On The Evaluation of Software Inspections and Tests

Jarir K. Chaar, Senior Member, IEEE

IBM T. J. Watson Research Center
P.O.Box 704
Yorktown Heights, NY 10598

Abstract

The goal of software inspections and tests 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 which cause defects to be discovered, to help evaluate the effectiveness of inspection and test activities. In the case of inspections, the defect trigger is defined as a set of values which associate the skills of the inspector with the discovered defect. Similarly, for tests, the defect trigger values embody the various strategies being used in creating test 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 progress made in improving such strategies.

1 Introduction

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]. 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 [2], AT&T [3], GTE [4], and TRW [5].

Validation may include the reviews and walk-throughs [6] 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 [1, 7].

Unlike the informal reviews and walk-throughs held by development teams, **software inspections** are formal evaluations of the work items of a software product [2, 8]. 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.

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 [9].

Wenneson used statistics to develop some guidelines for conducting software inspections [10]. 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.

In a separate study, Schulmeyer and McManus also used statistical techniques to analyze design and code inspections data [11]. 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.

Software testing is the process of executing the code of a software product with the intention of finding defects [12, 13]. Reliability measures, such as *interfail times* [14], can be used to track progress during test.

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 [15]. 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.

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 [16]. 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.

The goal of this paper is to present a technique for assessing the effectiveness of inspection and test activities that is 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 [17]. 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 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 2 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 2.1, the trigger set used to evaluate unit and function test scenarios is defined in section 2.2, and the set of defect types of a development process is defined in section 2.3. Section 3 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 4. Section 5 presents the conclusion of the paper.

2 Evaluation Technique

This paper presents a technique for assessing the effectiveness of inspection and test activities [18, 19] that enhances Orthogonal Defect Classification (ODC) [17]. ODC classification assumes that a defect measures the semantic content of the error, without explicit direct mapping to an existing process or an implied process. A wide range of reported benefits has resulted from applying this technique to software development. Benefits as little as 5 person-days for a 5 KLOC project and as high as 848 person-days for a 100 KLOC project were documented [20].

The classification enables in-process feedback to developers when the attributes explain in their distribution the progress of the process. To do so, the attributes should be different enough that they capture the needed information and complete enough that they span the process space. These conditions are explained concisely in the necessary and sufficent conditions for ODC [17]. Once such conditions are established, the classification essentially provides the purpose of state-variable type measurement, when collected over a sample in the defect stream.

In the evaluation technique, attributes, such as *defect type* and *defect trigger*, are assigned to each defect. Defect type specifies the actual fix of a defect [21] and defect trigger points to the event that helped detect such defect [22]. 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 is 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 the current activity. Such actions may result from performing causal analysis on a small subset of high impact defects that are 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 [18, 19]. 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, the percentage expected for the values of the cross-product (defect type, defect trigger) is estimated as 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.

2.1 Trigger in Design/Code Inspection

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:

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).

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:

- **Operational Semantics:** The inspector had in mind the flow of logic required to implement the function when the defect was noticed.
- **Side Effects:** The additional effect or side effect of some documented or some implemented action was under review when the defect was discovered.
- **Concurrency:** The inspector was considering the serialization necessary for controlling a shared resource when the defect was discovered.

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.

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.

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

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

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

2.2 Trigger in Unit/Function Testing

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 Figure 1.

White Box Triggers

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.

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.

Side Effects: The defect surfaced because of some unanticipated behavior which was not specifically tested for.

Coverage	*Sequencing*	*Interaction*	*Variation*
create(file, r)	create(file_1, rw)	create(file, r)	create(file, r)
modify(file)	create(file_2, rw)	delete(file)	create(file, w)
delete(file)	modify(file_2)	modify(file)	create(file, rw)
	modify(file_1)		
	delete(file_2)		
	delete(file_1)		

Figure 1: Sample Black Box Test Cases.

Black Box Triggers

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.

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.

