Fall 2006

Coverage Testing in a Production Software Development Environment

Kent Bortz
Regis University

Recommended Citation

Bortz, Kent, "Coverage Testing in a Production Software Development Environment" (2006). *Regis University Student Publications (comprehensive collection)*. 416.
https://epublications.regis.edu/theses/416

COVERAGE TESTING IN A PRODUCTION SOFTWARE DEVELOPMENT ENVIRONMENT

by

Kent Bortz

A thesis submitted in partial
fulfillment of the requirements
for the degree of

MASTER OF SCIENCE
IN
COMPUTER INFORMATION
TECHNOLOGY

REGIS UNIVERSITY

SCHOOL FOR PROFESSIONAL STUDIES
2006

ABSTRACT

By

Kent Bortz

This project proposes that current testing methodologies used by standard testing tools are not sufficient to ensure sufficient test coverage. Test tools provide important and irreplaceable test data but are not capable of guaranteeing high percentage of path exposure (coverage). If the code path includes loop statements like, “if” or “when” then the number of paths to test grows exponentially. The growth of the code path becomes exponential when nested decision statements are considered. The most common methodology used in today’s testing environment verifies each line of code but does not verify all path combinations. Testing per line of code can not guarantee complete test coverage when considering the variations of nested code paths. The result of lower coverage is a higher field defect rate that increases the overall product support costs.

TABLE OF CONTENTS

LIST OF FIGURES

Number **Page**

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my wife, Kimberly who without her support and belief in my abilities I would have failed long ago.

GLOSSARY

Functional Verification Test (FVT). Traditionally a test preformed in a development organization before releasing code/hardware to Formal Test.

System Level Test (SLT). Longest duration test cycle. Often, this test suite is considered formal test and is preformed by test groups external to the development team.

Manufacturing Verification Test (MVT). A short duration test preformed to verify hardware software before shipment to the customer.

Microcode. A computer program that resides on hardware and the end-user does not directly interact with

Unit Test (UT). A simple test preformed by the development teams.

Code. The set of instructions that are written by a software developer that are deployed on the given hardware platform.

Development. The team and/or effort to produce a solution that satisfies customer requirements while operating in the development organizations frame work.

KLOC. Thousand Lines Of Code. Metric used to define code size and gauge test effectiveness

General Availability (GA). The final milestone where a product becomes available to the customer.

FORWARD

This project is based on an actual hardware/software development company. In order to mitigate revealing any intellectual property the company will be referred to as Software Development Company or SDC. The data that is presented is only representative of actual data used. The representative data is accurate within the confines of this project and can be used for comparative calculations.

Chapter 1: Testing Introduction

INTODUCTION

Software testing is necessary part of any development approach. Traditional as well as object-oriented software development approaches both require software testing. What is the purpose of testing? The simplest answer is; “to execute code in order to find program errors (Wittaker, 2000).” As anyone who has written even a simple piece of code can attest too, errors are inevitable. No mater how experienced the developer or simple the code assignment, errors will exists (Wittaker, 2000).

The simple objective of executing code to find program errors is achieved by different methods depending on the desired outcome. Early software testing was narrow in focus and simple in it methods (Horgan, 1994). These early testing methods were sufficient because software was not complex and the environments that the software was deployed in were highly controlled (Musa, 1975). As computing started to become more common in business, the tasks required of software became more complex and varied. The requirement for software to become more robust also required software testing suites and testers to become more sophisticated.

The first digital computers were used almost exclusively for basic numerical problems. The primary concern of the programmers was representing the necessary algorithm, which commonly involved breaking the problem down into sequential steps (Miller, 1992). Early super computers were limited by their capacity and programmers had to work around daunting capacity and performance issues. Accordingly compact, fast-running code was necessary, even if testing became more difficult. With early code development readability and portability were hardly considerations at all. Early coding was almost exclusively “bit-level”; data was directly controlled at the hardware level by the coder (Musa, 1975). The typical result was spaghetti style coding and was very difficult to trace and debug.

Over time machines grew faster and more powerful. With the addition of capacity and speed programs were used for a wider range of tasks. Computer programs could now include compilers and operating systems. These new applications did little real world number crunching but did create a number of nested decision points (Wittaker, 2000). This led to the next stage of software design, top-down procedural design input (Boris, 1990). Pascal is an example of a language that uses the top-down approach. The spaghetti style code of the first generation computers was now replaced with the top-down approach. This means that the "goto" structure of the first generation machines was replaced by the structured flow of top-down (Boris, 1990). This made testing simpler because errors could be traced much more quickly and simply. By the 1980s, languages like C++ had been developed to allow the implementation of object-oriented design in a wide variety of situations (Horgan, 1994).

