrics: Quantity of defects found by inspection, quantity of defects found by ASA, preparation time, meeting time, static analysis tool cost, prescreening cost.*

Finding and fixing a problem in a later phase that was injected in an early phase of software development can be expensive because the longer the defects resides in the product, the larger the number of elements that will likely be involved in a fix [5]. Ideally, one would like to not inject problems in the first place. Barring that, one would like to detect them in the same phase in which they are injected. This is the reason why inspections and other non-execution-based methods of fault-detection are used in practice in phases where code execution is may not be an option. In general, inspections are considered an affordable fault detection technique [34], [36]. Hence, we use inspection as our reference point in our examination of the affordability

TABLE 3
Relative Final Product Quality

	Relative Quality (failures/KLOC$_C$)	Process step 1	Process step 2
Product B	0.25	Inspections	ASA (during Test)
Product C.2	0.32	Inspections	ASA (during Test)
Product C.0[6]	1.0	Inspections	
Product C.1	1.25	ASA	Inspections
Product A	1.84		ASA (during Test)

[6] *Product C.0 is the baseline because it was developed prior to ASA process.*

of ASA. Industrial data has shown that inspections are among the most effective of all verification and validation (V&V), measured by the percentage of faults typically removed from an artifact via the technique [36].

Most software inspections are performed manually. Software review meetings require 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 [34]. The effectiveness of ASA is less dependent upon these human factors due to the automation. However, ASA is not free from this dependence due to the need to identify the true defects from all those identified by the tool.

To determine the average cost of detecting a fault via inspections, we manually examined inspection records for Releases C.1 and C.2, a total of approximately 1.25 million LOC. Economic analysis was not performed on other products or releases because of the lack of either ASA faults or inspection data, as discussed in Section 3.2.

The inspection records for Releases C.1 and C.2 contained quantifications of preparation time and meeting time of each inspection participant and a profile of the faults, including location, type, complexity, and description identified in the inspection meeting. To obtain the average dollar cost of detecting a fault, we added the preparation and meeting time by all participants and divided by the number of faults found in the inspection, as shown in (1), where n is the number of inspection participants. We computed the cost per fault considering an average annual base salary plus benefits cost of $150,000 per inspection participant.[5]

$$\textbf{Avg. Cost of Fault Detection}_{\textbf{Inspection}} = \frac{\sum_{i=1}^{n}(minutes\ Time\ Meeting + Time\ Preparing)_n((Salary + Benefits)/minute)}{Quantity\ Faults\ Found}. \quad (1)$$

We computed the average cost of fault detection for ASA based on the cost of the tool license, the prescreening cost to eliminate false positives (on a per LOC basis, which is how the screeners are paid), and the number of remaining true positive faults. Some additional costs that were difficult to capture are the cost of learning how to use the ASA tools and computing resources to run the tool. This lack of information is a limitation of our findings, but we do not believe it would reverse our findings. Once an engineer learns to run the ASA tool, this cost can be amortized over all future projects. No additional computing resources were purchased specifically for running ASA. The computation is shown in (2):

$$\textbf{Avg. Cost of Fault Detection}_{\textbf{ASA}} = \frac{Tool\ Licence + (Cost\ Per\ Line)(LOC)}{Quantity\ True\ Positive\ Faults\ Found}. \quad (2)$$

To protect proprietary information, we only provide a ratio of the costs, as shown in (3).

$$\textbf{Cost Benefit}: \frac{Avg.\ Cost\ of\ Fault\ Detection_{ASA}}{Avg.\ Cost\ of\ Fault\ Detection_{Inspection}}. \quad (3)$$

Based upon our data, the computed ratios are 0.72 for C.1 and 0.61 for C.2, indicating that the cost of ASA per detected fault is of the same order of magnitude as the cost of inspections per fault detected. *These results indicate that ASA is a relatively affordable fault detection technique.*

4.2 Final Product Quality

- *Q2: Will a product be of higher quality if ASA is part of the development process?*
- *Metrics: Quantity of defects found by system testing, quantity of defects found by customer testing, churned thousand lines of code (KLOC).*

Table 3 provides a comparison of the final product quality. The measure used for final product quality is the number of total failures (both test and customer-reported failures) per churned KLOC (KLOC$_C$). We use failures per KLOC$_C$ as a measure of final product quality because it reflects the impact of change on the product. In the table, we use Product C.0 as the baseline product because this product/version was developed prior to ASA being instituted into Nortel's process. We normalized the failures per KLOC$_C$ metric relative to that of Product C.0 to protect Nortel's proprietary quality information. This gives us the Relative Quality column of Table 3.

As indicated, there is a wide variance in the relative quality of the products. As a result, *our analysis did not provide conclusive results about whether ASA will aid in the production of a higher quality product.*

4.3 Fault Detection Yield

- *Q3: How effective is ASA at detecting faults compared with inspection and testing?*
- *Metrics: Quantity of ASA faults, quantity of inspection faults, quantity of test failures, quantity of customer-reported failures.*

5. Based upon Nortel recourse costs.

TABLE 4
Defect Removal Yield for Different Fault Removal Techniques

Product / Release		Phase ASA performed	ASA (%)	Inspection (%)	Test (%)	Defect Removal Efficiency (%)
A		during test	23.39	Not performed	N/A[7]	
B		during test	Cannot compute due to unavailability of inspection records			97.76
C	0	Not performed	Not performed	39.55	96.73	98.02
	1	prior to inspection	31.00	20.48	98.18	99.00
	2	during test	36.53	33.21	62.57	99.40