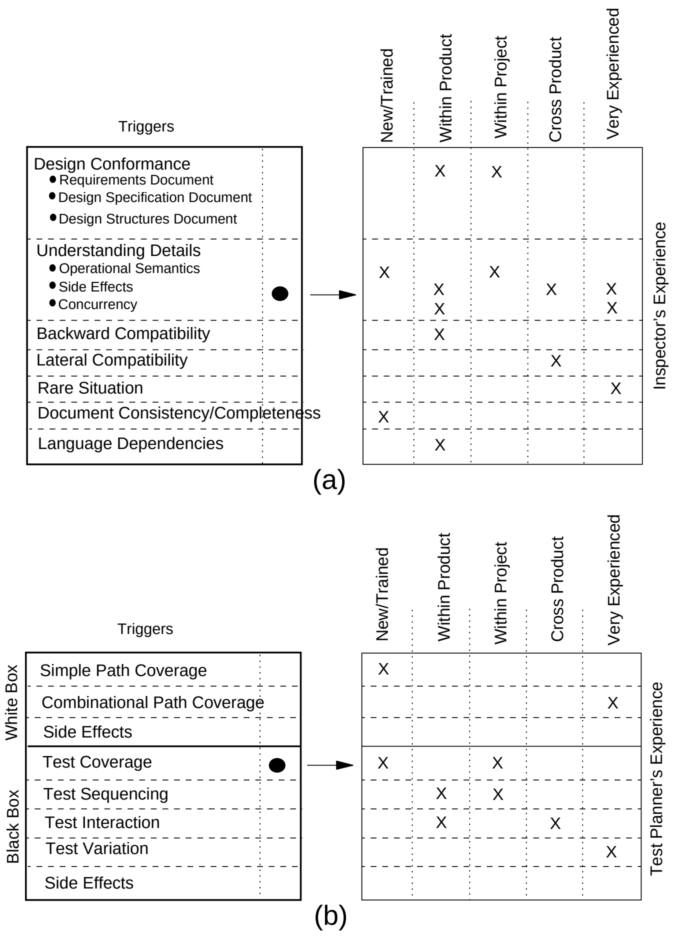
Test Interaction: The test case that found the defect initiated an interaction between two or more bodies of code each of which can be invoked independently by the tester. The interaction was more involved than a simple sequence of the executions.

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.

Side Effects: The defect surfaced because of some unanticipated behavior which was not specifically tested for.

2.3 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



Triggers	New/Trained	Within Product	Within Project	Cross Product	Very Experienced
Design Conformance • Requirements Document • Design Specification Document • Design Structures Document		X	X		
Understanding Details • Operational Semantics • Side Effects • Concurrency ●	X	X X	X	X	X X
Backward Compatibility		X			
Lateral Compatibility				X	
Rare Situation					X
Document Consistency/Completeness	X				
Language Dependencies		X			

	Triggers	New/Trained	Within Product	Within Project	Cross Product	Very Experienced
White Box	Simple Path Coverage	X				
	Combinational Path Coverage					X
	Side Effects					
Black Box	Test Coverage ●	X		X		
	Test Sequencing		X	X		
	Test Interaction		X		X	
	Test Variation					X
	Side Effects					

Figure 2: Level of Experience Associated with Triggers.

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:

Assignment: Value(s) assigned incorrectly or not assigned at all.

Checking: Errors caused by missing or incorrect validation of parameters or data in conditional statements.

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.

Timing/Serialization: Necessary serialization of a shared resource is missing, the wrong resource is serialized, or the wrong serialization technique is employed.

Interface: Communication problem between modules, components, and device drivers via macros, call statements, control blocks, or parameter lists.

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).

Build/Package/Merge: Problems encountered during the driver build process, in library systems, or with management of change or version control.

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 which might be classified as a *function* or an *interface* defect type.

3 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 which specifies the product requirements, and the design specification document which defines the functionality of the product, are reviewed by an independent team of software planners, designers and developers. Moreover, the design structures document which 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.

For each design/code inspection trigger, the skill required by the inspector can be assessed. Figure 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.

3.1 Project A

3.1.1 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 (Product A). The document describes the functionality of the product and was inspected by a team of independent software engineers. The proposed technique 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. Figure 3 shows the (defect type, defect trigger) distributions of both inspections. 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 Figure 3(a), documentation accounted for 40% of the total number of defects. Hence, the design document itself was considered inconsistent and incomplete. Further, the percentages of function (28%), interface (19%), and algorithm (7%) defects were significant but expected. 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 focussed on to yield the most defects?

To answer these questions, the distribution of defect trigger from the first inspection (Figure 3(a)) was considered. The defects discovered by backward compat-

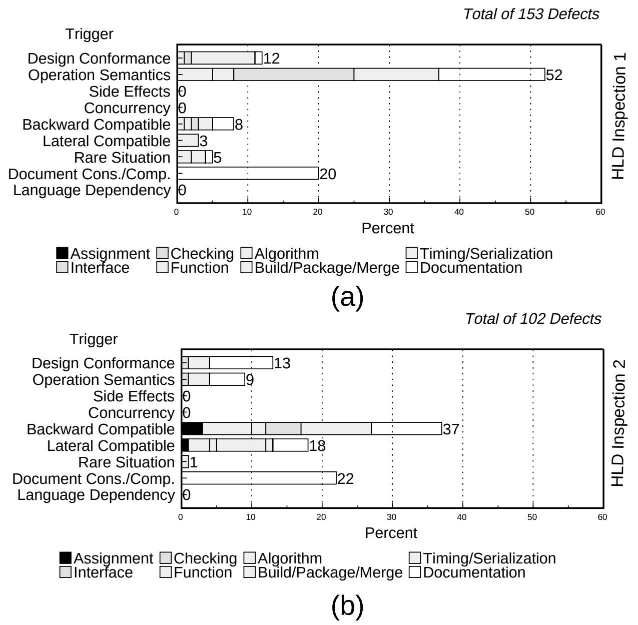


Figure 3: Trigger Distributions for Two HLD Inspections for Project A.

ibility (8%), rare situation (5%), and lateral compatibility (3%) were disproportionately low when compared to operational semantics (52%) and document consistency/completeness (20%). The team also concluded that design conformance (12%) was as expected.