The real danger of a code error, also called a defect, is not the glaring problems that crash a system or prevent compiling. The real danger comes from defects that are not catastrophic and only happen under very specific conditions resulting in slightly skewed results. This type of error results in output that looks correct but is flawed. Error-path defects are the most difficult class of defects for a coder/tester to find. These defects result from incorrect inputs being applied to the code. It is impossible for a coder to be able to anticipate how his/her code will react to all possible input (Boris, 1990).

Testing should be considered as a part of the development process. Often coders feel that testing is separate from code development and are hesitant to include test early, when the most good can be done. To understand the testing requirements for both object-oriented software development and traditional software development the history of the issues needs to be understood.

The development of more complex software testing was helped along by American quality initiatives of the 1980’s. Six-Sigma, Order-I, and STEP all reaffirmed the need to verify code before delivery to the customer (Cheung, 1980). No matter, the quality method chosen the goal of software testing is to verify function and content. Although the primary metric used to gauge testing effectiveness is, “defects captured per KLOC”, the primary goal of testing is not to create quality. It is impossible to predict were every defect exists in a body of code and therefore it is impossible to find every defect input (Boris, 1990). A common mistake made by many it is to assume that if something is tested there will not be any defects in the code. This of course is contrary to the primary goal of testing, verifying function and content.

The desire to find all defects is irresistible and considerable resources have been spent in the software testing domain to achieve 100% defect exposure (Miller, 1992). The most widely used test suite that attempts to expose all defects is coverage testing. This testing method falls short of its goal for fundamental reasons that will be explained later. When used in conjunction with other test suites across various test phases coverage testing may uncover defects invisible to the other suites.

TEST PHASES

Software testing does not only occur after all software development has been completed. Software Development Company's testing is broken into different phases and code enters into each phase based on the progress of the development cycle (see figure 1). The earliest testing phase is Unit Testing (Miller, 1992). This phase is preformed by the software engineer who wrote the code. Unit testing is preformed to verify a discreet segment of code that has not been integrated with other code segments (Hong, 2002). Unit testing is only intended to verify a single function in a code segment and the input and output values are often limited to true or false condition statements (Hong, 2002).

