In-Process Evaluation for Software Inspection and Test

Jarir K. Chaar, *Member, IEEE*, Michael J. Halliday, Inderpal S. Bhandari, *Member, IEEE*, and Ram Chillarege, *Senior Member, IEEE*

Abstract—**The goal of software inspection and test is to reduce the expected cost of software failure over the life of a product. This paper extends the use of *defect triggers*, the events that cause defects to be discovered, to help evaluate the effectiveness of inspections and test scenarios. In the case of inspections, the defect trigger is defined as a set of values that associate the skills of the inspector with the discovered defect. Similarly, for test scenarios, the defect trigger values embody the deferring strategies being used in creating these scenarios.**

The usefulness of triggers in evaluating the effectiveness of software inspections and tests is demonstrated by evaluating the inspection and test activities of some software products. These evaluations are used to point to both deficiencies in inspection and test strategies and to progress made in improving such strategies. The trigger distribution of the entire inspection or test series may then be used to highlight areas for further investigation, with the aim of improving the design, implementation, and test processes.

Index Terms—**Software development process, software inspection, software reliability, software testing.**

I. Introduction

To assist in achieving high quality and increased productivity, a disciplined process for software development must be instituted. This process must detail the activities required to manage the development of a software product, from requirements specification through product maintenance. Associated with an activity are a set of *entry* and *exit* criteria, *tasks* to be performed, and *validation* procedures to verify the quality of the work items produced by these tasks [1]. These activities may be overlapped or repeated when rework is required and are grouped into a set of stages; each stage represents a state of evolution of the product.

Over the years, many researchers attempted to group such stages into software-development process models. In particular, the *waterfall model* [2] and the *spiral model* [3], [4] were widely adopted in software development. Activities, such as *prototyping* [5], were also introduced to assess the tasks to be performed in early stages. Software inspection [6], [7] and software testing [8], [9] were deemed essential to help improve the quality of software products and to reduce the cost of software failure over the life of such products.

The goal of this paper is to present a technique for assessing the effectiveness of inspection and test activities based on a classification scheme of defects. The technique enables in-process feedback to developers by extracting signatures on the development process from such defects [10], [11]. Attributes, such as *defect type* and *defect trigger*, are assigned to each defect. Defect type specifies the actual fix of a defect and defect trigger points to the event that helped detect such a defect. These attributes are key to a more systematic evaluation of the effectiveness of software inspection and test processes. Such processes may involve different software technologies that can also be indirectly assessed.

The paper is organized as follows. Section II introduces software validation, a key activity in improving the quality of software products. Two practical techniques for validating software, software inspection (Section II.A.) and software testing (Section II.B.), are discussed. Next, Section III proposes a technique for evaluating the effectiveness of software inspection and test activities. Subsequently, the trigger set used to evaluate design and code inspections is defined in Section III.A., the trigger set used to evaluate unit and function test scenarios is defined in Section III.B., and the set of defect types of a development process is defined in Section III.C. Section IV discusses the use of the proposed technique in evaluating the inspection process of a software product. The use of this technique in evaluating the test process of another software product is presented in Section V. The paper concludes with Section VI.

II. Software Validation

Validation is a key activity that is essential to checking the correctness of the design and implementation of a software product against some predefined criteria [1], [12]. It aims at finding software defects (design and implementation errors) early in the development process to reduce the costs of removing these defects. These costs have been shown to increase with progress in the software development process: IBM [6], AT&T [13], GTE [14], and TRW [15].

Validation may include the reviews and walk-throughs [16] held by a design team to check that the refinements of accepted requirements are proceeding as desired through each transformation stage. However, the informal nature of such reviews and walk-throughs leaves some doubts about their overall effectiveness and their repeatability [12], [17].

On a more rigorous level, proof of correctness during design refinement offers some help. Proof of correctness is a mathematical method of verifying the logic or function of a program or program segment [18]. While being the

Manuscript received December 1992; revised May 1993. Recommended by F. Bastani.

The authors are with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

IEEE Log Number 9213114.

0098-5589/93$03.00 © 1993 IEEE

most labor-intensive among validation methods, it offers a consistent and repeatable approach. Refined specification or design can always be proven correct and probably defect-free against higher level specification or design. However, this technique has rarely been successfully applied to complex or large systems because of the high degree of effort involved.

Causal analysis [19] is used to identify the root cause of defects. To do so, defects are analyzed, one at a time, by a team knowledgeable in the area. The analysis is qualitative and only limited by the range of human investigative capabilities. Causal analysis can provide feedback to developers at any stage of their software-development process. However, it amounts to an analysis of discovered defects and does not provide guidance for detecting more product defects. Moreover, the resources required to administer this method are significant, although the rewards have proven to be well worth it. Given the qualitative nature of the analysis, the method does not lend itself well to measurement and quantitative analysis.

A more practical means of validating software involves two techniques that can help improve the quality of software products. These are software inspection and software testing. They both allow comprehensive process measurements and analysis, and can help achieve the most significant quality improvements.

A. Software Inspection

Unlike the informal reviews and walk-throughs held by development teams, software inspections are formal evaluations of the work items of a software product [6], [7]. A software inspection is led by an independent moderator with the intended purposes of effectively and efficiently finding defects early in the development process, recording these defects as a basis for analysis and history, and initiating rework to correct such defects. Reworked items are subsequently reinspected to ensure their quality.

Software developers can literally remove a part from the development line, rework it at the most appropriate time in the process, and replace it in the development line. Hence, inspections ensure that a higher level of quality is shipped to the testers and ultimately to the users of a software product. They can help evaluate the activities of the following stages of the software development process:

1) *Requirements Specification:* During this stage, a specific methodology is used to gather the requirements of a product. These requirements are analyzed, documented in a *requirements document*, and used to create a solution that describes new and enhanced functionality in response to each requirement. The requirements document is validated via inspection.

2) *High Level Design* (HLD): HLD defines the product functions that will satisfy the specified requirements. These functions are then partitioned into substructures with hierarchical relationships. Such substructures emulate the activities in a function-oriented design or the agents in an object-oriented design. Component names are defined, together with their principal data structures and control paths.

   Prototyping may be employed in this stage to assist in exploring various design approaches and to validate the capability of the product to meet the specified requirements.

   The details of the high-level design of a software product are documented in a *design specification document.* This document contains a complete and detailed description of all the externals of the product. The HLD stage is completed when the design specification document has been validated, via inspection, against the requirements document of the product.

3) *Low Level Design* (LLD): During LLD, the HLD substructures of a software product are further refined into procedures and their logic paths detailed. Data structures are defined to the lowest level of detail.

   The completed design of a software product is documented in a *design structures document.* This document details the structural and logical aspects of the product by describing the components and modules, their functions, data usage, and interfaces. The LLD stage is completed when the design structure document has been validated, via inspection, against the requirements and design specification documents of the product.

4) *Implementation:* The completed design of the software product is transformed into a compilable language. The source code, individual test cases, and test scenarios are then inspected prior to software testing.

Software inspections provide a powerful way to improve the quality and productivity of the software process [12]. Their basic objectives are to find errors at the earliest possible point in the development cycle, to ensure that the appropriate parties technically agree on the work, to verify that the work meets predefined criteria, to formally complete a technical task, and to provide data on the product and the inspection process. They are cost effective because they allow errors to be found early in the software process when the cost of making the corrections is far less [6], [13]–[15].

Many attempts to evaluate the effectiveness of software inspections and prove their usefulness have been reported. These attempts looked at the inspection data of various software products and inferred some general observations from these data. They are based on the concepts of statistical defect modeling [20]–[23].

Wenneson used statistics to develop some guidelines for conducting software inspections [24]. In particular, he observed that defect density, i.e., the number of defects found per thousand lines of code (Defects/KLOC) declines with increasing inspection rates (KLOC/hour). Further, this decline tails off for high inspection rates. Plots of defect density versus inspection rate for the releases of a product are used to estimate such high inspection rates.

Wenneson also attempted to quantify the relationship between an inspected line of design (LOD) and the equivalent number of lines of code (LOC) in a software project. This quantification extends the applicability of the above results to cover both design and code inspections.

In a separate study, Schulmeyer and McManus also used statistical techniques to analyze design and code inspections data [25]. For each component of a software product, the inspection rate (KLOC/hour), the defects found per inspection hour (Defects/hour), the defect density (Defects/KLOC), and the (hours of preparation)/(hour of inspection) ratio were computed. Plots of the control charts of each parameter were then generated and their values were correlated. Components with the lowest inspection rates, the highest number of defects found per inspection hour, the highest defect density, and the highest preparation rates were deemed troublesome.

A recent analysis of defect densities found during software inspections was reported by the Jet Propulsion Laboratory (JPL) of the California Institute of Technology [26]. The results showed a significantly higher density of defects during requirements inspections. Progress through design and code inspections was also characterized by an exponential decrease in defect densities because defects were fixed when detected and did not migrate to subsequent stages. The study also found that increasing the inspection rate resulted in a decrease in the density of defects found.

B. Software Testing

Testing is the process of executing the code of a software product with the intention of finding defects [8], [9]. Reliability measures, such as *interfail times* [27], can be used to track progress during the test. Testing can also help establish a specific reliability level, such as a predetermined mean time to failure (MTTF) [27], for a software product. Software testing can help evaluate the activities of the following stages of the software development process:

1) *Unit Testing* (UT): The inspected source code of the software product is unit tested. Unit tests are essentially path tests done on a white box basis.