While uncovering a high percentage of function, interface, and algorithm defects (54%) is explainable at this stage of the design, the small percentage of such defects triggered by backward compatibility, rare situation, and lateral compatibility was cause for concern; (backward compatibility, function) is only 2% of the distribution in Figure 3(a), (backward compatibility, interface) is 1%, (backward compatibility, algorithm) is 1%, (rare situation, function) is 2%, and (lateral compatibility, function) is 3%. 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 association with level of experience (Figure 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. Figure 3(b) shows the distribution of (defect type, defect trigger) for this second inspection.

In this second inspection, documentation accounted for 51% of the total number of defects. The percentages of function (20%), algorithm (13%), and interface (6%) defects were as expected. Most documentation, function, and interface defects were missing while most algorithm defects were incorrect. Furthermore, a drop in the overall percentage of function, interface, and algorithm defects pointed to a more stable design. However, the increase in documentation defects implied that the design document may still have been inconsistent and incomplete.

On the other hand, the distribution of triggers which uncovered the additional defects (Figure 3(b)) changed remarkably in terms of backward compatibility (37%) and lateral compatibility (18%). 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.

3.1.2 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. Figure 4 shows the (defect type, defect trigger) distributions of both inspections. 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 resulted in significantly high percentages of function (36%), interface (9%), and algorithm (6%) defects. Further, 43% of 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 (Figure 4(a)), which was dominated by operational semantics (42%), the team decided that this may have been a cursory inspection because of the relatively low percentages of concurrency (7%), backward compatibility

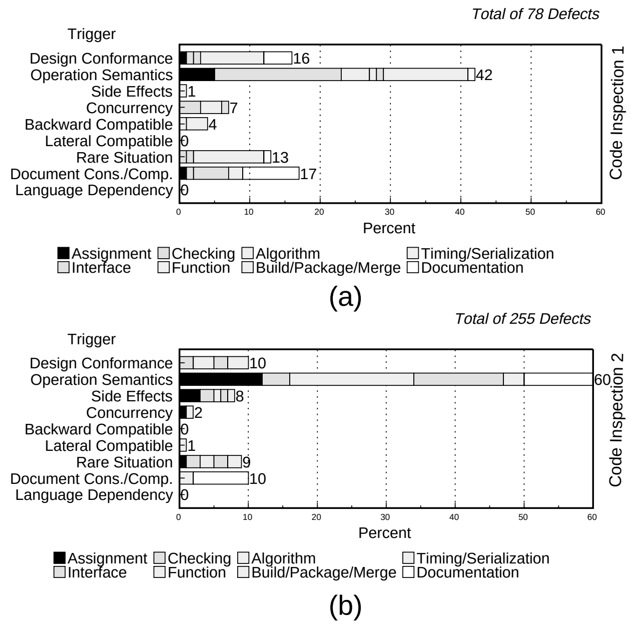


Figure 4: Trigger Distribution for Two Code Inspections for Project A.

(4%), and side effects (1%), and the lack of lateral compatibility and language dependencies.

Further, the weakness in design inspections was reflected by design conformance triggering 9% function and 1% interface defects and by rare situation triggering 10% function and 1% algorithm defects. Consequently, the team re-inspected their code with emphasis placed on functionality, and discovered an additional 255 defects. Figure 4(b) shows the (defect type, defect trigger) distribution of this second inspection.

In this second inspection, the team was surprised by the relatively low percentage of function (10%). However, algorithm (25%) and interface (18%) defects were significantly higher 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 Figure 4(b), it can be concluded that the second inspection uncovered many more interface, algorithm, assignment, and checking defects. The percentage increases in (side effects, algorithm) to 1%, (side effects, interface) to 1%, and (side effects, function) to 1% indicate that side effects may mark more design flows.