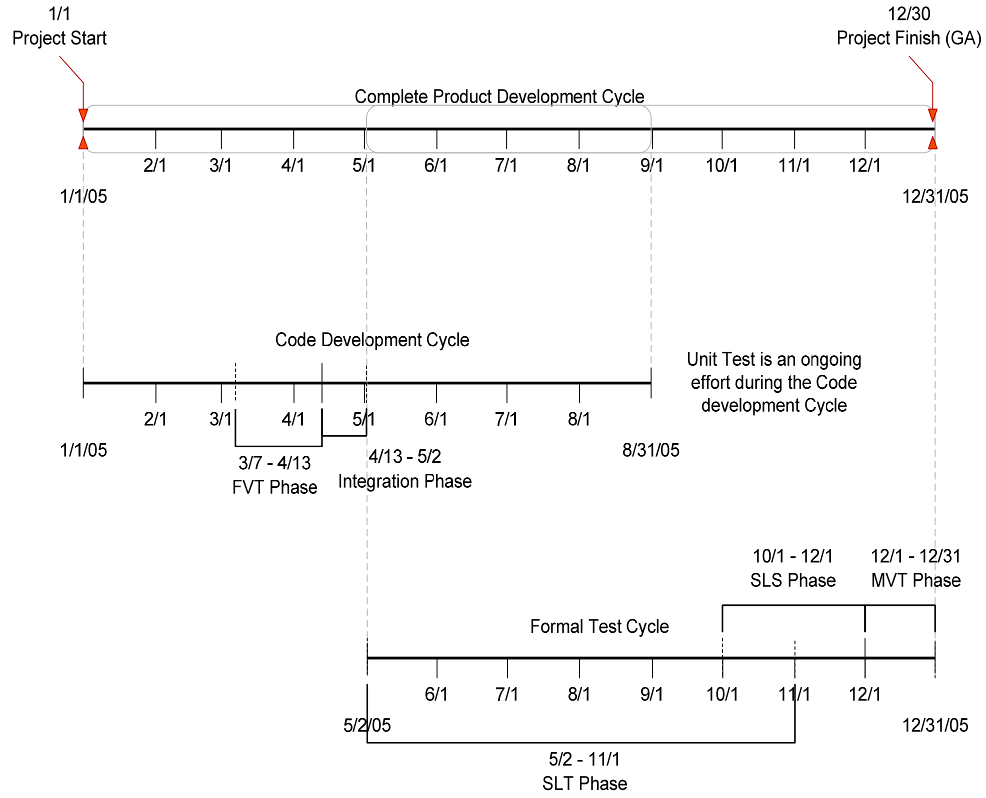


Figure 1 - SDC Development cycle

Integration test

After the code mass has reached a point where independent code segments can be married together to form meaningful function groups integration test is required. Depending on the development/test group structure integration test may be considered a part of the development organization or the test organization. The intent of integration testing is to verify the integration of finite functions into a macro-function (Hong, 2002). This verification is a logical follow on to unit testing but tests multiple functions and how the functions interact as a whole. Integration test will generate defects but they should not be considered as indicators of product quality as the code being tested is a collection a partial functions that are being tested together (Cheung, 1980). Not until formal test is entered can defect data be used to calculate quality numbers or coverage percentages.

Function Verification Testing (FVT)

Once enough functions have been developed Function Verification Testing (FVT) can be started (Piwowerski, 1993). FVT is intended to verify logical function groups. A common analogy used with software development and test is that of automotive manufacture. This analogy fits well to explain FVT testing. Unit test is similar to making sure a bolt will fit into a required hole. This test is simple and very limited. Integration testing is analogous to verifying that a fender will fit onto the car. FVT takes what was accomplished in Unit Testing and Integration Testing to a higher level. In the automotive analogy FVT would group functions logically and test them together. An automotive FVT test would be to verify that the engine starts or the head lights turn on. Function testing is meant to verify code function groups but not the entire solution (Duran, 1980).

System Level Test (SLT)

System Level Test (SLT) is intended to verify the entire code package from the perspective of the user (James, 1980). The SLT cycle is the longest and most involved test cycle (Elaine, 1990). In the automotive analogy SLT would be the road test. This is the test cycle where all the various code functions come together and are verified in an environment that simulates real world use. The SLT cycle is composed of various test suites (Elaine, 1990). These test suites are intended to verify as many code paths as possible.