[7] *The test yield and process yield could not be calculated due to lack of information on whether a failure is detected by test or customer.*

We examined fault detection yield as a measure of how well a fault detection practice identifies faults present in the artifact. Fault-detection yield refers to the percentage of defects present in the code at the time of the fault-detecting practice that were found by that practice [15], as shown in (4).

$$\textbf{Fault detection yield} = \frac{(100)(Quantity\ Faults\ Detected\ by\ Practice)}{Total\ Faults\ Detected\ by\ Practice\ and\ by\ Following\ Phases}. \quad (4)$$

Fault detection yield cannot be precisely computed until the product has been used extensively in the field and this measure decreases as more defects are found in the field. Additionally, we calculated the software defect removal efficiency [18] as a measure of how well a process removes faults before delivery. Software defect removal efficiency is the percentage of total bugs eliminated **before** the release of a product, as shown in (5). High levels of customer satisfaction correlate strongly with high levels of defect removal efficiency [18].

$$\textbf{Defect removal efficiency} = \frac{(100)(Quantity\ Faults\ Detected\ Except\ for\ Field\ Failure)}{Total\ Faults\ and\ Failures\ Detected}. \quad (5)$$

For all products, ASA was performed during test with the exception of Product C.1. The faults/failures yield and process yield are shown in Table 4. For Product C.2, the fault detection yield of test is relatively low because, in this case, ASA was performed during the test so that the denominator of (4) includes the number of ASA faults. However, the Defect Removal Efficiency for Product C.2 is 99.4 percent, which is essentially the same as that of other releases. Research indicates that top companies can achieve a greater than 95 percent software defect removal efficiency for commercial software [18], [19]. The values of Defect Removal Efficiency in the table are higher than industry benchmarks, indicating the high final quality of these products/releases.

These results indicate that the defect removal yield of ASA is not significantly different from that of inspections. The defect removal yield of execution-based testing is two to three times higher than that of ASA and therefore may be more effective at finding the defects remaining by that phase.

4.4 Problem Module Identification

- *Q4: Can ASA be helpful for identifying problem modules?*
- *Metrics: Quantity of ASA faults for individual modules, quantity of test failures per module, quantity of customer-reported failure per module.*

Software metrics have previously been used with multiple linear regression analysis to predict quality [26], [29]. Similarly to the LOC metric [25], code churn can also be used as an indicator of quality factors, such as fault-proneness [23], where code churn is a measure of the number of changed lines of code between two (not necessarily consecutive) releases of the code. Munson and Khoshgoftaar [30] used discriminant analysis for classifying programs as fault-prone with a large medical-imaging software system. Discriminant analysis is a statistical technique used to categorize modules into groups based on the metric values. The classification resulting from Munson's use of discriminant analysis/data splitting was fairly accurate. There were 10 percent false positives among the high quality programs (incorrectly classified as fault-prone) and 13 percent false negatives (incorrectly classified as not fault-prone) among the fault-prone programs. Other studies have demonstrated the use of metrics and discriminant analysis to identify problem modules [12], [13], [21], [22], [23], [24].

Other studies have also analyzed the ability of ASA defects to identify problem modules. Static analysis defects were used to predict the prerelease defect density of Windows Server 2003 [31]. The research demonstrated a positive correlation between the ASA defect density and prerelease testing defect density and that discriminant analysis of ASA defects could be used to separate high from low-quality components with 83 percent accuracy. Additionally, a preliminary investigation had been done on static analysis at Nortel [32]. Failure data from two releases of a large 800 KLOC product that underwent ASA during test were analyzed [32]. In addition, the ASA faults, code churn, and deleted LOC were used to form a multiple regression equation, which was effective for predicting the actual defects of the product. Finally, discriminant analysis indicated that ASA faults, code churn, and deleted LOC could be used as an effective means of classifying fault-prone programs. We continued this research by examining the potential of ASA faults alone for the identification of problem modules.

TABLE 5
Spearman Rank Correlation for Product B (for Modules)

		# of ASA faults	# of test failures	# of customer-reported failures	# of total failures
# of ASA faults	Correlation Coefficient	1.000	**.708**	**.604**	**.730**
	Sig. (2-tailed)	.	**.000**	**.002**	**.000**
# of test failures	Correlation Coefficient		1.000	.686	.992
	Sig. (2-tailed)		.	.000	.000
# of customer-reported failures	Correlation Coefficient			1.000	.750
	Sig. (2-tailed)			.	.000
# of total failures	Correlation Coefficient				1.000
	Sig. (2-tailed)				.

First, a Spearman rank correlation is computed on Product B to examine the relationship between ASA faults and the quantity of test/customer-reported failures at the module level. As a commonly used robust correlation technique [11], Spearman rank correlation can be applied even when the association between elements is nonlinear. We examined data of Product B because only Product B had clear module partition information. The numbers of ASA faults and test/customer-reported failures were counted for each module of Product B. The partition of the modules was provided by the development group. The correlation results of the ASA faults with test failures, customer-reported failures, and total failures is shown in Table 5. The relatively large correlation coefficient and small p-values indicate that a statistically significant[8] correlation exists between ASA faults and test/customer-reported failures. These results indicate that, when a module has a large quantity of ASA faults, the module is likely to be problematic in test and in the field.

Afterward, discriminant analysis was used as a tool to detect the fault-prone modules. In all the analysis, if there is no customer-reported failure in a module, then the module is classified as not fault-prone; otherwise, it is classified as fault-prone. The metrics used in the discriminant analysis include the following:

- the number of ASA faults,
- the number of test failures,
- the ASA fault density (number of ASA faults/source lines of code (SLOC)),
- the test failures density (number of test failures/SLOC),
- the normalized ASA faults density (number of ASA faults/churned SLOC), and
- the normalized test failures density (number of test failures/churned SLOC).