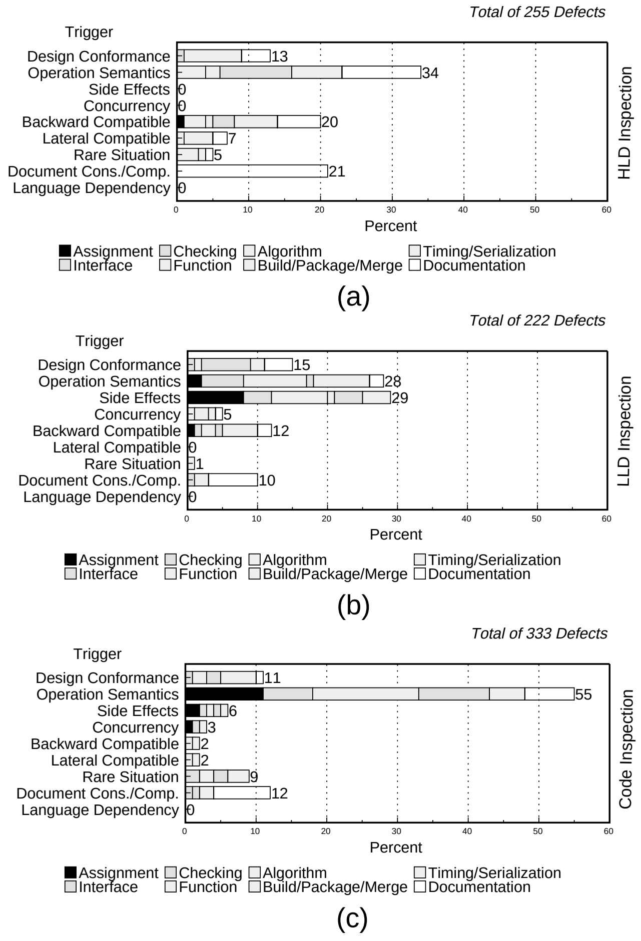


Figure 5: Trigger Distributions through Design and Code Inspections for Project A.

3.1.3 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 proposed technique has been used to evaluate progress through these stages. A total of 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. Figure 5 shows the (defect type, defect trigger) distributions of these inspections.

The diminishing trend of defect type function across all three stages was initially encouraging. 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 (Figure 5) 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 (Figure 5(b)) may have been inhibiting the discovery of more interface and algorithm defects.

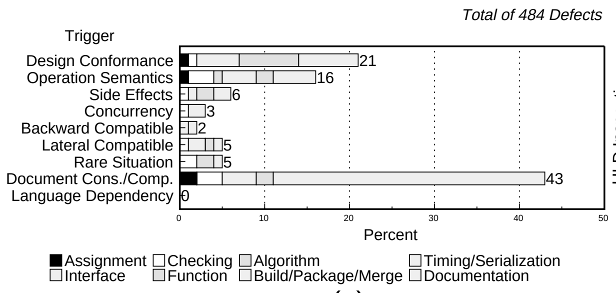
The overall decline of lateral compatibility and backward compatibility was perceived as a progressive trend (Figure 5). However, the significant percentages of side effects (29%) in design structures inspection and rare situation (9%) in code inspection were cause for concern.

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.

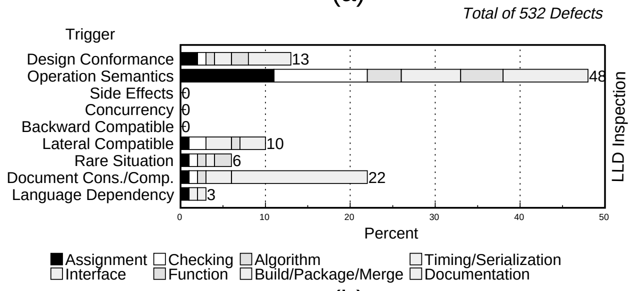
3.2 Project B

The usefulness of defect classification in evaluating progress through successive stages of the development process is also illustrated by presenting the results of inspecting the design specification document, the design structures document, and the actual code of another software product (Product B). 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 484 (32.97%) defects were found by inspecting the design specification document, 532 (36.24%) defects were found by inspecting the design structures document, and 452 (30.79%) defects were found during code inspection. Figure 6 shows the distributions of the (defect trigger, defect type) cross-product for these inspections.

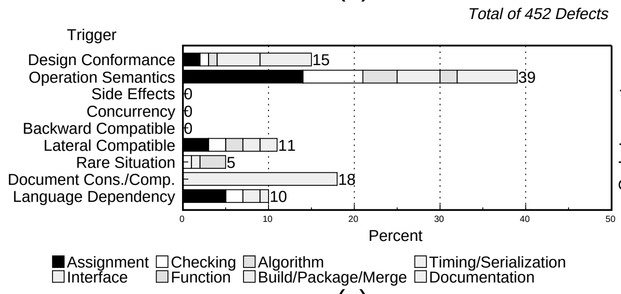
The design specification document of a software product describes the functionality of the product and was inspected by a team of independent software en-



(a)



(b)



(c)

Figure 6: Trigger Distributions in Inspection Process for Project B.

gineers. The proposed technique was used to evaluate this inspection. A total of 484 defects were detected. The distribution of their (defect trigger, defect type) cross-product is shown in Figure 6(a).