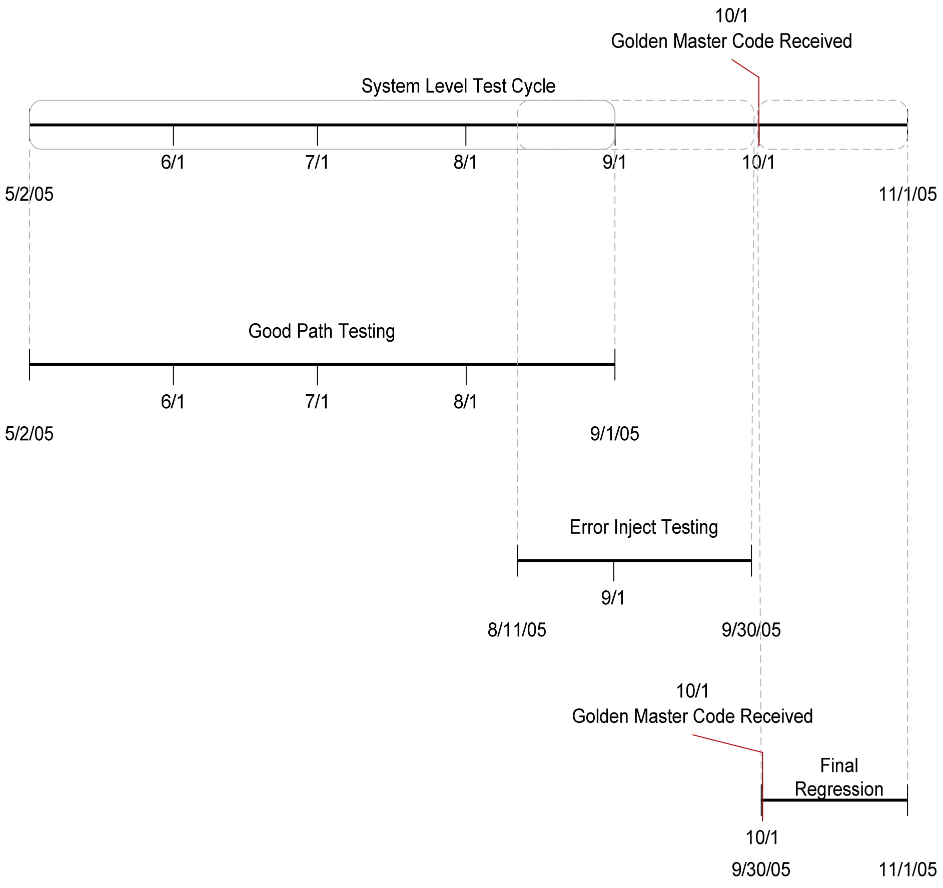


Figure 2 – SDC Test cycle

Traditionally, this is when coverage testing also takes place. Figure 2, shows what a typical SLT test is composed of. The durations of each test suite are relative and vary drastically from one test cycle to another. The primary test suite that takes place during SLT is good path testing (Duran, 1980). The focus of good path testing is to verify how the code will react in a customer environment during normal use (Elaine, 1990). Good path requires the code to be tested in an environment that simulates the customer environment as closely as possible and used data pushing tools as its primary source of input to exercise the code. The input is intended to all be “good” and errors conditions not are expected (Duran, 1980). During good path testing if an error condition is achieved then defect reports are generally created to log the event. The defect rate generated during good path is the primary source of product quality numbers and reliability calculations (Musa, 1975).

ERROR PATH TESTING

Error path testing is used to verify that the code can detect bad input or output data and error conditions are appropriate (Wittaker, 2000). This testing uses bugging devices and data pushers intended to create errors. With the automotive analogy this testing is when you see vehicles driving on wet courses or swerving at high speed to avoid a traffic cone. Error path testing is intended to verify that the code can perform in the worst conditions and is able to detect a data error (Wittaker, 2000). When a data error is detected the code should take the correct action and log the problem. The defects that are generated in this phase of testing are a challenge to debug as the conditions that were used to enter into the error condition must be fully understood.

FINAL REGRESSION

Once Good path and Error path testing have been completed a final version of code is created. This version called, the Golden Master, contains fixed to the defects found during the previous SLT phases of testing. The Golden Master is the code development teams' best effort as a final, production ready code drop. The Golden Master is subjected to a custom build SLT test suite that is based on the failures seen during the SLT prime testing. Once the final regression testing is complete then SLT is complete.

System Level Serviceability (SLS) Testing

System Level Serviceability (SLS) Testing is used to verify documentation that will be used to service the software. The primary mechanism for software support is the service point of entry. The service point of entry is where the code recognized a problem and alerts the user or service agent. Once a service point has been created the error should be logged and as much data captured as possible. Thorough SLS testing will verify that all problems are:

1. Logged – A meaningful entry is made into the error report

2. Notification is sent – Depending on the error and customer service contract the user of a support center may be contacted when an error occurs.

3. Data logging takes place – Error data must be collected at the time of an error.

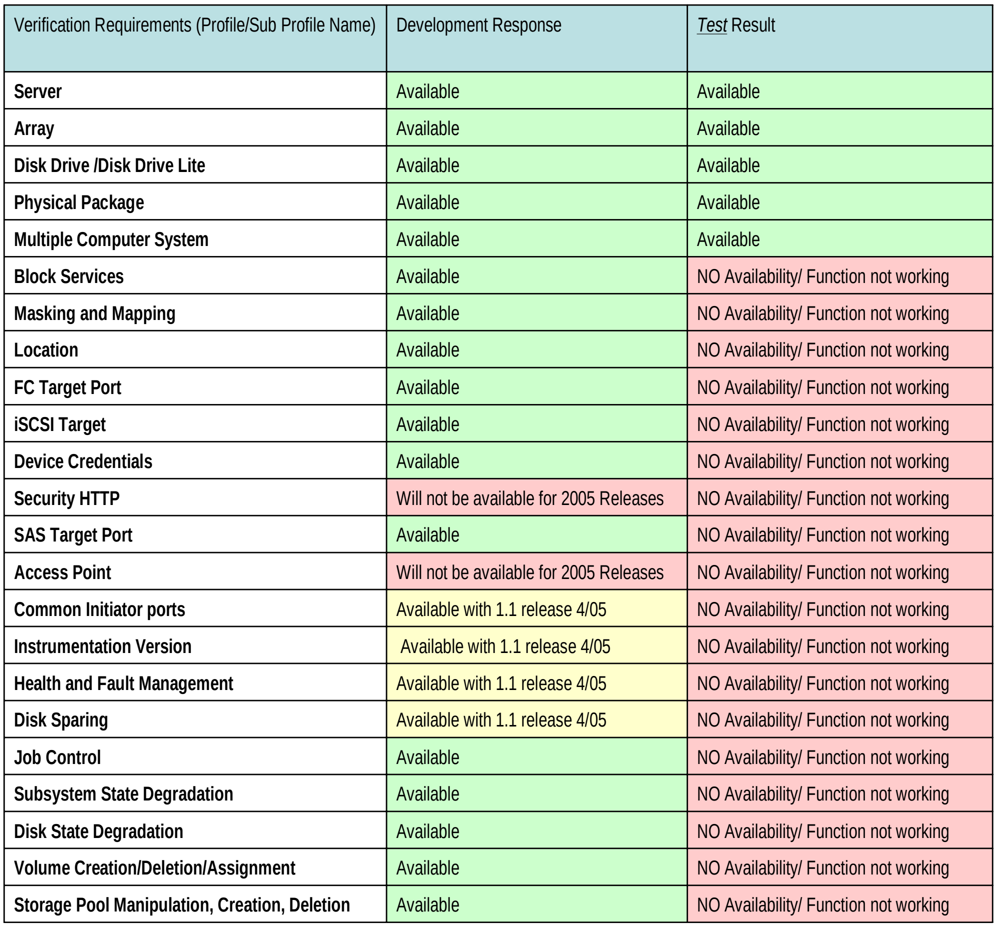
4. External documentation verification – The problem determination guides must be verified and shown that they help resolve any problems.