We built the discriminant function using either one of the above metrics only or the combination of two of them. Table 6 illustrates the summary of the discriminant functions built using the 21 models.

8. All statistical analysis was performed using SPSS®. SPSS does not provide statistical significance beyond three decimal places. So, (p = 0.000) should be interpreted as (p < 0.0005). Statistical significance is calculated at 95 percent of confidence.

For each analysis, the eigenvalue and the percentage of correctly classified modules are shown in the table. The eigenvalue is a measure of how good the discriminative function is with respect to the classification of the data. The larger the eigenvalue, the greater the discriminatory power of the model. We found that the model using the number of ASA faults and the number of test failures, henceforth referred to as Model 1, has the highest eigenvalue, indicating the discriminative ability of this model is the best. With this model, 87.5 percent of the modules are correctly classified. Additionally, 91.7 percent of the modules are correctly classified if the model uses the number of ASA faults and normalized test failures density, or the number of test failures and normalized test failures density, henceforth referred to as Models 2 and 3. However, the eigenvalues of these two models are relatively smaller than that of Model 1. The model parameters of the discriminant functions for Models 1 through 3 are shown in Table 7. For all three best models, no high quality module was incorrectly classified as fault-prone. However, 33 percent of the fault-prone modules were incorrectly classified as not fault-prone using Model 1, and there were 22 percent false negatives among the fault-prone modules using Model 2 and Model 3.

Alternately, 83.3 percent of the modules are correctly classified if the model uses the number of ASA faults only. Although this model has a lower eigenvalue and less accuracy, this model can be used to identify the fault-prone modules earlier in the development process, prior to test. The model using the number of ASA faults only is more valuable for more affordable corrective action and may have more utility than one that needs test data.

These statistical analysis results indicate that the number of ASA faults in a module can be a fairly good measure of fault-prone module identification prior to test. Developers can test and rework more on the identified fault-prone modules to improve their reliability.

4.5 Classes of Faults and Failures

- *Q5: What classes of faults and failures are most often detected by ASA, by inspection, or by system testing? What classes of defects escape to customers?*

TABLE 6
Summary of the Discriminant Analysis

		# of ASA faults	# of test failures	ASA faults density	test failures density	normalized ASA faults density	normalized test failures density
# of ASA faults	Eigen	.671	**1.156**	.702	.918	.700	**1.107**
	Correct	83.3 %	**87.5 %**	83.3 %	83.3 %	79.2 %	**91.7 %**
# of test failures	Eigen		.916	.917	.929	.923	**.984**
	Correct		87.5 %	87.5 %	87.5 %	87.5 %	**91.7 %**
ASA faults density	Eigen			.009	.182	.288	.245
	Correct			33.3 %	70.8 %	70.8 %	66.7 %
test failures density	Eigen				.180	.190	.299
	Correct				70.8 %	70.8 %	62.5 %
normalized ASA faults density	Eigen					.028	.259
	Correct					50.0 %	66.7 %
normalized test failures density	Eigen						.240
	Correct						66.7 %

TABLE 7
Model Parameters of the Discriminant Functions

Model	Metrics	Eigenvalue	False Positives	False Negatives
1	ASA faults; test failures	1.156	0 %	33 %
2	ASA faults; normalized test failures density	1.107	0 %	22 %
3	test failures; normalized test failures density	0.984	0 %	22 %

- *Metrics: Quantity of ASA faults by ODC type, quantity of inspection faults by ODC type, quantity of system test failures by ODC type, quantity of customer-reported failures by ODC type.*

We counted and classified (according to the ODC) the ASA and inspection faults and the test and customer-reported failures for the three products. In this section, we present our classification of the types of defects detected by each of these phases.

4.5.1 ASA Faults

Each fault detected by ASA had a documented problem report that contained detailed information, such as fault descriptions, location, preconditions, impact, severity, suggestion, and code fragment. The report for each problem was manually read, and then faults were classified according to ODC categories. Finally, the faults were counted to form a profile of faults. A summary of the results of this analysis is shown in Table 8. For the purpose of protecting proprietary information, only percentage is displayed in the tables. Because Flexelint was the only tool used on all three products, the table also shows a comparison of Flexelint only.

The results shown in the table indicate that ASA is effective at identifying two of the eight ODC defect types: Assignment and Checking. As we discussed in Section 2.4, Checking defects would most likely be injected in the low level design or coding phase, while Assignment defects would be injected in the coding phase. Therefore, it is logical that static analysis would be able to detect these types of faults.

4.5.2 Inspection Faults

At Nortel, inspections are guided by checklists. Unlike ASA faults, not all inspection faults were logged formally. Some of the faults were communicated via e-mails and could not be analyzed. However, the minutes of inspection meetings for Product C were well-recorded in text files via a recording tool. Product C.1 and C.2 underwent both ASA and inspection, while product C.0 underwent inspection only. Similarly to the analysis on ASA, every inspection file was manually read, and inspection faults were counted and classified according to ODC. The results of this classification are shown in Table 9. Note that inspectors also documented comments about code

TABLE 8
Mapping of ASA Faults Identified by All ASA Tools to ODC Defect Types

	A (%)	B (%)	C.1 (%)	C.2 (%)
Assignment				
-- All tools	70	77	73	73
-- Flexelint	73	77	53	71
Checking				
-- All tools	30	23	27	27
-- Flexelint	27	23	47	29
Other ODC types	0	0	0	0