In examining the distribution of the defect types, it was noted that the percentage of documentation defects (48%) was dominant. Hence, the design document itself was considered incomplete and inconsistent. Few defects were discovered by backward compatibility (2%), lateral compatibility (5%), rare situation (5%), and side effects (6%). The team also concluded that the percentage of design conformance (21%) could be explained.

While uncovering a good percentage of function, interface, and algorithm defects (34%) is expected at this stage of the design, the small percentage of such defects triggered by lateral compatibility and backward compatibility was cause for concern. Their relatively high percentages triggered by side effects were also deemed problematic; (function, side effects) is 2% and (interface, side effects) is 1% of the distribution in Figure 6(a). 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 association with level of experience (Figure 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. Further, it was feared that the defects triggered by side effects were masking some more serious design flaws. Hence, a second inspection of the *updated* design specification document using a team of very experienced software engineers was recommended. However, the design team only acknowledged the absence of an experienced member in related hardware products and decided to proceed to the next stage of the development process. It was also decided to monitor the percentages of defects triggered by lateral compatibility, backward compatibility, rare situation, and side effects.

The design structures document of a software product details the structural and logical aspects of the product and was inspected by the development team. The technique proposed in this paper was used to evaluate this inspection. A total of 532 defects were detected. The distribution of their (defect trigger, defect type) cross-product is shown in Figure 6(b).

In examining the distribution of the defect types, it was noted that the percentage of documentation defects (33%) was still high. Hence, the design document itself was not deemed consistent or complete. The design stability was improved and no defects were discovered by backward compatibility and side effects. Few defects were discovered by lateral compatibility (10%) and rare situation (6%). The team also concluded that design conformance (13%) was improving.

Design structures document inspection uncovered a good percentage of function, interface, and algorithm defects (34%). However, the percentage of interface defects triggered by lateral compatibility increased from 1% in HLD (Figure 6(a)) to 3% in LLD (Figure 6(b)) and the percentage of function defects triggered by lateral compatibility was 1% in both HLD and LLD. These percentages were associated with a known cause, namely, the absence of an experienced reviewer in hardware related products. Hence, their presence did not affect the decision of the design team to proceed to the next stage of the development process. Again, the percentages of defects triggered by lateral compatibility and rare situation were to be monitored.

The implementation code of this same software product was inspected next by the development team. The technique proposed in this paper was used to evaluate this inspection. A total of 452 defects were detected. The distribution of their (defect trigger, defect type) cross-product is shown in Figure 6(c).

In examining the distribution of the defect types, it was noted that the percentage of documentation defects (34%) was still high. However, only 2% of the documentation defects were triggered by lateral compatibility and were deemed major defects. There was further proof of design stability; no defects were discovered by backward compatibility and side effects, and few defects were discovered by lateral compatibility (12%) and rare situation (6%). The team also concluded that the percentage of design conformance (15%) could be explained. An increase in the percentage of defects triggered by language dependencies (10%) was also as expected.

Code inspection uncovered a smaller, yet significant, percentage of function, interface, and algorithm defects (28%) and bigger percentages of assignment and checking defects (39%). However, the percentage of algorithm defects triggered by lateral compatibility increased to 2% (Figure 6(c)) for the first time and the percentage of interface defects triggered by lateral compatibility increased to 3%. Similarly, the percentage of function defects triggered by rare situation increased to 3% (Figure 6(c)) from 1% in HLD and 2% in LLD and the percentage of interface defects triggered by rare situation stayed at 2%. These percentages were again associated with the absence of very experienced reviewers of the implementation code of this product. In order to cure the above deficiencies, a very experienced hardware tester was added to the test team and was assigned the task of testing all hardware interfaces of the product. 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.

4 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. Figure 2(b) shows the skill level appropriate for 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 might need people with skill in more than just the product under test. Similarly, both test sequencing and test interaction of black box testing may 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.

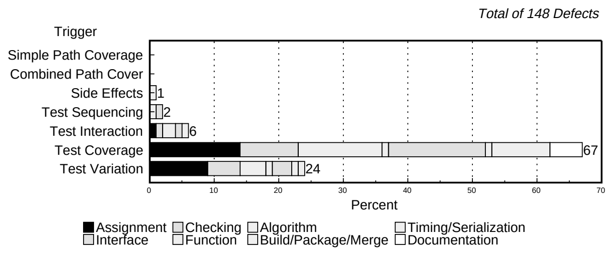
On the other hand, side effects, is an inadvertent event that can occur during both modes of testing. Hence, side effects is not associated with a particular level of experience. When side effects is the trigger for a significant number of defects in a test scenario, its presence may indicate a lack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

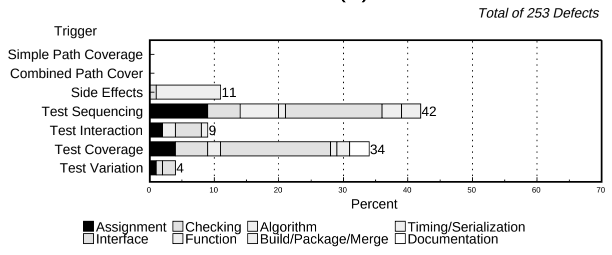
4.1 Project C

4.1.1 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 (Product C). The proposed technique 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. Figure 7 shows the (defect type, defect trigger) distributions for both scenarios. By suggesting a second test scenario that uncovered 63.1% of the Function Verification Test (FVT) defects, the proposed technique helped reduce the large costs associated with fixing these defects in the field.