Manufacturing Verification Test (MVT)

Manufacturing verification testing is the only testing that takes place outside the Development/Test environment (Piwowerski, 1993). MVT traditionally takes place at the manufacturing facility and is used to verify Hardware and software for manufacturability. MVT verifies that the software can be loaded on the hardware and a very basic bring up test suite is preformed. MVT is a short duration and simple test that generally does not generate a significant number of defects.

TEST CRITERIA

Regardless of the testing phase criteria needs to be established (Miller, 1992)(Horgan, 1994). Testing criteria is generally broken into entry/exit and pass/fail requirements. As an example is a Functional Verification Test is to be preformed the entry criteria must first be met. Entry criteria are defined by the respective test group and agreed to by the appropriate development group. The entry verification of all test phases is run as a T0 regression test. Figure 3 shows a typical entry criteria matrix. A typical FVT test entry verification test would be limited to verifying that a subset of function is available and working in the code (Wittaker, 2000). Not until the formal pass/fail portion of testing does the full expected code function get tested.



Verification Requirements (Profile/Sub Profile Name)	Development Response	*Test* Result
Server	Available	Available
Array	Available	Available
Disk Drive /Disk Drive Lite	Available	Available
Physical Package	Available	Available
Multiple Computer System	Available	Available
Block Services	Available	NO Availability/ Function not working
Masking and Mapping	Available	NO Availability/ Function not working
Location	Available	NO Availability/ Function not working
FC Target Port	Available	NO Availability/ Function not working
iSCSI Target	Available	NO Availability/ Function not working
Device Credentials	Available	NO Availability/ Function not working
Security HTTP	Will not be available for 2005 Releases	NO Availability/ Function not working
SAS Target Port	Available	NO Availability/ Function not working
Access Point	Will not be available for 2005 Releases	NO Availability/ Function not working
Common Initiator ports	Available with 1.1 release 4/05	NO Availability/ Function not working
Instrumentation Version	Available with 1.1 release 4/05	NO Availability/ Function not working
Health and Fault Management	Available with 1.1 release 4/05	NO Availability/ Function not working
Disk Sparing	Available with 1.1 release 4/05	NO Availability/ Function not working
Job Control	Available	NO Availability/ Function not working
Subsystem State Degradation	Available	NO Availability/ Function not working
Disk State Degradation	Available	NO Availability/ Function not working
Volume Creation/Deletion/Assignment	Available	NO Availability/ Function not working
Storage Pool Manipulation, Creation, Deletion	Available	NO Availability/ Function not working

Figure 3 – Entry criteria matrix

Pass/fail criteria are also determined by the test group and are generated by documents such as the functional specification, marketing requirements and development design documentation. Unlike the entry verification portion of a test phase the pass/fail portion is unique to each test phase.

For example, SLS pass/fail requirements are far different from those of SLT. The pass/fail criteria are used by the test groups to define the test exit criteria.

Chapter 2: Evolution of Testing

Computer technology has become woven into every aspect of human society (Horgan, 1994). The reliance on computer technology has placed ever higher demands on hardware and software testing (Wittaker, 2000). The requirement for low defect incident rates in released products has forced testing to evolve (Miller, 1992). Testing techniques used in the past have been outmoded by more modern and effective methods. Cost, consumer requirements, and rapid technical change are the driving forces behind the evolution of testing.

BUSINESS MODEL OF TESTING

At both the consumer and the business level, the cost of computing has dropped drastically dropped and the reliability and performance has increased. Over the past 30 years the average cost of computing has exponentially decreased (New Economy, 2006). Figure 4, shows the exponential dive of computing costs (New Economy, 2006). The push to continually reduce price and improve in all other measurable aspects has forced development teams to look for efficiency improvements within their processes. Test has not been excluded from the market driven pressures to shorten test schedules and cut costs while decreasing field defect rates (Wittaker, 2000). To meet market demands genuine solutions must be implemented to be successful. Reducing cost by simply cutting headcount or improving time to market by reducing testing schedules are examples of short sighted business based solutions that are destine to fail.