TABLE 9
Classification of Inspection Faults

Defect Type	C.0 (%) No ASA	C.1 (%) After ASA	C.2 (%) Prior to ASA
Algorithm	30.60	38.27	37.44
Documentation	29.85	37.65	25.99
Checking	27.61	17.59	18.94
Assignment	5.22	4.01	7.93
Function	0.75	1.23	0.88
Interface	1.49	0.62	0
Build/Package/Merge	4.48	0.62	8.81
Timing/Serialization	0	0	0

TABLE 10
Priority Summary of CR Data for Product B (Test Failures)

Defect Type	Priority (%)				
	1	**2**	**3**	**4**	**Total**
Function	1.21	48.89	5.63	0	55.73
Assignment	0	3.22	0.60	0	3.82
Interface	0	0.20	0	0	0.20
Checking	0	0.80	0	0	0.80
Timing/Serialization	0	0	0	0	0
Build/Package/Merge	0	1.61	0.10	0	1.81
Documentation	0	0	0	0	0
Algorithm	0	29.78	7.85	0	37.63
Total	1.21	84.51	14.29	0	100

readability and/or maintainability, such as indentation, redundant code segment, naming convention, coding standard, and programming style, in the inspection records. These readability/maintainability comments account for about 25-35 percent of the statements in inspection records, but are not recorded in the ODC classification.

The results indicate that inspection most often identifies Algorithm, Documentation, and Checking faults. Approximately 90 percent of all the faults belong to these three types, and the distribution between these three types seems to remain relatively constant regardless of when ASA was performed.

4.5.3 System Test Failures

A detailed CR was created for each test and customer-reported failure. Besides the information similar to those in ASA fault reports, dynamic information, such as failure status, fix and submit history, and discussion minutes, was updated frequently during the process of fault removal. A priority rating was also assigned to the failure by the tester or by an agreement between design management and test management. In general, the priority indicates the impact of the failure on the operation of the system. However, sometimes the priority may be elevated if an important customer or many customers are affected. The scale is from 1 to 4, with 1 being the highest priority. Priority 1 means the system will not perform its critical mission and Priority 2 indicates the failure will affect service or will have significant functional impact. The remaining lower priority CRs (Priorities 3 and 4) report the failures that do not impact a release or milestone declaration.

Because only the CR data for Product B contained clear and detailed fix information, we investigated the test failures and customer-reported failures for Product B. The CR data for Product A did not distinguish between system test and customer-reported failures. The CR data for Product C did not provide enough information for distinguishing by ODC. We examined CRs and scrutinized the description of the problem being addressed by the updates and the description of the resulting code fix to classify the failures for Product B. The results of our ODC classification of test failures can be found in Table 10.

Overall, 85 percent of the test failures are of a high priority. *The results indicate that a large majority of test failures are in the Function and Algorithm types*.

Customer-reported failures were classified for Product B as well. The summary of the results is shown in Table 11. Ninety-seven percent of the customer-reported failures are high priority failures.

The results indicate that almost all failures surfaced by customers can be classified as Function or Algorithm defects. However, this phenomenon may be the result of a lack of data in the CR record to more accurately classify the defect and when the defect might have been injected. Function defects are injected in the design phase and can be hard to detect until system testing, when functionality is validated against requirements. Algorithm defects are injected in the low-level phase and had the potential to be found in earlier V&V practices.

The comparison between different fault removal filters is shown in Table 12. *The results indicate that ASA tools predominantly identify two ODC defect types: Checking and*

TABLE 11
Priority Summary of CR Data for Product B (Customer-Reported Failures)

Defect Type	Priority (%)				
	1	2	3	4	Total
Function	24.24	42.42	3.03	0	69.70
Algorithm	6.06	24.24	0	0	30.30
Other ODC types	0	0	0	0	0
Total	30.30	66.67	3.03	0	100

TABLE 12
Mapping of Defects Found by Different Filters to ODC Defect Types

Defect Type	ASA (%)	Inspection (%)	Test (%)	Customer (%)
Function	0	1.09	**55.73**	**69.70**
Assignment	**72.27**	4.37	3.82	0
Interface	0	0.87	0.20	0
Checking	**27.73**	**20.52**	0.80	0
Timing/Serialization	0	0	0	0
Build/Package/Merge	0	1.77	1.81	0
Documentation	0	**35.37**	0	0
Algorithm	0	**36.03**	**37.63**	**30.30**

Assignment. *Approximately 90 percent of all the faults identified by inspection belong to Algorithm, Documentation, and Checking faults*. *A large majority of test/customer-reported failures are in Function and Algorithm types*. Additionally, if ASA is performed prior to inspection (such as was done with Product C.1), fewer Checking faults are identified by the inspection.

4.6 Programmer Errors

- *Q6: What kinds of programmer errors are most often identified via ASA? How often does ASA find these errors?*
- *Metrics: Quantity of ASA faults classified by defect type.*

To avoid the impact of definition difference in defect types among different static analysis tools, data of only one tool was analyzed to answer this question. We chose Flexelint data because Flexelint was the only tool that was used on all three products and more types of faults were identified by Flexelint. We merged the same or very similar static analysis faults for all three products to perform an aggregate analysis of the types of defects identified by the tool. The detailed summary of fault types is shown in the Appendix, ranked with the most frequent faults at the top of the list. While FlexeLint can detect more than 800 defect types, only 33 of these were found in our projects. Severity information was added by the prescreeners. The faults were given one of the following severity ratings based on their potential failure impact.