   White box tests examine the basic design of the program and require that the tester have detailed knowledge of the program's internal structure. The logic of each module is tested to ensure that a high percentage of statements are executed at least once, and similarly, a high percentage of branches are traversed at least once. White box tests focus on a relatively small segment of code and aim to exercise a high percentage of the internal paths of this code.
2) *Function Verification Testing* (FVT): Functional or black box tests are designed to exercise the program to its external specifications. The product is integrated and all documented product functions are executed to test for conformance with design specifications. Because exhaustive black box testing may not be achievable, these tests check the most likely scenarios of a software product.
3) *System Verification Testing* (SVT): An integrated hardware and software system that emulates the user environments is built, and all documented product functions are executed from the user perspective. Further, the system is stressed from the performance, reliability, availability, and capability viewpoints. When this test is complete, the product will successfully perform to the requirements and from the user perspective.

Software testing presents a problem in economics. With large systems, it is almost always true that more tests will find more defects. The question is not whether all the defects have been found but whether the cost of discovering the remaining defects can be justified. This trade off should consider the probability of finding more defects in test, the marginal cost of doing so, the probability of the users encountering the remaining defects, and the impact of these remaining defects on the users. Unfortunately, the general lack of data on the software process prevents making intelligent trade-off decisions.

Statistical usage testing attempts to make such a trade-off decision by testing software the way users intend to use it [28]. First, the usage probability distributions of a software product are specified. They define all possible usage patterns and scenarios, including erroneous and unexpected usage, together with their probabilities of occurrence. Test cases are then derived from the distributions, such that every test represents actual usage and will effectively rehearse user experience with the product. Next, each test case is executed and its results are verified against system specifications. As a result, errors left behind at the completion of statistical usage testing tend to be those infrequently encountered by users.

As part of the test reporting process, it is desirable to have standard definitions for defect severity. Because defects in software testing are always associated with system failures, defect severity attempts to capture the impact of such failures. At IBM, a severity in the range 1–4 is assigned to test defects, with the most severe problem assigned a severity of 1. A ten-level priority scheme has also been proposed by Beizer [29]. This scheme is more detailed than is generally appropriate for defect reporting. However, it provides a valuable framework for judging defect severity.

Mills proposed *error seeding* as a technique for measuring the efficiency of tests [30]. This technique purposely injects known defects into a program and then notes the percentage of such defects that are found during testing. The implication is that the percentage of these injected defects found in testing provides a reasonable indication of the efficiency of the tests at finding the other defects as well. While there have been doubts about the validity of this approach, Knight and Amman have reported experimental results that support the concept [31]. The method should thus be considered, at least for experimental use.

Card used statistical measures such as *testing efficiency, delivered error rate*, and *total defect rate* to evaluate whether a product release is under or out of statistical control [32]. Testing efficiency is defined as the percentage of all defects found during system verification testing (SVT). In effect, it measures the quality of the testing activity. Delivered error rate is defined as the percentage of noncosmetic defects found during operation. Total defect rate is defined as the percentage of all noncosmetic defects reported during SVT and field operation. The control limits of each product release are computed as the average of the total defect rates of all releases of this product. A new release is then judged out of control if its total defect rate is greater than the control limit, and it is deemed under statistical control otherwise.

III. Proposed Technique

In the absence of feasible and cost-effective theoretical methods for verifying the correctness of software designs and implementations, software inspection and test play a vital role in validating both. The goal of both inspections and tests is to reduce the cost of software failure over the life of a product. The techniques of Sections II.A. and II.B. can provide good evaluations of their effectiveness. However, these techniques do not offer software designers, developers, and test planners any significant guidance for rectifying, in-process, the weaknesses of their procedures, and for assessing the implications of any rectifying actions on their inspection and test processes.

This paper presents a technique for assessing the effectiveness of inspection and test activities based on Orthogonal Defect Classification (ODC) [11]. ODC enables in-process feedback to developers by extracting signatures on the development process from such defects [10], [11]. A wide range of reported benefits has resulted from applying this technique to software development. Benefits as little as five person-days for a five-KLOC project and as high as 848 person-days for a 100-KLOC project were documented [33].

In this proposed technique, attributes, such as *defect type* and *defect trigger*, are assigned to each defect. Defect type specifies the actual fix of a defect [34] and defect trigger points to the event that helped detect such defect [35]. These attributes are key to a more systematic evaluation of the effectiveness of software inspection and test processes. Such processes may involve different software technologies that can also be indirectly assessed.

The set of values of an attribute is derived by iterative refinement. Starting with an initial set of attribute values, the values of the attributes of a defect found during inspection or test are picked from the elements of this set. When all the elements of the current set of attribute values fail to provide a value appropriate for this defect, a new value is added to the current set. All defects are then reclassified using the new, augmented set of attribute values. Only extensive classification of the defects of numerous software products can prove the stability of the latest set of attribute values. These values should offer some consistency between the stages of the software development process and should not depend on the specifics of a software product or a software organization.

The technique consists of three major activities: data gathering, data classification, and data interpretation. The data-gathering activity has the aim of collecting and recording the defects of a design inspection, a code inspection, or a test execution. This is followed by the data-classification activity. During this latter activity, the defect type and defect trigger of each defect are specified and checked. The trigger is decided by the design inspector, the code inspector, or the test scenario planner, and the defect type is determined by the software designer or the programmer correcting the defect. The classified data are then used in the data interpretation activity.

Data interpretation includes evaluating an inspection or a test activity and tracking progress between the various activities of a stage and between the various stages of the software-development process. The findings of such evaluations report both the strengths and the weaknesses of an inspection or a test activity and are presented to the software-development team. Reported strengths signal the start of the next activity, while reported weaknesses are followed by specific actions that aim at improving the outcome of current activity. Such actions may result from performing causal analysis on a small subset of high-impact defects identified during data interpretation.

To evaluate the outcome of an inspection or test activity, both expected and observed percentages of the values of defect type, defect trigger, and the cross-product (defect type, defect trigger) are computed. The percentage expected is estimated based on historical data, when available [36]. In the absence of such historical data, the percentage expected for the values of a single attribute is estimated, *a priori*, by the development team of a software product and is based on their intuitive knowledge of current progress in development. Further, it is assumed that the defect type and defect-trigger attributes are statistically independent. Hence, the percentage expected for the values of the cross-product (defect type, defect trigger) is the product of the expected percentages of the corresponding defect type and defect-trigger values. Differences between observed and expected percentages form the basis for evaluating an inspection or a test activity. Likewise, differences between the observed percentages of similar charts are used to evaluate the progress of successive inspection and test activities.

A. Trigger in Design/Code Inspection Stages

Design and code inspection includes those stages in which high-level design (HLD) documents (e.g., design specification documents), low-level design (LLD) documents (e.g., design structures documents), or code are inspected. During such inspections, a trigger describes the event that helps an inspector detect a defect of the design document or the code segment. Only one such trigger may be chosen for any given defect.

The set of design/code inspection triggers that follow has been derived by considering the activities performed by different inspectors in accomplishing their task. Defects found from these triggers can potentially be identified by the inspector of a design document or a code segment. Hence, triggers are linked to the level of skills of the inspection team, and their distribution can help evaluate the inspection process. They are defined as:

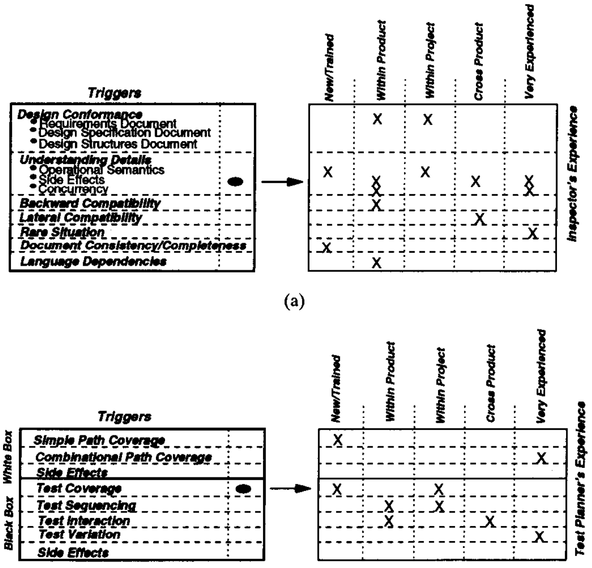
1) *Design Conformance:* The document reviewer or the code inspector detects the defect while comparing the design element or code segment being inspected with its specification in the preceding stage(s).
2) *Understanding Details:* The inspector detects the defect while trying to understand the details of the structure and/or operation of a component. This trigger may be further refined into:

   a. *Operational Semantics:* The inspector had in mind the flow of logic required to implement the function when the defect was noticed.

Coverage	*Sequencing*	*Interaction*	*Variation*

create(file, r)
modify(file)
delete(file)
	create(file_1, rw)
	create(file_2, rw)
	modify(file_2)
	modify(file_1)
	delete(file_2)
	delete(file_1)
		create(file, r)
		delete(file)
		modify(file)
			create(file, r)
			create(file, w)
			create(file, rw)

Fig. 1. Sample black box test cases.