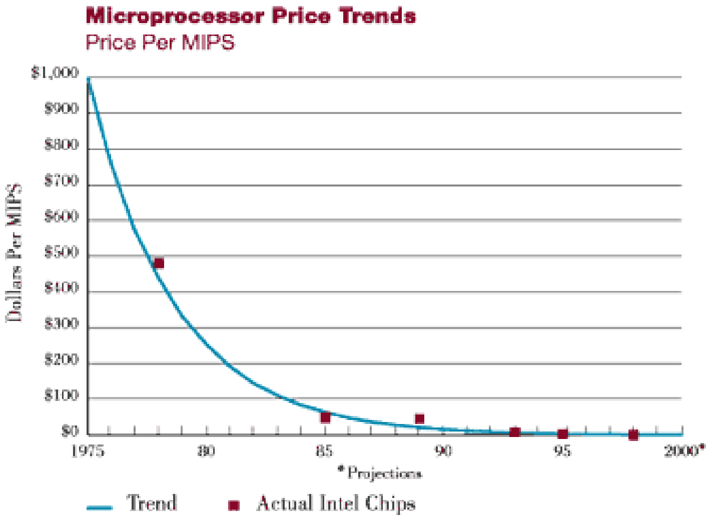


Figure 4- Price trend

Personnel Reduction

Personal reduction can only be successful if the person hours spent on the product are more efficient and effective (Piwowerski, 1993), (Wittaker, 2000). Automation is one of the leading solutions being adopted by industry to effectively reduce test headcount. The use of automation does not remove all human elements from data analysis but relives personnel from mundane and repetitive tasks. One example of automation being used in a test environment is defect detection. Historically, a technician would sit in front of a consol and monitor a test waiting for an error. Automation removes the technician and replaces him/her with an automated support system to monitor multiple tests at the same time. When an error occurs the data logs are collected and the test engineering team can perform failure analysis. Automation has a higher rate of first time failure detection because the human characteristics of fatigue and boredom are no longer an issue. This translates to a lower defect incident rate for released products.

Shortened Test Cycle

Market demands often require a products development/test schedule to be compressed. The driving issues behind schedule compression include beating a competitor to market, meeting specific revenue targets or remedy known field issues with pervious releases. Regardless of the root of the requirement to compress the schedule burden placed on the development and test teams are the same. The function that is expected to be delivered does not change but the amount of time the development and test teams have to work with is shortened. Solutions like automation can help but are may not be enough to keep a shortened schedule (Wittaker, 2000). The most effective solution would be similar to the manufacturing process of JIT (Just In Time). This manufacturing model increases manufacturing efficiency by having good delivered to each manufacturing process only when needed. The JIT model directly translates to software testing. Instead of waiting for large and complex code segments to be delivered for test; smaller less complex segments can be delivered more often (Boris, 1990). This will allow test to start earlier in the development process and detect problems sooner. All the traditional test stations like FVT and SLT are present but the amount of time allotted to each will be proportionally shortened. The combined effect is an overall shorter test and development cycle. The danger of this approach is the integration of discreet functions into larger more complex function happens later in the test cycle. Pushing function integration out in the schedule caries the risk finding a catastrophic integration defect so late in the development cycle that GA will have to be delayed.

Statistical Test Model

Automation and function delivery management are only a part of the evolution of testing. Statistical testing is becoming a standard in most major test labs because the use of normalized data allows trends that would be invisible with traditional methods to become apparent (Miller, 1992). By utilizing usage and performance data, statistics can be applied directly the testing function, resulting in a reduction of redundant testing and allowing test to focus on the portions of the software with the biggest impact on the system, and reducing the overall test schedule. These improvements can significantly decrease the amount of resources required for software testing (Hong, 2002). Statistical testing can also be used to determine when it is time to stop testing a software product, through reliability and entropy metrics. Strategically designed application of statistical testing can improve reliability measures and reduce the levels of uncertainty present in the testing.

Formal Statistical Verification

A statistical model test is composed of both white-box and black-box testing used to establish if code or a code segment conforms to the established functional specifications. The goal of white and black box statistical testing is to use statistical techniques to ensure software quality and to provide quantitative measures of stability, reliability, and conformance to specifications. White-box testing assumes that the code is complete enough for examination and conformity measurements (Boris, 1990). Black-box testing is intended to only test code from the user's point of view through the defined interface (Boris, 1990). Black-box testing is inherently a superficial test and makes deriving quantifiable data difficult.