- **Critical**: A fault that could cause an application core dump, service outage, or system reboot.
- **Major**: A fault that could cause a segmentation fault or performance degradation, such as memory leaks, resource leaks, data corruption.
- **Minor**: A fault that may result in erratic and unexpected behavior, but may have little impact on the system.
- **Coding Standard**: Code that violates a coding standard that has the potential to decrease the maintainability and readability of the software. (Note: No coding standard violations were identified.)

Our results are consistent with the 80-20 rule/Pareto Principle in that a great majority of the faults identified by ASA are produced by a few key types of programmer errors, as shown in Table 13. "Possible use of NULL pointer" is the most often identified fault via ASA, accounting for approximately 46 percent of all faults. About 90 percent of faults are focused on 10 fault types, no matter what level of severity. To improve the code quality in future projects, we can use this information as feedback to programmers so that they pay more attention to these specific types of errors.

A limitation of this analysis is that the screening of the ASA output and the assigning of a severity rating is a manual process and subjective. Different products were screened and evaluated by different screeners. Therefore, the same or a very similar fault might be evaluated as different severity. For example, the fault "Possible Use of Null Pointer" occurred many times in all three products. Most of the faults in this type were assessed as Critical faults in Product A and Product C.1. However, screeners for Product B deemed 72.3 percent of faults in this type were Minor faults and 92.8 percent of faults in this type were considered Major faults in Product C.2.

4.7 Identification of Security Vulnerabilities

- *Q7: Can ASA be used to find programming errors that have the potential to cause security vulnerabilities?*
- *Metrics: Quantity of ASA faults classified by defect type.*

TABLE 13
Pareto Effect in ASA Faults

	% all faults	% critical faults	% major faults	% minor faults
Top 1 fault: Possible use of NULL Pointer	45.92	60.86	37.96	46.32
Top 5 faults: Top 1 fault + Possible Access Out-of-Bounds (Custodial) pointer not freed or returned Memory leak Variable not initialized before using	77.26	85.11	76.56	74.24
Top 10 types: Top 5 faults + Inappropriate deallocation Suspicious use of ; Data overrun Type mismatch with switch expression Control flows into case/default	89.87	90.42	89.42	90.04

TABLE 14
Security Vulnerabilities Identified by ASA

Fault Description	*Explanation*
Possible use of NULL Pointer	Possibly causing the application to crash.
Possible Access Out-of-Bounds	Perhaps on a buffer overflow attack; Possibly causing denial of service.
Suspicious use of ;	Malformed data; Cross site scripting vulnerability.
Type mismatch with switch expression	Possibly causing the application to crash if a user gives a float instead of a Boolean.
Possibly passing a null pointer to function	Possibly causing the application to crash.
Passing null pointer to function	Possibly causing the application to crash.
Possible division by 0	Possibly causing the application to crash; Possibly causing denial of service.
Null Pointer Dereference	Possibly causing the application to crash.
Unrecognized format	Format string vulnerabilities, malformed data.
Wrong Output Message	Cross site scripting vulnerability that can print out information about a system.

Increasingly, static analysis tools are being used to identify security vulnerabilities [6], [7]. Attackers have the ability to exploit the programmer errors to violate a security policy. We highlight the programmer errors that were found by ASA that have the potential to cause security vulnerabilities if proper protections are not in place, as shown in Table 14. These types of programmer errors have been documented as attacks on in SecurityFocus.[9] The ASA tool does not have contextual information and, as such, can produce false positives related to security. For example, cross site scripting is a domain-dependent classification that applies to Web applications and not to the types of embedded systems in our Nortel study.

These results indicate that ASA can be used to find programming errors that have the potential to cause security vulnerabilities.

5 Conclusions

To examine the value of automated static analysis, we analyzed the automated inspection faults, manual inspection faults, and CR data for three large Nortel software products. Our analysis provides some results that can be beneficial to the understanding and utilization of ASA, subject to the limitations discussed in Section 3.3.

- The cost of ASA per detected fault is of the same order of magnitude as the cost of inspections per fault detected, which indicates that ASA is a relatively affordable fault detection technique.
- The defect removal yield of ASA is not significantly different from that of inspections. The defect removal yield of execution-based testing is two to three times higher than that of ASA and, therefore, may be more effective at finding the defects remaining by that phase.
- The number of ASA faults in a module can be a fairly good measure of fault-prone module identification prior to test.
- The mapping of ASI faults to ODC defect types indicated that ASI tools predominantly identify two ODC defect types: Checking and Assignment.
- Approximately 90 percent of all the faults identified by manual inspection belong to Algorithm, Documentation, and Checking faults.

9. http://www.securityfocus.com, owned by Symantec.

TABLE 15
Detailed Classification of Static Analysis Faults Ordered by Total Occurred Time (% of Total Static Analysis Faults Found)