(a)



(b)

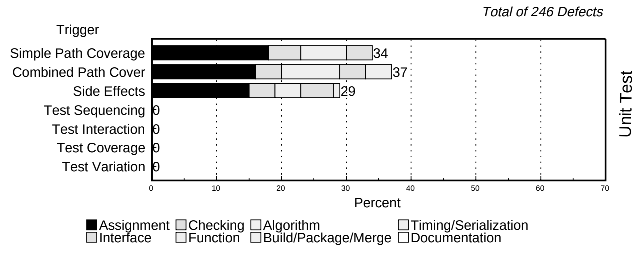
Figure 7: Trigger Distribution in Two Function Verification Test Scenarios for Project C.

The first test scenario resulted in relatively high percentages of assignment (24%) and checking (15%) defects that were attributed to the absence of any formal code inspections. Algorithm (20%) and interface (20%) defects pointed to inefficient LLD inspections. Most defect types were incorrect. In contrast, the percentage of function (2%) defects was extremely low and a cause for concern. To help evaluate the effectiveness of the test scenario, the distribution of the defect trigger (Figure 7(a)) was analyzed. It was observed that the distribution was dominated only by test coverage (67%) and test variation (24%), while test interaction (6%) and test sequencing (2%) were very low.

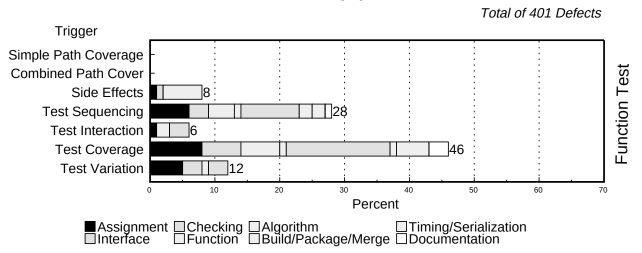
Consequently, the team decided to expand the test scenarios with test interaction and test sequencing in mind with the goal of discovering more function defects. Further, testers with extensive experience with the product and its underlying hardware platforms were added to the test planning team. Figure 7(b) shows the (defect type, defect trigger) distribution of this second test scenario.

In the second test scenario, an additional 253 defects were found, but the distribution of defect type was not as expected. Specifically, function (5%) only increased slightly, while interface (39%) increased remarkably. However, the goal of the expanded test scenarios was achieved as the percentages of test sequencing (42%) and test interaction (9%) increased. The increase in side effects (11%) 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 System Verification Test (SVT) stage of the development process.

The percentages of interface (39%) and algorithm (10%) led the team to conclude that design structures document re-inspection was necessary for some components. Likewise, the percentages of assignment (16%) and checking (11%) indicated re-inspection of code for certain components.



(a)



(b)

Figure 8: Trigger Distribution in Testing Process for Project C.

4.1.2 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. Figure 8 shows the (defect type, defect trigger) distributions of these tests.

White box testing resulted in expected percentages of assignment (49%) and checking (13%) defects. Interface (13%) and algorithm (20%) defects indicated inefficient LLD inspection. Furthermore, while the percent-