By implementing the collection of test generated data into operational profiles, developers can utilize statistics to direct how the testing resources are applied, thereby reducing redundant testing, focusing testing on portions of the software with the biggest impact on the system, and reducing the amount of testing required overall. These improvements can significantly decrease the amount of resources required for software testing. Statistical testing can also be used to determine when it is time to stop testing a software product, through reliability and entropy metrics. Strategically designed application of statistical testing can improve reliability measures and reduce the levels of uncertainty present in the testing.

Statistical Testing by Test Phase

Each testing phase has a specific goal and the use and type of statistical model differs between each. For a statistical test to be successful the function specifications need to be defined. It is the requirement for clearly defined specifications that excludes the use of statistical testing from some early testing. This early testing is usually considered a pre-formal test and includes developer based testing. Pre-formal test should be limited to go/no go testing due to the diminutive range of function returns and lack of defined specifications.

Functional Verification Test is the earliest phase of formal testing where a statistical test can be successfully introduced. The function delivered to FVT is grouped and complex operations can be preformed. In the case of maintenance releases the entire code function of plan will be available for testing. A function verification test by nature is a white-box testing environment (Boris, 1990). The code front-end would generally not be available during an FVT test. Appendix A shows what the statistical data would look like for a device/microcode function verification test. A statistical test would more accurately describe the codes performance during FVT. The decision to release the code to other test functions would be based on quantitative data and not simple the test schedule.

System Level, Service Level and Manufacturing Verification testing operate in a black-box testing mode. These tests do not measure the performance within a function or device but as a system as a whole. Black-box testing uses a statistical model like white-box testing but at a higher level. The specifications that are tested with black-box are more based on user experience (Piwowerski, 1993). This limits the use of statistical testing to only the quantitative portions of each test.

Chapter 3: SDC Coverage Testing

Code coverage evaluation involves identifying the segments of code that are not executed with multiple runs of a program. Coverage testing is a measure of the proportion of a program exercised by a test suite, usually expressed as a percentage. Theoretically 100% coverage can be achieved but is not practical in real testing. Testers use the coverage test percent to help ensure that a substantial portion has been executed. Coverage measurement is critical to evaluate the effectiveness of the test. The most basic level of test exposure is code coverage testing and path coverage is the most methodical form of coverage testing. Some intermediate levels of test coverage exist, but are rarely used. The coverage model used by SDC is traditional code coverage. Traditional code coverage tools are integrated as the code is being developed. Each code segment or code path will have a hook added that the coverage tool will monitor for. The added code hook provides and index counter to record which statements are executed. The inserted code hook remains in the executable throughout the testing process. The inserted coverage test code is only used during the execution of the of each code path.

LIMITATIONS OF THE SBC COVERAGE TESTING MODEL

Coverage testing at Software Development Company is fraught with same problems seen industry wide. When a path is not being executed the code coverage hooks are not used to generate any test data. The coverage hooks are static and are present throughout the code. The addition of the code hooks can affect execution time and code behavior. Altering execution time changes error timing windows. Because released code will not include any coverage hooks the test level code does not accurately represent the release code. SDC has had a problem when calculating the number of hooks expected by the test group. Because the coverage hooks are added either by the development team or when the code is compiled the total number of hooks is highly dependant on developer buy in to the coverage test process. Each code segment owner is responsible for adding coverage hooks the compiler recognizes for each command in the code segment. There is no accurate method to ensure that each hook is accurately incorporated. In the early 1990's SDC developed a coverage testing system named Execution Time Mapping Tool (EXMAP). EXMAP was an attempt to apply coverage theory into a more usable system that could be deployed company wide (Piwowerski, 1993). At the time of the first implementation of EXMAP the SDC code portfolio was considerably smaller and narrow in function. The SDC software offering was limited primarily to device driver support software and some vendor applications when EXMAP was first implemented. Over time SDC turned its corporate focus from hardware development to offering a full support solution. By 2000, SDC offered a full solution package for mid to enterprise class customers. This refocusing required the SDC development team to develop a broader and more complex function set for all its products. With the increase in function, EXMAP no longer could be used as a coverage tool. Code size moved from an average of 20-30 KLOC to 500+ KLOCK. The human and machine overhead required to run EXMAP had become too high and it was abandoned.

With the removal of EXMAP SDC code coverage testing had been limited to long duration user experience test runs. Long duration user experience test runs were intended to flush out code defects by running the code in a black box manner long enough that each function/path had been executed. The use of long duration user experience testing has caused a steady increase in defects per KLOC year to year.

The Failure of SDC EXMAP

The current average SDC new function release is ~500 KLOC for enterprise class products. Each KLOC is comprised of hundreds of simple code functions that pass values to other functions. Each code segment can contain multiple code paths/hooks. Figure 5, represents the simple code segment:

If P then F1 else F2.