Fault Description	Critical (%)	Major (%)	Minor (%)	Total (%)	ODC Classification
Possible use of NULL Pointer	11.91	14.73	19.28	45.92	*Assignment*
Possible Access Out-of-Bounds	0.49	3.46	6.18	10.13	*Checking*
(Custodial) pointer has not been freed or returned	1.04	6.87	0.20	8.11	*Assignment*
Memory Leak	2.92	3.76	0.79	7.46	*Assignment*
Variable not initialized before using	0.30	0.89	4.45	5.64	*Assignment*
Inappropriate deallocation	0.74	1.88	0.79	3.41	*Assignment*
Suspicious use of ;	0.10	0.35	2.03	2.47	*Checking*
Data Overrun	0.05	0.15	1.93	2.13	*Checking*
Type mismatch with switch expression	0.10	1.93	0.15	2.18	*Checking*
Control flows into case/default	0.05	0.69	1.68	2.42	*Checking*
Possibly passing a null pointer to function	0.35	0.00	1.04	1.38	*Checking*
Ignore return value of function	0.10	0.84	0.40	1.33	*Assignment*
Passing null pointer to function	1.09	0.00	0.00	1.09	*Checking*
Unusual use of a Boolean	0.00	0.54	0.54	1.09	*Checking*
Pointer member neither freed nor zero'ed by destructor	0.00	0.94	0.00	0.94	*Assignment*
Loop not entered	0.00	0.20	0.59	0.79	*Checking*
Unreachable code	0.00	0.30	0.49	0.79	*Checking*
Boolean argument to relational	0.00	0.30	0.05	0.35	*Checking*
Unparenthesized parameter	0.00	0.00	0.35	0.35	*Checking*
Boolean test of assignment	0.00	0.30	0.00	0.30	*Checking*
Possibly negative subscription	0.00	0.25	0.05	0.30	*Checking*
Constant value Boolean	0.00	0.00	0.25	0.25	*Checking*
Boolean within 'String' always evaluates to [True/False]	0.00	0.10	0.10	0.20	*Checking*
Referencing data from already freed pointer	0.20	0.00	0.00	0.20	*Assignment*
Logic Error and Typo	0.05	0.10	0.00	0.15	*Checking*
Possible division by 0	0.00	0.15	0.00	0.15	*Checking*
Non-negative quantity is never less than zero	0.00	0.00	0.10	0.10	*Checking*
Null Pointer Dereference	0.05	0.05	0.00	0.10	*Assignment*
Variable Depends on Order of Evaluation	0.00	0.00	0.10	0.10	*Checking*
Dereferencing a constant string to a pointer	0.05	0.00	0.00	0.05	*Assignment*
Resources not freed	0.00	0.05	0.00	0.05	*Assignment*
Unrecognized format	0.00	0.00	0.05	0.05	*Checking*
Wrong Output Message	0.00	0.00	0.05	0.05	*Checking*
Total	***19.57***	***38.81***	***41.62***	***100.0***	

- A large majority of test/customer-reported failures is in Function and Algorithm types.
- The 80-20 rule/Pareto effect found in faults and failures distribution analysis can be considered as useful feedback to help us improve the code quality in future projects.
- A large percentage of programmer errors detected by ASA have the potential to cause security vulnerabilities.

In conclusion, our results indicate that ASA is an economical complement to other verification and validation techniques.

6 Future Work

In this research, our results focused on the use of three ASA tools for C/C++ programs. We will examine the defect data identified by more static analysis tools for other programming languages besides C/C++. Also, we will enhance the economic analysis by using more data and more refined methods or models considering the severity or impact of the defects. Additionally, the raw output generated by static analysis tools can be examined to find out whether there is an indicator in the raw tool output that could help focus the screening, such as focus on a specific file when a particular set of warnings occur.

Appendix

Detailed classification of static analysis faults ordered by total occurred times (percent of total static analysis faults found) can be found in Table 15.[10]

Acknowledgments

The authors would like to thank the North Carolina State University (NCSU) Software Engineering reading group for their helpful suggestions on this paper. In particular, they would like to thank Michael Gegick for his help on the security aspects of this paper and Kiem Ngo for his help with data collection and for explaining aspects of the Nortel development process. This work was funded in part by a Nortel-funded NCSU Center for Advanced Computing and

10. Note: Static analysis tools assign a probability to certain warnings for certain defect types. For example, defect type "Access of Out-of-bounds" has three different probabilities of warnings (Likely, Possible, and Conceivable). We grouped all these into one type—"Possible Access of Out-of-bounds".

Communications (CACC) enhancement grant. This material is also based upon work supported by the US National Science Foundation (NSF) 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 NSF.

REFERENCES

[1] A.F. Ackerman, L.S. Buchwalk, and F.H. Lewski, "Software Inspections: An Effective Verification Process," *IEEE Software,* vol. 6, no. 3, pp. 31-36, May 1989.

[2] V. Basili, G. Caldiera, and D.H. Rombach, "The Goal Question Metric Paradigm," *Encyclopedia of Software Eng.*, vol. 2, pp. 528-532, 1994.

[3] V.R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sorumgard, and M.V. Zelkowitz, "The Empirical Investigation of Perspective-Based Reading," *Empirical Software Eng. —An Int'l J.,* vol. 1, no. 2, 1996.

[4] B. Beizer, *Software Testing Techniques.* London: Int'l Thompson Computer Press, 1990.

[5] B.W. Boehm, *Software Engineering Economics.* Prentice-Hall, 1981.

[6] B. Chess, "Improving Computer Security Using Extended Static Checking," *Proc. IEEE Symp. Security and Privacy,* pp. 160-173, 2002.

[7] B. Chess and G. McGraw, "Static Analysis for Security," *IEEE Security & Privacy,* vol. 2, no. 6, pp. 76-79, 2004.

[8] R. Chillarege, I.S. Bhandari, J. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.Y. Wong, "Orthogonal Defect Classification—A Concept for In-Process Measurements," *IEEE Trans. Software Eng.,* vol. 18, no. 11, pp. 943-956, Nov. 1992.

[9] C. Differding, B. Hoisl, and C.M. Lott, "Technology Package for the Goal Question Metric Paradigm," Fraunhofer Inst. for Empirical Software Eng. Internal Report 281/96, Apr. 1996.