(a)

(b)

Fig. 2. Level of experience associated with triggers.

b. *Side Effects:* The additional effect or side effect of some documented or some implemented action was under review when the defect was discovered.

c. *Concurrency:* The inspector was considering the serialization necessary for controlling a shared resource when the defect was discovered.

3) *Backward Compatibility:* The inspector used extensive product experience to determine an incompatibility between the functionality described by the design document or the code and that of earlier versions of the same product.

4) *Lateral Compatibility:* The inspector with broad-based experience detected an incompatibility between the functionality described by the design document or the code and the other (sub)systems and services with which it must interface.

5) *Rare Situation:* The inspector used extensive experience or product knowledge to foresee some system behavior not considered or addressed by the documented design or code under review.

6) *Document Consistency/Completeness:* The defect surfaces because of some inconsistency or incompleteness within the document or code.

7) *Language Dependencies:* The developer detects the defect while checking the language-specific details of the implementation of a component or a function.

B. Trigger in Unit/Function Testing Stages

During the unit/function testing stages, a trigger captures the intent behind creating the test case which, when executed, uncovered the defect and can therefore potentially be identified by the designer of a test scenario. To choose the right trigger, the test designer must decide if the test case that found the defect was written with a black box or white box model in mind.

In white box testing, the tester must be familiar with the internals of a body of code. This is in contrast with black box testing where the tester relies upon the ability to invoke the execution of one or more bodies of code, where little is known of the internals. In Function Test, a body of code would implement an externally invoked function, whereas in Unit Test, a body of code could be a single module. Sample black box test cases of a file-management system are presented in Fig. 1.

White Box Triggers:

1) *Simple Path Coverage:* The test case that found the defect was created by the tester with the specific intention of exercising branches in the code. In other words, the test case was motivated by knowledge of specific branches in the code and not by knowledge of module functionality that could be invoked by a caller external to the module. Furthermore, a branch targeted for execution by the tester was exercised only once by the test case.

2) *Combinational Path Coverage:* Same as Simple Path Coverage, *except* that branches targeted for execution by the tester were exercised more than once by the test case. In other words, the tester attempted to invoke the execution of these branches under several different conditions. In contrast, the Simple Path Coverage case above only attempts to cover the branches and does not exercise them under different conditions.

3) *Side Effects:* The defect surfaced because of some unanticipated behavior not specifically tested for.

Black Box Triggers:

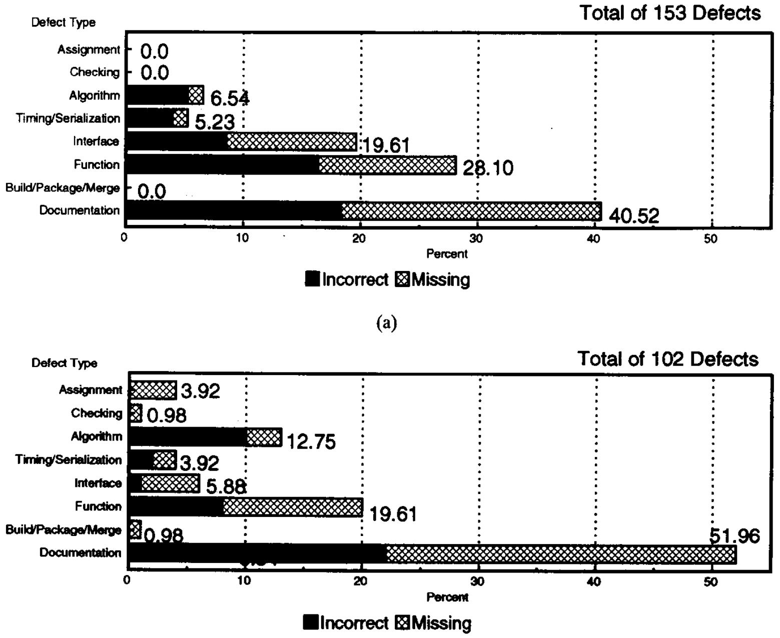
1) *Test Coverage:* The test case that found the defect was a straightforward attempt to exercise a single body of code using a single input. An input is a single combination of parameter values. Note that this test case does not link the executions of bodies of code as in Test Sequencing or Test Interaction. Instead, it is an obvious test case of a body of code.

2) *Test Sequencing:* The test case that found the defect executed, in sequence, two or more bodies of code each of which can be invoked independently by the tester.

3) *Test Interaction:* The test case that found the defect initiated an interaction between two or more bodies of code each of which can be invoked independently by the tester. The interaction was more involved than a simple sequence of the executions.

4) *Test Variation:* The test case that found the defect was a straightforward attempt to exercise a single body of code using different inputs. An input is a single combination of parameter values.

5) *Side Effects:* The defect surfaced because of some unanticipated behavior not specifically tested for.



(a)

(b)

Fig. 3. Defect type distributions for two high level design inspections.

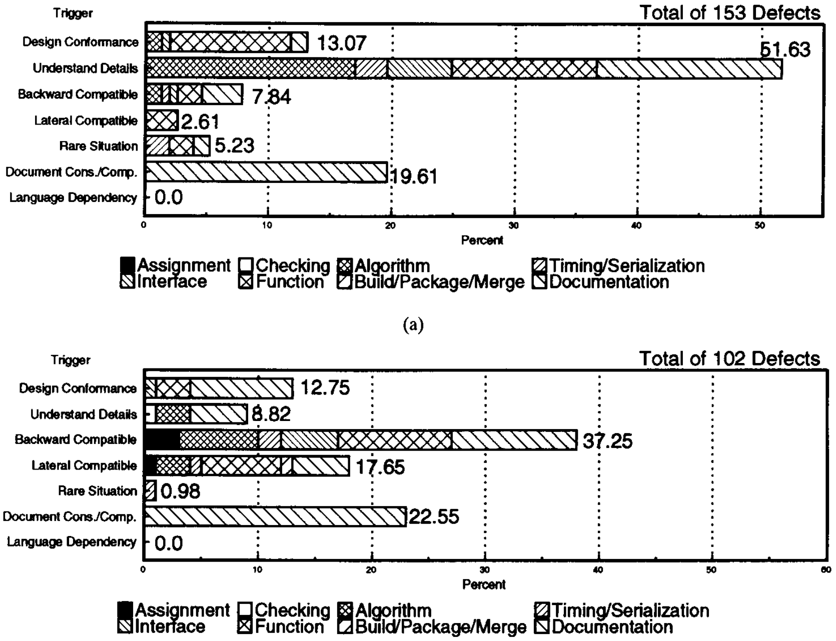
C. Defect Type

Defect type describes a software fix and is therefore usually chosen by the programmer making the correction. In other words, the selection is implied by the eventual correction. A distinction is made between an error of omission (Missing) and an error of commission (Incorrect). Only one defect type can be selected, and it must be further classified as either Missing or Incorrect. Defect type is linked to the software development paradigm used by the developers of a team. Hence, the defect types of the function-oriented and the object-oriented design and programming paradigms differ. The defect types of a typical software development process that involves function-oriented design and programming are defined as follows:

1) *Assignment:* Value(s) assigned incorrectly or not assigned at all.
2) *Checking:* Errors caused by missing or incorrect validation of parameters or data in conditional statements.
3) *Algorithm:* Efficiency or correctness problems that can be fixed by (re)implementing an algorithm or a local data structure without the need for requesting a design change. A fix involving multiple *assignment* or *checking* corrections may be of type algorithm.
4) *Timing/Serialization:* Necessary serialization of a shared resource is missing, the wrong resource is serialized, or the wrong serialization technique is employed.
5) *Interface:* Communication problem between modules, components, and device drivers via macros, call statements, control blocks, or parameter lists.
6) *Function:* The error should require a formal design change, as it affects significant capability, end-user interfaces, product interfaces, interfaces with hardware architecture, or global data structure(s).
7) *Build/Package/Merge:* Problems encountered during the driver build process, in library systems, or with management of change or version control.
8) *Documentation:* The problem is with the written description contained in user guides, installation manuals, prologs, and code comments. This is not to be confused with errors in documented design that might be classified as a *function* or an *interface* defect type.

IV. Inspection Process Evaluation

The concept of the trigger fits very well into assessing the effectiveness and eventually the completeness of a design or a code inspection. In such an inspection, the requirements document that specifies the product requirements and the design specification document that defines the functionality of this product are reviewed by an independent team of software planners, designers, and developers. Moreover, the design structures document that describes the implementation details of the product and the code that implements the product are reviewed by the members of the development team. A critical part of this inspection process is to assess whether such documents have been reviewed by enough people with the right skill level. The importance of such assessment cannot be understated because the process that follows design and code inspections tests the product implementation. Hence, any missing or incorrect information will have a serious impact on testing and maintaining this product.



(a)

(b)

Fig. 4. Trigger distributions for two high level design inspections.

For each design/code inspection trigger, the skill required by the inspector can be assessed. Fig. 2(a) shows the skill level appropriate for each trigger. Note that some of the triggers, such as checking for the consistency and/or completeness of a document, may not require substantial knowledge or experience of the subject product, whereas lateral compatibility clearly indicates the need for people with knowledge of more than just the product under inspection. Similarly, backward compatibility requires people with experience within the product. People who can identify rare situations need a lot of experience, both with the product and otherwise.