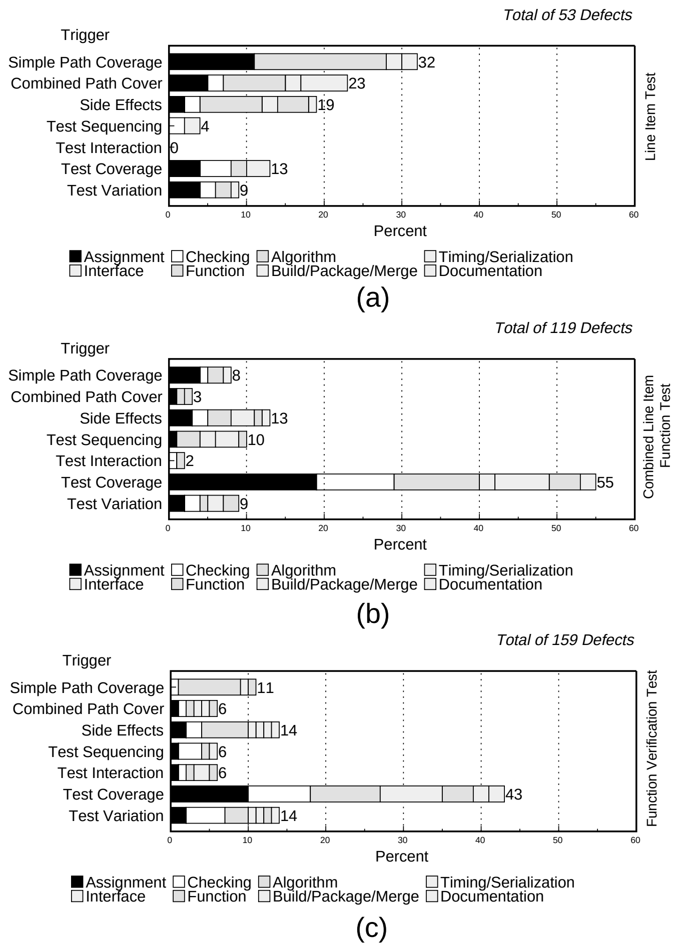


Figure 9: Trigger Distribution in Testing Process for Project D.

ages of combinational path coverage (37%) and simple path coverage (34%) in Figure 8(a) indicate thoroughness in the test, the relatively high percentage of side effects (29%) needs closer examination.

When the cross-product of trigger and defect type (Figure 8(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 that was ignored by the team when they decided to proceed with FVT.

The re-iterated black box testing (Figure 8(b)) exhibited more complete test scenarios. However, the same warning signal involving a high percentage of interface (32%) and a significant proportion of side effects (8%) triggering such 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.

4.2 Project D

A large software development project may be partitioned into more manageable subprojects, referred to as line items. Design, coding, white box testing, and black box testing are performed on each such line item. The mixed white/black box testing of a line item is labeled a line item test (LIT). Related line items are subsequently integrated and tested in black box mode via combined line item function test (CLIFT). The whole software release in then integrated and black box tested via Function Verification Test (FVT). The usefulness of defect classification in evaluating progress through successive stages of the development process is illustrated by presenting the results of the LIT, CLIFT, and FVT activities of another software product (Product D). The proposed technique has been used to evaluate progress through these stages. A total of 53 (16.01%) defects were found via LIT, 119 (35.95%) defects were found via CLIFT, and 159 (48.04%) defects were found via FVT. Figure 9 shows the distributions of the (defect trigger, defect type) cross-product for these tests.

Following the completion of LIT, it was noted that the percentage of algorithm defects (29%) was dominant. Further, function, interface, and algorithm accounted for 51% of the defects. A large percentage of assignment and checking defects (38%) was also reported. Hence, the structures document inspection and the code inspection of this product were deemed inefficient. The development team aknowledged that they did not conduct any formal code inspections and were completely relying on test scenarios to uncover the defects of their product. Consequently, evaluating the effectiveness of such tests was exceptionally crucial.

To help evaluate such effectiveness, the trigger distributions of Figure 9(a) were examined. While the percentages of combinational path coverage (23%), simple path coverage (32%), test coverage (13%), and test variation (9%) in Figure 9(a) indicated thoroughness in the test, the relatively high percentage of side effects (19%) needed closer examination. Side effects triggered relatively high percentages of function defects (4%), interface defects (8%), and timing/serialization defects (2%); these were further proof of a deficient structures document inspection because they pointed to detailed design problems.

When associated with the level of experience of the testers (Figure 2(b)), the very low percentages of test sequencing (4%) and test interaction (0%) pointed to the absence of testers with extensive experience with the product and its underlying hardware platforms. Further, it was feared that the defects triggered by side effects were masking some more serious design flaws. Hence, more elaborate testing was recommended before progressing to the next test activity. This recommendation was partially adopted by the development team in their decision to proceed to CLIFT activities.

Consequently, the team decided to expand the test scenarios of CLIFT with test interaction and test sequencing in mind with the goal of discovering more function, interface, and algorithm defects. Further, testers with extensive experience with the product and its underlying hardware platforms were added to the test planning team. It was also decided to monitor the percentage of defects triggered by side effects in subsequent test activities.

CLIFT test cases helped detect more algorithm defects (22%). Further, function, interface, and algorithm accounted for 47% of the defects. A large percentage of assignment and checking defects (46%) was also reported. Hence, further proof of deficient structures document inspection and code inspection of this product was at hand.

While the percentages of test coverage (55%), test variation (9%), and test sequencing (10%) in Figure 9(b) indicated thoroughness in the test, the relatively high percentage of side effects (13%) was again a cause for concern. The percentage of function defects triggered by side effects dropped from 4% to 1%, the percentage of interface defects triggered by side effects dropped from 8% to 3%, and the percentage of timing/serialization defects triggered by side effects dropped from 1% to 0%. However, the percentage of algorithm defects triggered by side effects increased from 0% to 3%. They pointed to design integration problems of the various line items of the release.