[10] K.E. Emam and I. Wieczorek, "The Repeatability of Code Defect Classifications," *Proc. Ninth Int'l Symp. Software Reliability Eng.,* p. 322, Nov. 1998.

[11] N.E. Fenton and S.L. Pfleeger, *Software Metrics.* Boston: Int'l Thompson Publishing, 1997.

[12] R. Hochman, T.M. Khoshgoftaar, E.B. Allen, and J.P. Hudepohl, "Using the Genetic Algorithm to Build Optimal Neural Networks for Fault-Prone Module Detection," *Proc. Seventh Int'l Symp. Software Reliability Eng.,* pp. 152-162, 1996.

[13] R. Hochman, T.M. Khoshgoftaar, E.B. Allen, and J.P. Hudepohl, "Evolutionary Neural Networks: A Robust Approach to Software Reliability Problems," *Proc. Eighth Int'l Symp. Software Reliability Eng.,* pp. 13-26, 1997.

[14] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," *Proc. Conf. Object Oriented Programming Systems Languages and Applications (OOSPLA) Companion,* pp. 132-135, 2004.

[15] W.S. Humphrey, *A Discipline for Software Engineering.* Addison Wesley, 1995.

[16] IEEE, "IEEE Standard Glossary of Software Engineering Terminology,"IEEE Standard 610.12-1990, 1990.

[17] IEEE, "IEEE Standard Classification for Software Anomalies," IEEE Standard 1044-1993, 1993.

[18] C. Jones, "Software Defect Removal Efficiency," *Computer,* vol. 29, no. 4, pp. 94-95, Apr. 1996.

[19] C. Jones, *Software Assessments, Benchmarks, and Best Practices.* Addison-Wesley, May 2000.

[20] D. Kelly and T. Shepard, "A Case Study in the Use of Defect Classification in Inspections," *Proc. IBM Centre for Advanced Studies Conf.,* pp. 7-20, 2001.

[21] T.M. Khoshgoftaar, E.B. Allen, J.P. Hudepohl, and S.J. Aud, "Applications of Neural Networks to Software Quality Modeling of a Very Large Telecommunications System," *Trans. Neural Networks,* vol. 8, no. 4, pp. 902-909, 1997.

[22] T.M. Khoshgoftaar, E.B. Allen, J.P. Hudepohl, and W. Jones, "Classification Tree Models of Software Quality over Multiple Releases," *Proc. 10th Int'l Symp. Software Reliability Eng.,* pp. 116-125, 1999.

[23] T.M. Khoshgoftaar, E.B. Allen, K.S. Kalaichelvan, N. Goel, J.P. Hudepohl, and J. Mayrand, "Detection of Fault-Prone Program Modules in a Very Large Telecommunications System," *Proc. Sixth Int'l Symp. Software Reliability Eng.,* pp. 24-33, 1995.

[24] T.M. Khoshgoftaar, E.B. Allen, A. Naik, W. Jones, and J.P. Hudepohl, "Using Classification Trees for Software Quality Models: Lessons Learned," *Int'l J. Software Eng. and Knowledge Eng.,* vol. 9, no. 2, pp. 217-231, 1999.

[25] T.M. Khoshgoftaar and J.C. Munson, "The Lines of Code Metric as a Predictor of Program Faults: A Critical Analysis," *Proc. 14th Computer Software and Applications Conf. (COMPSAC),* pp. 408-413, 1990.

[26] T.M. Khoshgoftaar, J.C. Munson, and D.L. Lanning, "A Comparative Study of Predictive Models for Program Changes During System Testing and Maintenance," *Proc. Ninth Int'l Conf. Software Maintenance,* pp. 72-79, 1993.

[27] N. Kikuchi and T. Kikuno, "Improving the Testing Process by Program Static Analysis," *Proc. Asia-Pacific Software Eng. Conf. (APSEC),* pp. 195-201, 2001.

[28] J.R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S.K. Rajamani, and R. Venkatapathy, "Righting Software," *IEEE Software,* vol. 21, no. 3, pp. 92-100, 2004.

[29] J.C. Munson and T.M. Khoshgoftaar, "Regression Modelling of Software Quality: Empirical Investigation," *Information and Software Technology,* vol. 32, no. 2, pp. 106-114, 1990.

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

[31] N. Nagappan and T. Ball, "Static Analysis Tools as Early Indicators of Pre-Release Defect Density," *Proc. Int'l Conf. Software Eng. (ICSE),* pp. 580-586, 2005.

[32] N. Nagappan, L. Williams, M. Vouk, J. Hudepohl, and W. Snipes, "A Preliminary Investigation of Automated Software Inspection," *Proc. IEEE Int'l Symp. Software Reliability Eng. (ISSRE),* pp. 429-439, 2004.

[33] L. Osterweil, "Integrating the Testing, Analysis, and Debugging of Programs," *Proc. Symp. Software Validation,* 1984.

[34] A.A. Porter and P.M. Johnson, "Assessing Software Review Meetings: Results of a Comparative Analysis of Two Experimental Studies," *IEEE Trans. Software Eng.,* vol. 23, no. 3, pp. 129-145, 1997.

[35] Reasoning Inc. "Automated Software Inspection: A New Approach to Increase Software Quality and Productivity," http://www.reasoning.com/pdf/ASI.pdf, 2003.

[36] I. Rus, F. Shull, and P. Donzelli, "Decision Support for Using Software Inspections," *Proc. 28th Ann. NASA Goddard Software Eng. Workshop,* p. 11, 2003.