Given that defect triggers can be mapped to skills required to find the defect, the defect trigger distribution can help gain insight into the effectiveness of an inspection. It is common to also have several inspections of a design document or a code segment, each incorporating the accepted comments from earlier ones. Thus, the change in the trigger distribution may be tracked to verify if it reflects anticipated trends.

Evaluations of the design specification document, the design structures document, and the implementation code of a software product are detailed next.

A. Design Inspection Evaluation

The usefulness of defect classification in evaluating design inspections is illustrated by presenting the results of inspecting the design specification document of a software product. The document describes the functionality of the product and was inspected by a team of independent software engineers. The technique proposed in this paper was used to evaluate the first inspection of this document and pointed to the need for a second inspection of the same design document. A total of 255 defects were detected during these two inspections; 153 defects (60%) were detected and fixed in the original design document during the first inspection and an additional 102 defects (40%) were detected and fixed in the updated design document during the second inspection. Fig. 3 shows the defect type distributions of both inspections and Fig. 4 shows the distributions of the triggers that helped detect these defects. By suggesting a second round of inspections that uncovered 40% of the defects, the proposed technique helped reduce the cost of fixing these defects in subsequent stages of development of this product.

In examining the distribution of the defect types in Fig. 3(a), it was noted that the percentages of documentation (40.52%), function (28.10%), and interface (19.61%) defects were dominant. Hence, the design document itself was considered incomplete and inconsistent, and there were significant problems related to function and interface. But would an additional inspection at this stage have been cost-effective in terms of additional defects uncovered for the effort expended? And what aspects of the design should be focused on to yield the most defects?

To answer these questions, the distribution of defect trigger from the first inspection (Fig. 4(a)) was considered. The defects discovered by backward compatibility (7.84%), rare situation (5.23%), and lateral compatibility (2.61%), were disproportionately low when compared to understanding details (51.63%) and document consistency/completeness (19.61%). The team also concluded that design conformance (13.07%) was as expected.

While uncovering a high percentage of function, interface, and algorithm defects (54.25%) is explainable at this stage of

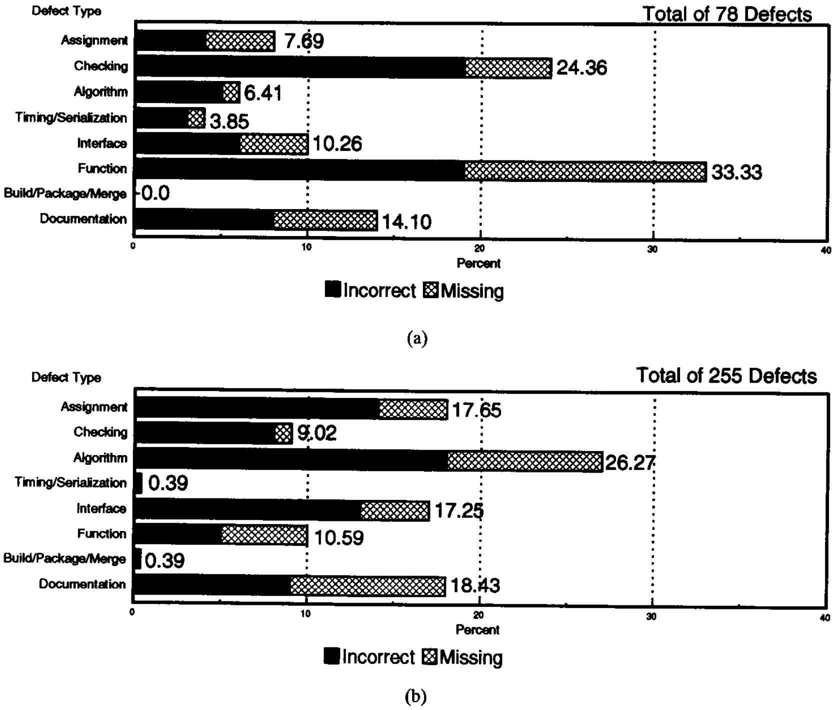


Fig. 5. Defect type distribution for two code inspections.

the design, the small percentage of such defects triggered by lateral compatibility and backward compatibility was cause for concern; (backward compatibility, function) is only 1.97% of the distribution in Fig. 4(a), (backward compatibility, interface) is 1.57%, (backward compatibility, algorithm) is 1.0%, (rare situation, function) is 1.4%, and (lateral compatibility, function) is 2.61%. Given that this product was improving the functionality of a previous release by significantly enhancing its interactions with other products, the team concluded that either their design was excellent or their inspection was deficient.

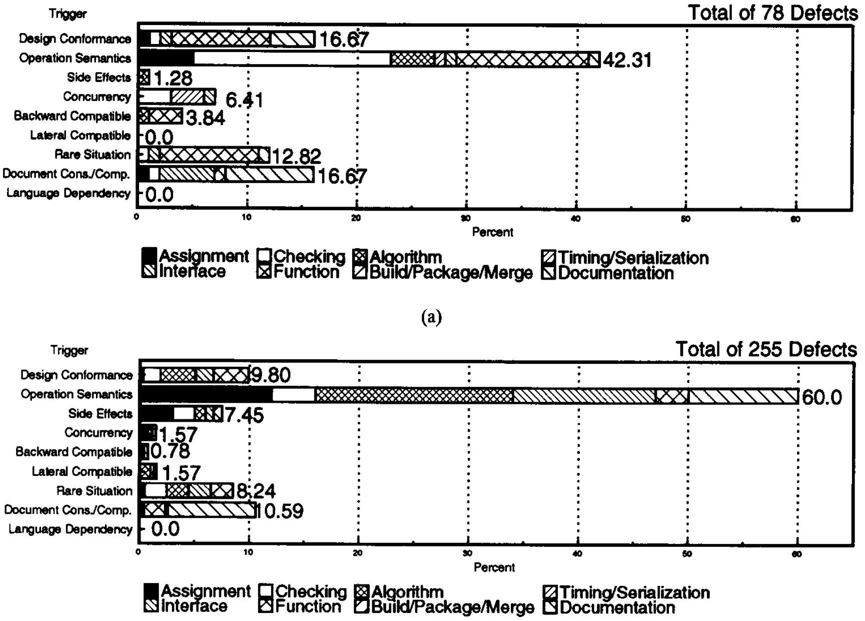
In looking back to the triggers associated with level of experience (Fig. 2(a)), it became obvious that the defects so far discovered had been discovered by reviewers with, or using, little experience in this or other related software and hardware products. Therefore, the development team decided that initiating a second inspection of the *updated* design specification document using a team of very experienced software engineers would be cost effective. To validate this decision, the defects from the second design inspection were classified and analyzed. Fig. 3(b) shows the distribution of defect type for this second inspection and Fig. 4(b) shows the distribution of its triggers.

In this second inspection, the percentages of documentation (51.96%), function (19.61%), and algorithm (12.75%) defects were high, and the percentage of interface (5.88%) defects was low. Most documentation, function, and interface defects were missing while most algorithm defects were incorrect. Furthermore, a drop in the overall percentage of both missing and incorrect function, interface, and algorithm defects (16.01%) was reported. This drop pointed to a more stable design. However, an increase in both missing and incorrect documentation defects (11.44%) implied that the design document may still have been incomplete and inconsistent.

On the other hand, the distribution of triggers that uncovered the additional defects (Fig. 4(b)) changed remarkably in terms of backward compatibility (37.25%) and lateral compatibility (17.65%). The percentage increases of (backward compatibility, function) to 10%, (backward compatibility, interface) to 5%, (backward compatibility, algorithm) to 7%, (lateral compatibility, function) to 7%, and (lateral compatibility, algorithm) to 3%, were further proof of the effectiveness of this second inspection. As a result, the team felt more confident in proceeding to the next stage of the development process.

B. Code Inspection Evaluation

The usefulness of defect classification in evaluating code inspections is illustrated by presenting the results of inspecting the actual code of the previous software product. Such an inspection was carried out by the development team of this product. The technique proposed in this paper was used to evaluate the first code inspection and pointed to the need for a second inspection of this code. A total of 333 defects were detected during these two inspections; 78 defects (23.4%) were fixed following the first inspection and an additional 255 defects (76.6%) were detected by the second inspection. Fig. 5 shows the defect type distributions of both inspections and



(a)

(b)

Fig. 6. Trigger distribution for two code inspections.

Fig. 6 shows the distributions of the triggers that helped detect these defects. By suggesting a second round of inspections that uncovered 76.6% of the defects, the proposed technique helped reduce the cost of fixing these defects in subsequent test stages.

The first inspection (Fig. 5(a)) resulted in a relatively high percentage of checking (24.36%) and documentation (14.10%) defects, and relatively low percentages of interface (10.26%), assignment (7.69%), and algorithm (6.41%) defects. Most defect types were incorrect. However, it was noted that the proportion of function defects (33.33%) was significantly high for a code inspection, when the functionality is already implemented. Further, 43% of these function defects were missing and would require the insertion of new logic to resolve, instead of simply fixing in place. After examining the distribution of defect triggers (Fig. 6(a)), which was dominated by operational semantics (42.31%), the team decided that this may have been a cursory inspection because of the relatively low percentages of concurrency (6.41%), backward compatibility (3.84%), and side effects (1.28%), and the lack of lateral compatibility and language dependencies.