On the other hand, it was concluded that adding testers with extensive experience with the product and its underlying hardware platforms to the CLIFT test planning team helped increase the percentages of test sequencing and test interaction. Further, it was decided to proceed with FVT and to defer any decisions on re-inspecting the design structures documents or the implementation code of the release until FVT is complete. During FVT, the percentage of defects triggered by side effects would be monitored and would form the basis for any such decisions.

Following the completion of FVT, it was noted that the percentage of algorithm defects (29%) was dominant. Further, function, interface, and algorithm accounted for 50% of the defects. A large percentage of assignment and checking defects (38%) was also reported. Hence, further proof of deficient structures document inspection and code inspection of this product was at hand.

The percentages of test coverage (43%), test variation (14%), test sequencing (6%), and test interaction (6%) in Figure 9(c) indicated thoroughness in the test. However, the relatively high percentage of side effects (14%) was again a cause for concern. The percentage of function defects triggered by side effects dropped further from 1% to 0% and the percentage of interface defects triggered by side effects dropped further from 3% to 1%. However, the percentage of algorithm defects triggered by side effects increased further from 3% to 6% and the percentage of timing/serialization defects triggered by side effects increased from 0% to 1%.

Consequently, the team concluded that while the test scenarios were adequate, the stability of the design and/or code of certain components in the release precluded advancing to the system verification test (SVT) stage where testing is performed on the actual hardware platform of the product. The components where function, interface, algorithm or timing/serialization defects were triggered by side effects during test were targeted for design structures document re-inspection. Similarly, components where assignment and checking defects were triggered by side effects during test were targeted for code re-inspection. Following such inspection activities, the test cases created during LIT, CLIFT, and FVT will be re-executed and their detected defects will be analyzed before progressing to the SVT stage of the development process.

5 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 which 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 which associate the skills of the inspector with the discovered defect. Similarly, for test scenarios, the defect trigger values embody the various 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 the 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 some 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 used to highlight areas for further investigation, with the aim of improving the inspection and test processes.

References

[1] W. S. Humphrey, *Managing the Software Process.* Addison-Wesley Publishing Company, 1990.

[2] M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[3] W. E. Stephenson, "An Analysis of Resources Used on the SAFEGUARD System Software Development," tech. rep., Bell Labs, August 1976.

[4] E. B. Daly, "Management of Software Engineering," *IEEE Transactions on Software Engineering*, vol. 3, pp. 229–242, May 1977.

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

[6] R. A. Radice, N. K. Roth, A. C. O'Hara, and W. A. Ciarfella, "A Programming Process Architecture," *IBM Systems Journal*, vol. 24, no. 2, 1985.

[7] W. S. Humphrey, "Characterizing the Software Process: A Maturity Framework," *IEEE Software*, March 1988.

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

[9] C. V. Ramamoorthy and F. B. Bastani, "Software Reliability – Status and Perspectives," *IEEE Transactions on Software Engineering*, vol. 8, no. 4, 1982.

[10] G. Wenneson, "Quality Assurance Software Inspections at NASA Ames: Metrics for Feedback and Modification," in *Proceedings of The* 10th *Annual Software Engineering Workshop, Goddard Space Flight Center*, December 1985.

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

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

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

[14] R. C. Linger, "Cleamroom Software Engineering for Zero-Defect Software," in *Proceedings of The* 15th *International Conference on Software Engineering*, pp. 2–13, 1993.

[15] P. A. Curritt, M. Dyer, and H. D. Mills, "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 3–11, January 1986.

[16] D. N. Card, T. L. Clark, and R. A. Berg, "Improving Software Quality and Productivity," *Information and Software Technology*, vol. 29, June 1987.

[17] R. Chillarege, I. Bhandari, J. Chaar, and M. Halliday et al., "Orthogonal defect classification – a concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, November 1992.

[18] J. Chaar, M. Halliday, I. Bhandari, and R. Chillarege, "In-process evaluation for software inspection and test," *IEEE Transactions on Software Engineering*, vol. 19, November 1993.

[19] J. Chaar, M. Halliday, I. Bhandari, and R. Chillarege, "On the evaluation of software inspections and tests," in *1993 International Test Conference*, October 1993.

[20] I. S. Bhandari, M. J. Halliday, J. K. Chaar, and R. Chillarege et al., "In-Process Improvement through Defect Data Interpretation," *The IBM Systems Journal*, vol. 33, no. 1, pp. 182–214, 1994.

[21] R. Chillarege, W.-L. Kao, and R. G. Condit, "Defect Type and its Impact on the Growth Curve," in *The 13th International Conference on Software Engineering*, pp. 246–255, 1991.

[22] M. Sullivan and R. Chillarege, "Software Defects and their Impact on System Availability - a study of Field Failures in Operating Systems," in *The 21st International Symposium on Fault-Tolerant Computing*, pp. 2–9, 1991.