[37] N. Rutar, C.B. Almazan, and J.S. Foster, "A Comparison of Bug Finding Tools for Java," *Proc. IEEE Int'l Symp. Software Reliability Eng. (ISSRE),* pp. 245-256, 2004.

[38] G.M. Schneider, J. Martin, and W.T. Tsai, "An Experimental Study of Fault Detection in User Requirements Documents," *ACM Trans. Software Eng. and Methodology,* vol. 1, no. 2, pp. 188-204, Apr. 1992.

[39] G.H. Travassos, F. Shull, M. Fredericks, and V.R. Basili, "Detecting Defects in Object Oriented Designs: Using Reading Techniques to Improve Software Quality," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA),* pp. 47-56, Nov. 1999.

[40] M. Young and R.N. Taylor, "Rethinking the Taxonomy of Fault Detection Techniques," *Proc. Int'l Conf. Software Eng.,* pp. 53-62, 1989.

Jiang Zheng received the BS degree from Fudan University in 1999 and the MS degree from North Carolina State University in 2005. He is a third-year PhD student in the Computer Science Department at North Carolina State University under the supervision of Dr. Laurie Williams. His research interests are software testing, software quality assurance, and software development processes. He is a student member of the IEEE and a member of the ACM.

Laurie Williams received the PhD degree in computer science from the University of Utah, the MBA degree from Duke University, and the BS degree in industrial engineering from Lehigh University. She is an assistant professor in the Computer Science Department of the College of Engineering at North Carolina State University. She leads the Software Engineering Realsearch group and is also is also the director of the North Carolina State University Laboratory for Collaborative System Development located in Engineering Building 2 and the codirector of the NC State eCommerce education initiative. She worked for IBM for nine years in Raleigh, North Carolina, before returning to academia. Her office is located on the NCSU Centennial Campus. Her current research interests include agile software development methodologies and practices, collaborative/pair programming, software reliability, and testing (particularly of secure applications). She is a member of the IEEE.

Nachiappan Nagappan received the BTech degree from the University of Madras in 2001 and the MS and PhD degrees from North Carolina State University in 2002 and 2005, respectively. He is a researcher in the Software Reliability Research Group at Microsoft Research. His research interests include software reliability and measurement, statistical modeling, and defect analysis. He is a member of the IEEE and the ACM.

Will Snipes received the BS degree in computer science from North Carolina State University, Raleigh, and holds a certification for Project Management Professional (PMP) from the Project Management Institute. He joined Nortel Networks in 1992, working in DMS100 Product Test Metrics and Measurements, where he participated in the development of software reliability engineering tools and methods for Nortel Networks' DMS100. Past work includes developing and implementing a system to analyze characteristics of software modules that relate to risk of field defects and predicting high-defect risk areas of code and designing remediation programs to reduce the risk of defects in very large software systems. He is currently working with the Software Dependability Design program in the office of the CTO. His interests include software failure modes analysis, mathematical modeling of software reliability, software metrics, and software failure rate prediction. He is a coauthor of several published works, including papers for *IEEE Communications*, and the International Switching Symposium 1997 (ISS '97). He is a member of the IEEE.

John P. Hudepohl received the BS degree in electronic engineering technology and communication arts from the University of Dayton, Dayton, Ohio, in 1973, and the MS degree in systems management from Florida Institute of Technology, Melbourne, in 1985. He has over 30 years experience in the fields of reliability, quality, and process improvement, managing teams focused on design assurance, performance analysis, software reliability, and tool development with Cincinnati Electronics, ITT, and Nortel. He joined Nortel in 1986, initiated the Enhanced Measures for Early Risk Assessment of Latent Defects (EMERALD) project in 1993, received a patent for Method and System for Dynamic Risk Assessment of Software Systems, and is currently leader of the Software Dependability Design Team in the Chief Technical Office organization. He has been active in the IEEE Communications Society's Network Quality and Reliability Committee, and held the positions of vice chairman, chairman, and chair emeritus.

Mladen A. Vouk received the PhD degree from the King's College, University of London, United Kingdom. He is interim department head and a professor of computer science, and associate vice provost for information technology at North Carolina State University, Raleigh. He has extensive experience in both commercial software production and academic computing. He is the author/coauthor of more than 180 publications. His research and development interests include software engineering, scientific computing, information technology (IT) assisted education, and high-performance networks. He is closely associated with the Computer Science Computer-Based Education Laboratory and the Undergraduate and Graduate Networking Laboratories. He is the cofounder, former codirector, and current member of the Computer Science Software Systems and Engineering Laboratory. He is the founder, former director, and current member of the NC State Multimedia and Networking Laboratory. He is a member of the CENTAUR Labs. He is a member and former Technical Director of the NC State Center for Advanced Computing and Communication. He is an associate graduate faculty member in the Department of Electrical and Computer Engineering at NC State, an affiliated faculty member in the BioMedical Engineering Department, and a member of the NC State E-Commerce Faculty, NC State Information Security Faculty, NC State Genomics Faculty, NC State Bioinformatics Program Faculty, and NC State Operations Research Program Faculty. Dr. Vouk is a member, former chairman, and former secretary of the IFIP Working Group 2.5 on Numerical Software and a recipient of the IFIP Silver Core award. He is an IEEE fellow and a member of the IEEE Reliability, Communications, Computer, and Education Societies, and of the IEEE Technical Committee on Software Engineering. He is a member of the ACM, ASQ, and Sigma Xi. He is an associate editor of the *IEEE Transactions on Reliability*, a member of the editorial board for the *Journal of Computing and Information Technology*, and a member of the editorial board for the *Journal of Parallel and Distributed Computing Practices*.