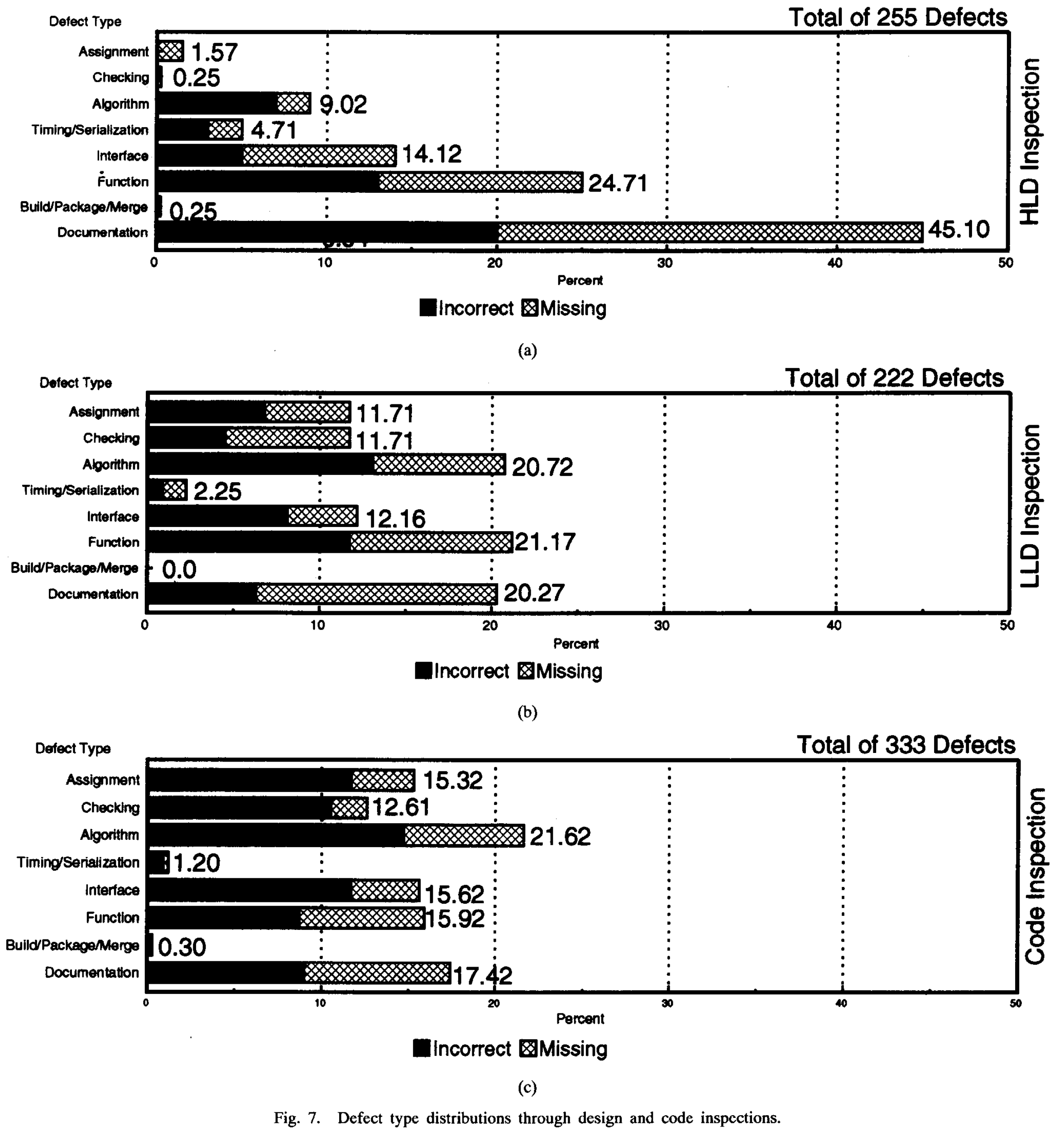
The small percentage of defects triggered by rare situation, concurrency, and side effects is cause for concern; (rare situation, checking) is only 1% of the distribution in Fig. 6(a), (rare situation, interface) is 1%, (concurrency, checking) is 2.4%, (concurrency, interface) is 1.41%, (concurrency, timing/serialization) is 2.6%, and (side effects, algorithm) is 1.28%. Consequently, the team reinspected their code with emphasis placed on functionality and discovered an additional 255 defects. Fig. 5(b) shows the defect-type distribution of this second inspection and Fig. 6(b) shows its trigger distribution.

In this second inspection, the team was surprised by the relatively low percentage of function (10.59%). The defects were more evenly distributed with algorithm (26.27%), assignment (17.65%), and interface (17.25%) being the most significantly increased over the first inspection. They concluded that the extent of function problems during the first inspection had inhibited the detection of the more usual defects discovered during a code inspection. Further, had they simply fixed the function defects and proceeded to the next development process stage, many defects would have escaped into subsequent test stages.

By checking Fig. 6(b), it can be concluded that the second inspection is marked by an increase in the percentage of operational semantics (60.0%). Furthermore, the percentage increases of the combinations of (rare situation, checking) to 2%, (rare situation, interface) to 2%, (rare situation, algorithm) to 2%, (rare situation, assignment) to 0.63%, (concurrency, assignment) to 0.9%, (side effects, assignment) to 3%, (side effects, checking) to 2%, (side effects, algorithm) to 1%, (side effects, interface) to 0.57%, and (side effects, function) to 0.95% indicate a more thorough inspection.

C. Progress Evaluation

The usefulness of defect classification in evaluating progress through successive stages of the development process is illustrated by presenting the results of inspecting the design specification document, the design structures document, and the actual code of the previous software product. Such inspections are carried out during HLD, LLD, and code implementation, respectively. The technique proposed in this paper has been used to evaluate progress through these stages. A total of



(a)

(b)

(c)

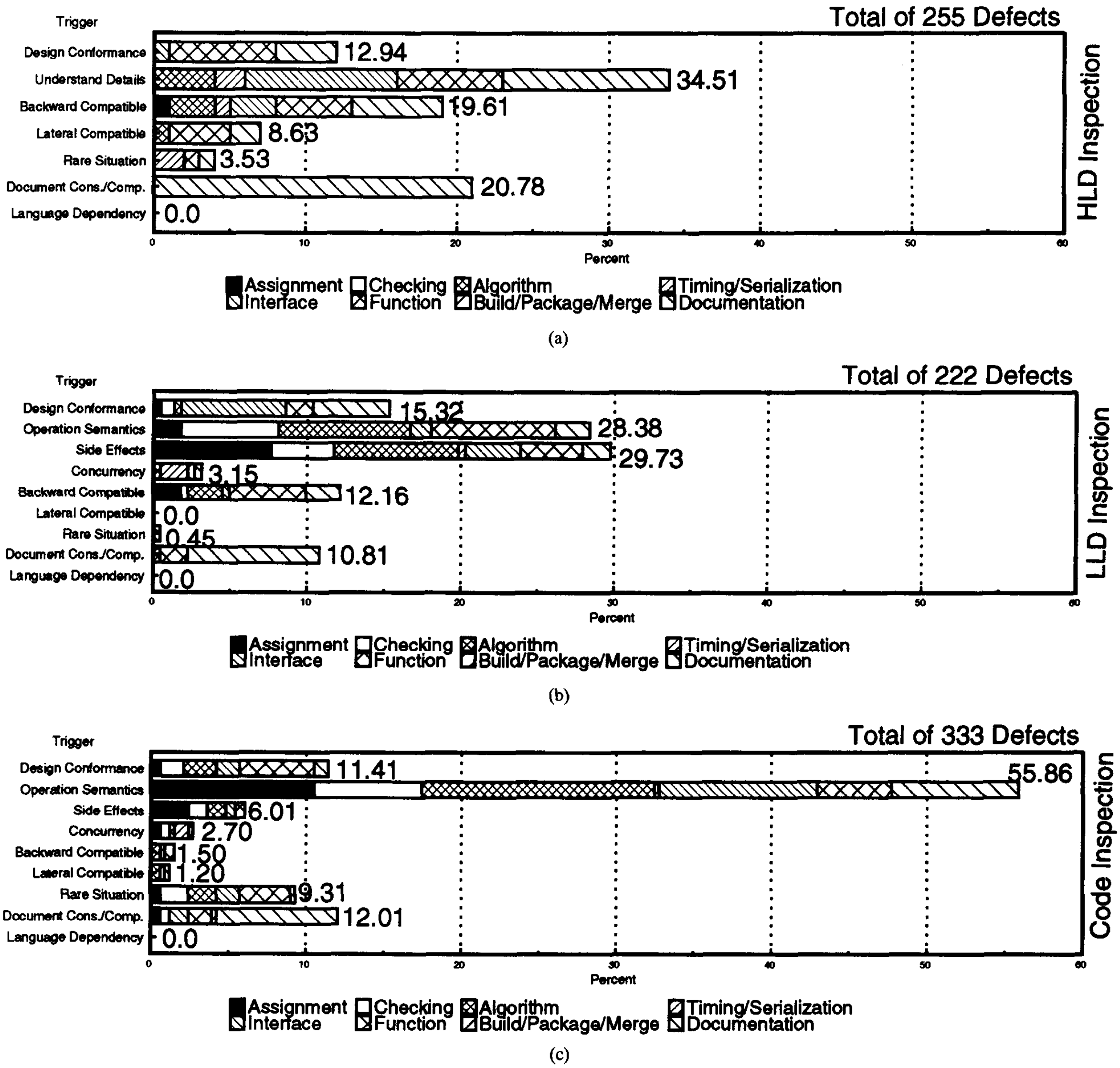
Fig. 7. Defect type distributions through design and code inspections.

255 (31.48%) defects were found by inspecting the design specification document, 222 (27.4%) defects were found by inspecting the design structures document, and 333 (41.12%) defects were found during code inspection. Fig. 7 shows the defect type distributions of these inspections and Fig. 8 shows their trigger distributions.

The diminishing trend of defect type function across all three stages was initially encouraging (Fig. 7). However, the proportion of missing function remained constant instead of decreasing with respect to incorrect function. A closer examination of the (function, rare situation) tuple (Fig. 8) indicated that a significant proportion of these function defects were being triggered by rare situation. Such defects may be hard to correct and were cause for concern.

Interface and algorithm defects increased between design structures document inspection and code inspection. They were mostly incorrect, were triggered by operational semantics, and would probably have been fairly easily corrected. However, the large proportion of side effects triggers in design structures document (Fig. 8(b)) may have been inhibiting the discovery of more interface and algorithm defects.

The overall decline of lateral compatibility and backward compatibility was perceived as a progressive trend (Fig. 8). However, the significant percentages of side effects (29.73%)



(a)

(b)

(c)

Fig. 8. Trigger distributions through design and code inspections.

in design structures inspection and rare situation (9.31%) in code inspection were cause for concern.

Based on the absence of a clear assessment of the progress, the team decided on a schedule adjustment. Components that revealed function, interface, or algorithm defects, that were triggered by side effects and rare situation during document structures and code inspections, were targeted for further design structures inspection. The remaining components were allowed to progress to unit test. Furthermore, the team decided to hold more frequent analyses of classified data during FVT to further evaluate the progress of testing on a component basis.

V. Test Process Evaluation

The concept of the trigger also fits very well into assessing the effectiveness and eventually the completeness of a test scenario. In such a scenario, test cases are created that cover all logic paths in a module or examine whether the implemented product conforms with its external specification. A critical part of this test process is to assess whether the implemented product has been adequately tested before customer use. Hence, capturing the intent behind creating the test cases, or determining the trigger, is the key to improving the overall effectiveness of the scenarios that test the functionality of the product. The importance of such assessment cannot be understated because a deficient test strategy might deliver a product with a large number of latent defects. If such defects are found by the customers, the product may be perceived as having low quality.

For such triggers, the skill required by the test planner can be assessed. Fig. 2(b) shows the skill level appropriate for

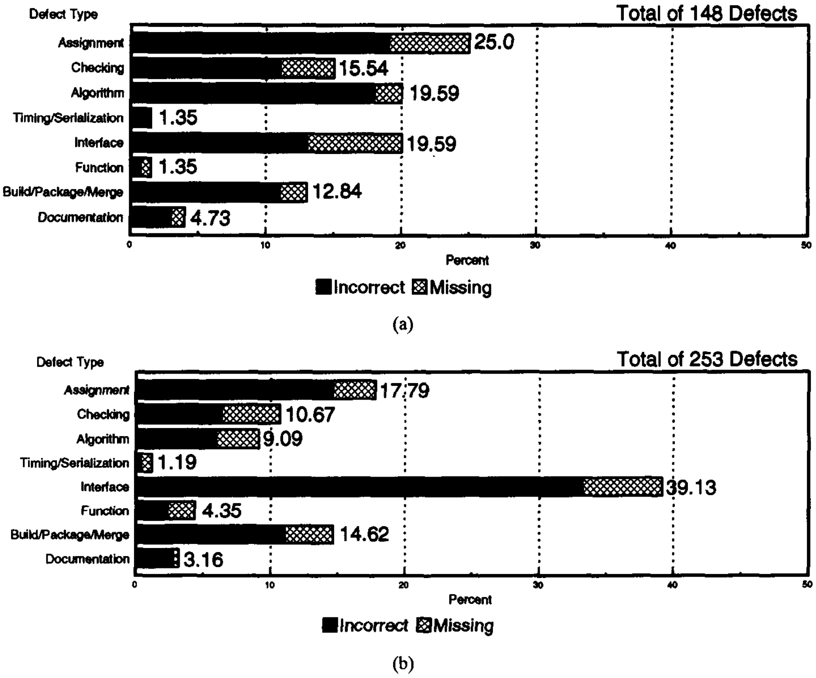


Fig. 9. Defect type distribution in two function verification test scenarios.

each trigger. Note that some of the triggers, such as simple path coverage in white box testing, may not require substantial knowledge or experience of the subject product, whereas test interaction in black box testing clearly indicates the need for people with skill in more than just the product under test. Similarly, both test sequencing and test interaction of black box testing require people with experience within the product. People who can generate test cases to achieve combinational path coverage in white box testing and test variation in black box testing need a lot of experience, both with the product and otherwise.

On the other hand, side effects are inadvertent events that can occur during both modes of testing. Hence, side effects are not associated with a particular level of experience. When side effects are the trigger for a significant number of defects in a test scenario, their presence may indicate a lack of stability in the test.

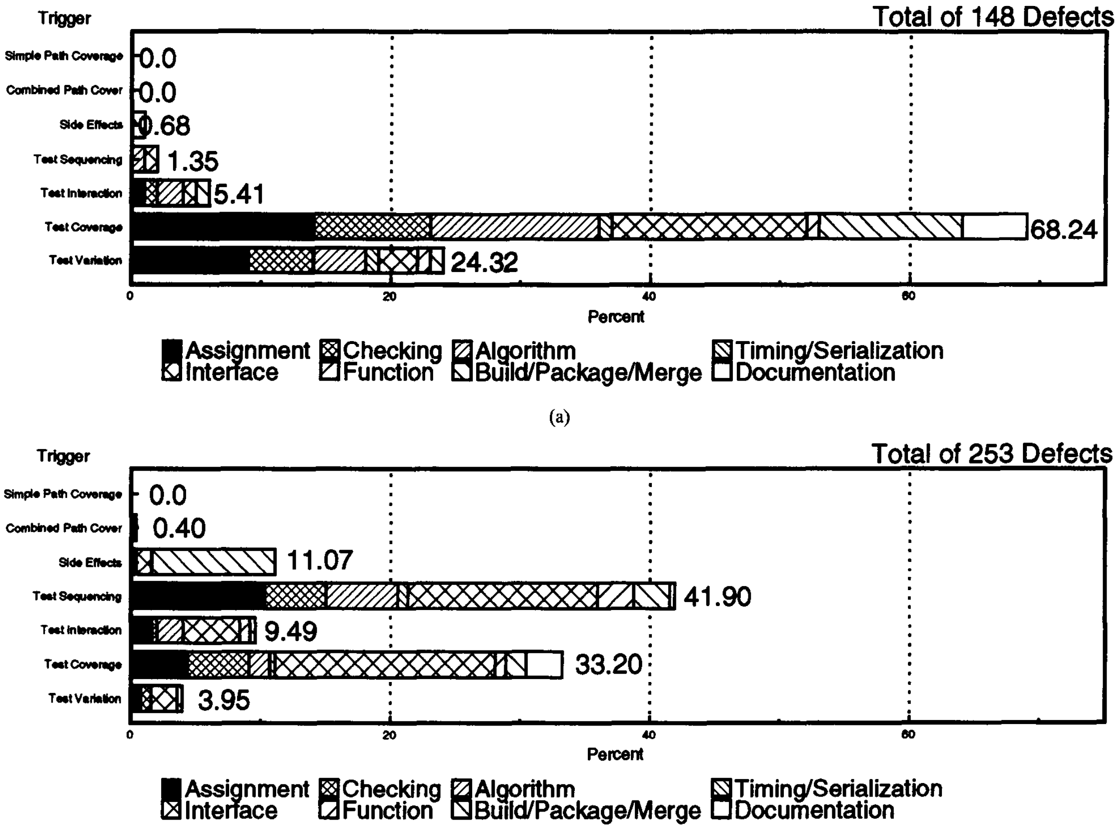
Given that defect triggers can be mapped to skills required to generate the test case that finds the defect, the defect-trigger distribution can help gain insight into the effectiveness of a test scenario. It is common to also create several test scenarios for a product, each improving the effectiveness of earlier ones. Thus, the change in the trigger distribution may be tracked to verify that it reflects anticipated trends. Examples of the evaluations of the FVT test scenario of a product and of progress between the UT and FVT development stages of this product are detailed next.

A. Test Scenario Evaluation

The usefulness of defect classification in evaluating test scenarios is illustrated by presenting the results of the functional or black box testing of a software product. The technique proposed in this paper was used to evaluate an initial test scenario of the product functions and pointed to the need for a second scenario for testing further this same functionality. A total of 401 defects were detected by these two test scenarios; 148 defects (36.9%) were detected by executing the first scenario and an additional 253 defects (63.1%) were detected by executing the second scenario. Fig. 9 shows the defect type distributions for both scenarios and Fig. 10 shows their trigger distributions. By suggesting a second test scenario that uncovered 63.1% of the FVT defects, the proposed technique helped reduce the large costs associated with fixing these defects in the field.

The first test scenario (Fig. 9(a)) resulted in relatively high percentages of assignment (25.0%), algorithm (19.59%), interface (19.59%), checking (15.54%), and build/package/merge (12.84%) defects. Most defect types were incorrect. In contrast, the percentage of function (1.35%) defects was extremely low and a cause for concern. To help evaluate the effectiveness of the test scenario, the distribution of the defect trigger (Fig. 10(a)) was analyzed. It was observed that the distribution was dominated only by test coverage (68.24%) and test variation (24.32%), while test interaction (5.41%) and test sequencing (1.35%) were very low.

Consequently, the team decided to expand the test scenarios with test interaction and test sequencing in mind with the goal of discovering more function defects. Further, testers with extensive experience with the product and its underlying hardware platforms were added to the test planning team. Fig. 9(b) shows the defect type distribution of this second test scenario and Fig. 10(b) shows its trigger distribution.



(a)

(b)

Fig. 10. Trigger distribution in two function verification test scenarios.

In the second test scenario, an additional 253 defects were found, but the distribution of defect type was not as expected. Specifically, function (4.35%) only increased slightly, while interface (39.13%) increased remarkably. However, the goal of the expanded test scenarios was achieved as the percentages of test sequencing (41.9%) and test interaction (9.49%) increased. The increase in side effects (11.07%) implied that the code under test was unstable. Consequently, the team concluded that while the test scenarios were adequate, the stability of the code precluded advancing to the SVT stage of the development process.

The percentages of interface (39.13%) and algorithm (9.09%) led the team to conclude that design structures document reinspection was necessary for some components. Likewise, the percentages of assignment (17.79%) and checking (10.67%) indicated reinspection of code for certain components.

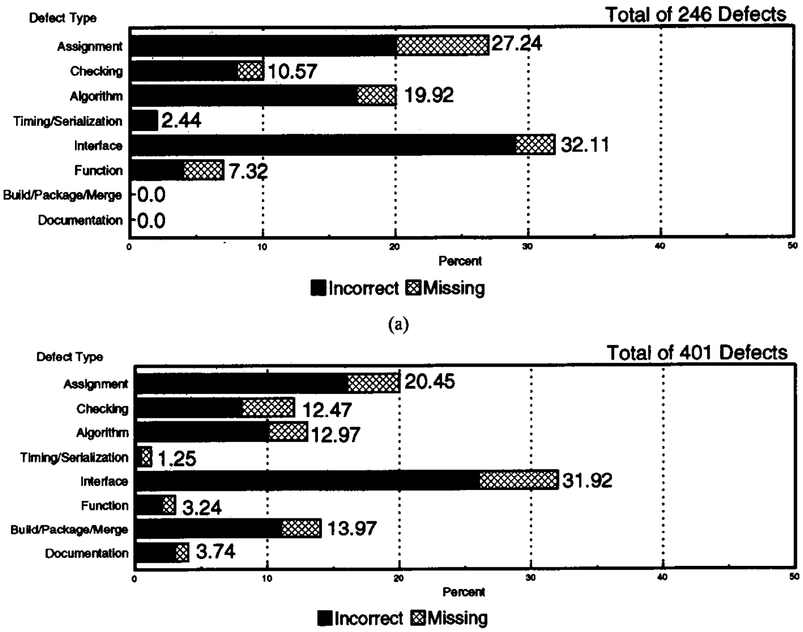
B. Progress Evaluation

The usefulness of defect classification in evaluating progress through successive stages of the development process is illustrated by presenting the results of both the white box and black box modes of testing for the previous software product. White box testing and black box testing are carried out during UT and FVT, respectively. The technique proposed in this paper has been used to evaluate progress through these two stages. A total of 246 (38.02%) defects were found via white box testing and 401 (61.98%) defects were found via black box testing. Fig. 11 shows the defect type distributions of these tests and Fig. 12 shows their trigger distributions.

White box testing (Fig. 11(a)) resulted in relatively high percentages of interface (32.11%), assignment (27.24%), and algorithm (19.92%) defects. While assignment defects are expected to be high in white box testing, interface and algorithm were cause for concern. Furthermore, while the percentages of combinational path coverage (37.5%) and simple path coverage (33.93%) in Fig. 12(a) indicate thoroughness in the test, the relatively high percentage of side effects (28.57%) needs closer examination.

When the cross-product of trigger and defect type (Fig. 12(a)) was considered, it was observed that a significant percentage of interface was being triggered by side effects. In retrospect, this discovery was a warning ignored by the team when they decided to proceed with FVT.

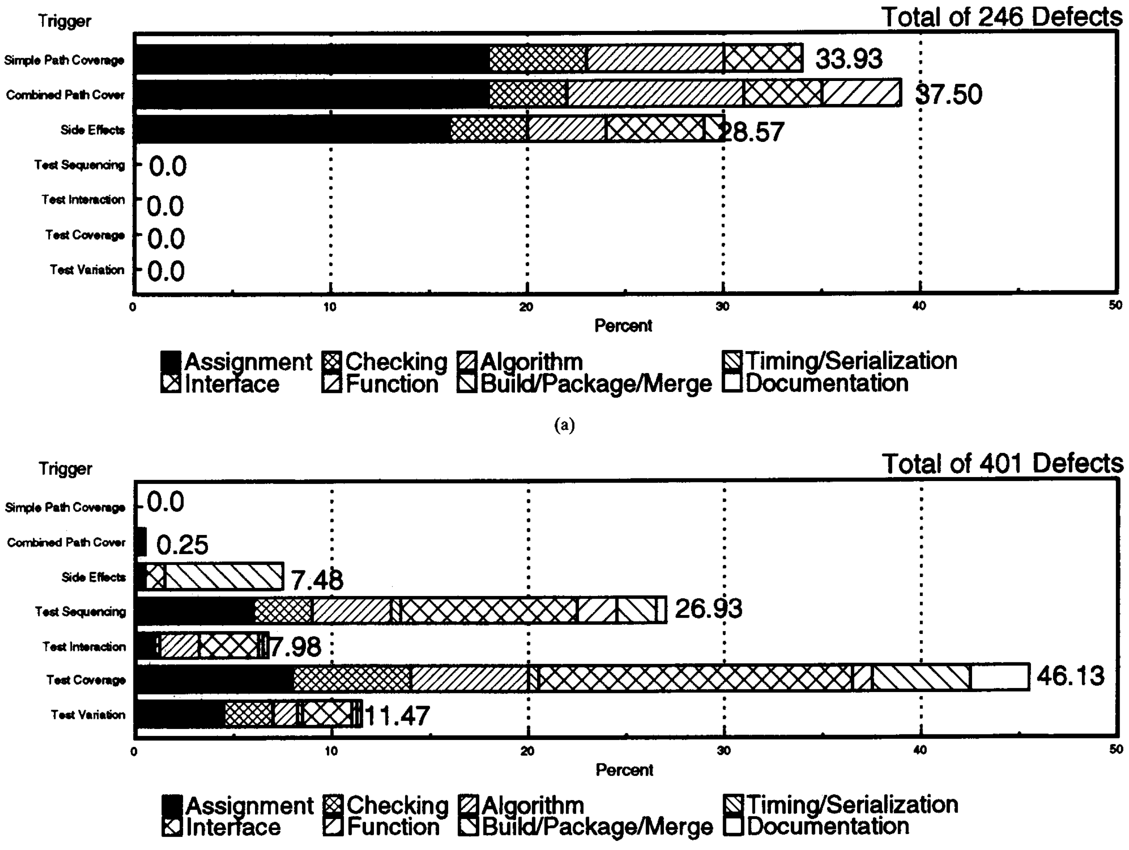
The reiterated black box testing (Fig. 12(b)) exhibited more complete test scenarios. However, the same warning signal involving a high percentage of interface (31.92%) and a significant proportion of side effects (7.48%) triggering such



(a)

(b)

Fig. 11. Defect type distribution in testing process.



(a)

(b)

Fig. 12. Trigger distribution in testing process.

interface defects was still present. Thus, the team concluded that there was insufficient progress through white box and black box testing to justify proceeding with SVT.

VI. Conclusion

In the absence of viable theoretical methods for verifying the correctness of software designs and implementations, software inspection and test play a vital role in validating both. The goal of both inspections and tests is to reduce the expected cost of software failure over the life of a product.

This paper proposes a technique that offers software designers, developers, and test planners significant guidance for rectifying, in-process, the weaknesses of their procedures, and for assessing the implications of any rectifying actions on their inspection and test processes. Such processes may involve different software technologies that can also be indirectly assessed. The technique extends the use of *defect triggers*, the events that cause defects to be discovered, to help evaluate the effectiveness of inspections and test scenarios. In software inspections, the defect trigger is defined as a set of values that associate the skills of the inspector with the discovered defect. Similarly, for test scenarios, the defect trigger values embody the deferring strategies being used in creating these scenarios.

The technique evaluates an inspection or a test activity and tracks progress between the various activities of a stage and between the various stages of the software development process. The findings of such evaluations report both the strengths and the weaknesses of an inspection or a test activity and are presented to the software development team. Reported strengths signal the start of the next activity, while reported weaknesses are followed by specific actions that aim at improving the outcome of current activity.

The usefulness of this technique in evaluating the effectiveness of software inspections and tests is demonstrated by evaluating the inspection and test activities of software products. These evaluations are used to point to both deficiencies in inspection and test strategies and progress made in improving such strategies. The trigger distribution of the entire inspection or test series may then be used to highlight areas for further investigation, with the aim of improving the design, implementation, and test processes.

Acknowledgment

The authors would like to thank G. Davidson and C. Byrne of IBM Santa Teresa Lab for numerous discussions on the use of triggers in inspection and test evaluations. Many more people across IBM development, not all of whom can be individually named, have helped run pilot studies in their development environments that proved the usefulness of the proposed technique.

References

[1] R. A. Radice and R. W. Phillips, *Software Engineering: An Industrial Approach.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

[2] W. W. Royce, "Managing the development of large software systems: concepts and techniques," in *Proc. 9th Int. Conf. Software Eng.* (a reprint of 1970 article), Mar. 1987, pp. 328–338.

[3] B. W. Boehm, "A spiral model of software development and enhancement," *ACM SIGSOFT Software Engineering Notes*, vol. 11, pp. 14–24, Aug. 1986.

[4] B. W. Boehm, "Improving software productivity," *Comput.*, pp. 43–57, Sept. 1987.

[5] S. Hekmatpour, "Experience with evolutionary prototyping in a large software system," *ACM SIGSOFT Software Engineering Notes*, vol. 12, pp. 38–41, Jan. 1987.

[6] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 182–211, 1976.

[7] M. E. Fagan, "Advances in software inspections," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 744–751, July 1986.

[8] G. J. Myers, *Software Reliability: Principles and Practices.* New York: Wiley, 1976.

[9] G. J. Myers, *The Art of Software Testing.* New York: Wiley, 1979.

[10] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. K. Ray, and M. Y. Wong, "Orthogonal defect classification for defect control," Tech. Rep. RC 17320 (Log 76564), IBM Research, 1991.

[11] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. K. Ray, and M. Y. Wong, "Orthogonal defect classification—a concept for in-process measurements," *IEEE Trans. Software Eng.*, vol. 18, Nov. 1992.

[12] W. S. Humphrey, *Managing the Software Process.* Reading, MA: Addison-Wesley, 1990.

[13] W. E. Stephenson, "An analysis of resources used on the SAFEGUARD system software development," Tech. Rep., Bell Labs, Aug. 1976.

[14] E. B. Daly, "Management of software engineering," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 229–242, May 1977.

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

[16] R. A. Radice, N. K. Roth, A. C. O'Hara, and W. A. Ciarfella, "A programming process architecture," *IBM Syst. J.*, vol. 24, no. 2, 1985.

[17] W. S. Humphrey, "Characterizing the software process: a maturity framework," *IEEE Software*, Mar. 1988.

[18] R. C. Linger, H. D. Mills, and B. J. Witt, *Structured Programming: Theory and Practice.* Reading, MA: Addison-Wesley, 1979.

[19] R. Mays, C. Jones, G. Holloway, and D. Studinski, "Experiences with defect prevention," *IBM Syst. J.*, vol. 29, no. 1, 1990.

[20] B. Littlewood and J. L. Verrall, "A Bayesian reliability growth model for computer software," *J. Royal Statistical Soc.*, vol. 22, no. 3, pp. 332–346, 1973.

[21] C. V. Ramamoorthy and F. B. Bastani, "Software reliability—status and perspectives," *IEEE Trans. Software Eng.*, vol. 8, no. 4, pp. 354–371, 1982.

[22] A. L. Goel, "Software reliability models: assumptions, limitations, and applicability," *IEEE Trans. Software Eng.*, vol. SE-11, no. 12, pp. 1411–1423, 1985.

[23] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability—Measurement, Prediction, Application.* New York: McGraw-Hill, 1987.

[24] G. Wenneson, "Quality assurance software inspections at NASA Ames: metrics for feedback and modification," in *Proc. 10th Ann. Software Eng. Workshop* (Goddard Space Flight Center), Dec. 1985.

[25] G. G. Schulmeyer and J. I. McManus, *Handbook of Software Quality Assurance.* New York: Van Nostrand Reinhold, 1987.

[26] J. C. Kelly, J. S. Sherif, and J. Hops, "An analysis of defect densities found during software inspections," *J. Syst. Software*, vol. 17, pp. 111–117, 1992.

[27] R. C. Linger, "Cleamroom software engineering for zero-deficit software," in *Proc. 15th Int. Conf. Software Eng.*, pp. 2–13, 1993.

[28] P. A. Curritt, M. Dyer, and H. D. Mills, "Certifying the reliability of software," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 3–11, Jan. 1986.

[29] B. Beizer, *Software Testing Techniques.* New York: Van Nostrand Reinhold, 1983.

[30] H. D. Mills, "On the statistical validation of computer programs," in *Software Productivity*, H. D. Mills, Ed. New York: Dorset House, 1988.

[31] J. C. Knight and P. E. Amman, "An experimental evaluation of error seeding as a program evaluation technique," in *Proc. 10th Ann. Software Eng. Workshop* (Goddard Space Flight Center), Dec. 1985.

[32] D. N. Card, T. L. Clark, and R. A. Berg, "Improving software quality and productivity," *Inform. Software Technol.*, vol. 29, June 1987.

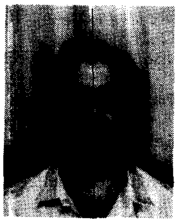
[33] I. S. Bhandari, M. J. Halliday, J. K. Chaar, and R. Chillarege *et al.*, "In-process improvement through defect data interpretation," Tech. Rep. (Log 81922), IBM Research, 1993; also submitted to *IBM Syst. J.*

[34] R. Chillarege, W.-L. Kao, and R. G. Condit, "Defect type and its impact on the growth curve," in *Proc. 13th Int. Conf. Software Eng.*,

pp. 246–255, 1991.

[35] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability—a study of field failures in operating systems," in *Dig. Papers 21st Int. Symp. Fault-Tolerant Computing*, pp. 2–9, 1991.

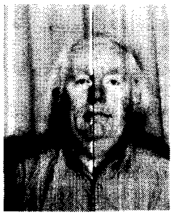
[36] I. Bhandari, "Attribute focusing: open-world data exploration applied to software production process control," Tech. Rep. RC 18320 (Log 80194), IBM Research, 1992.



Jarir K. Chaar (S'90–M'90) received the B.E. degree in electrical engineering (with Distinction) from The American University of Beirut, Lebanon, in 1981, and the M.S.E. degree in electrical and computer engineering and the Ph.D. degree in computer science and engineering from The University of Michigan, Ann Arbor, in 1982 and 1990, respectively.

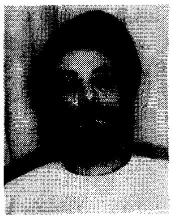
He is a Research Staff Member at the IBM T. J. Watson Research Center, Yorktown Heights, NY. Prior to joining IBM, he worked as a Senior Software Engineer at Deneb Robotics, Inc. and held the position of Research Associate at The University of Michigan. While with the University of Michigan, he researched and developed a methodology for generating the control software of manufacturing systems and coauthored many technical articles on the various aspects of this methodology. Prior to working toward the Ph.D. degree, he held the position of Instructor in the Electrical Engineering department at the American University of Beirut (1983–1985). He also was the Manager and Systems Engineer of Computeknix Microcomputers (1982–1984). His research interests include software design and verification.

Dr. Chaar is a member of Tau Beta Pi.



Michael J. Halliday graduated from the University of Nottingham, U.K.

He is a Senior Programmer at the IBM T. J. Watson Research Center. He joined IBM in 1969 and has worked on the design and development of IBM's mainframe operating systems (MVS, VM, and TPF) for much of that time.



Inderpal S. Bhandari (M'91) received the B.Engg. degree from the Birla Institute of Technology and Science, Pilani, India, the M.S. degree from the University of Massachusetts, Amherst, and the Ph.D. degree from Carnegie Mellon University, Pittsburgh, PA (1990).

He is a member of the research staff at the IBM T. J. Watson Research Center, Yorktown Heights, NY, where he is investigating system-level and process-level troubleshooting. His primary research areas are software engineering—process, metrics, and feedback—and artificial intelligence—knowledge discovery and diagnosis.

Dr. Bhandari is a member of the IEEE Computer Society.

Ram Chillarege (S'78–M'86–SM'91) received the B.Sc. degree in mathematics and physics from the University of Mysore, India, and the B.E. degree (with distinction) in electrical engineering and the M.E. degree (with distinction) in automation from the Indian Institute of Science, Bangalore. He worked as an independent hardware design consultant before returning to school. He received the Ph.D. degree in electrical and computer engineering from the University of Illinois, Urbana, in 1986.

He is currently Manager of Continuous Availability and Quality at the T. J. Watson Research Center, Yorktown Heights, NY. His work has predominantly been in the area of experimental evaluation of reliability and failure characteristics of machines. His work includes the first measurements of error latency under real workload and the concept of *failure acceleration* for fault injection based measurement of coverage. More recently his work has focused on error and failure characteristics in software.

Dr. Chillarege is an Associate Editor of the IEEE TRANSACTIONS ON RELIABILITY.