This function states that if the value assigned to *P* is equal to the value entered then the value for *F1* is returned and if the value entered is not *P* then the value for *F2* is returned.

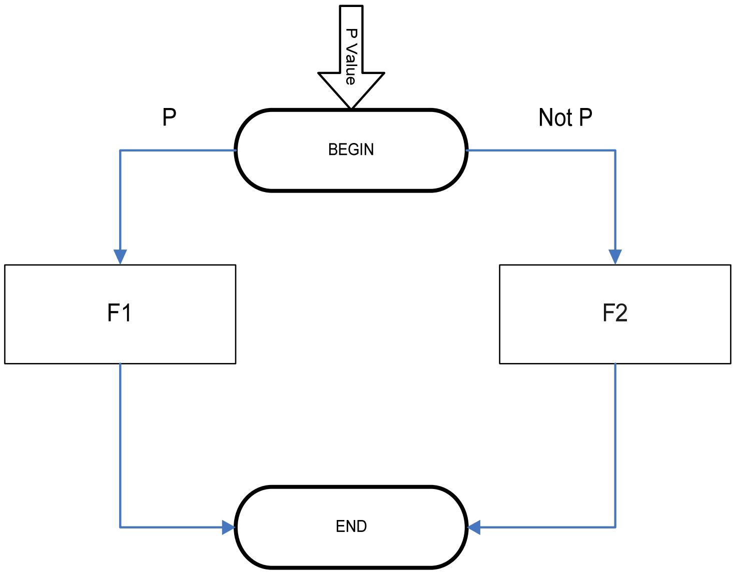


Figure 5 - Simple Code Segment

The Figure 5 code segment is only one line but is spawns two separate functions. With the addition of more condition statements a logically simple code segment can become much more complex to test with the SDC MAPEX coverage model. Figure 6 shows the code segment;

```
Begin
   input (x, y);
       while (x > 0 and y>. 0) do
         if (x> y)
             then x: = x - y
             else y: = y - x
         endif
       endwhile;
  output (x +  y);
end
```

This code segment uses two inputs *X, Y*. If X and Y are greater than 0 and X is greater than Y then X is equal to X-Y. If Y is greater than X then Y is equal to Y-X. The value returned is X + Y post the above calculation.

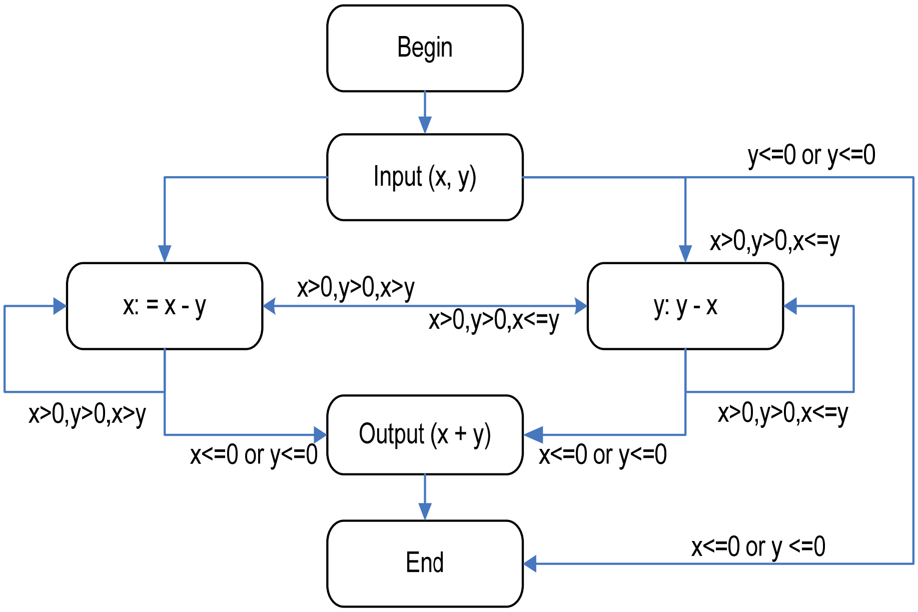


Figure 6 - Comparative code segment

Even though the function shown in Figure 6 is only 9 lines it branches 3 calculations and 9 comparison functions. A single KLOC comprised of functions similar to Figure 6 would generate 1000 comparison functions and 300 calculations. The SDC EXMAP coverage model required code developers to place a hook at each function. If coders are 99% accurate when placing code hooks in a 500 KLOC release approximately 5000 functions would be missed. No matter how long EXMAP was run the 5000 missing hooks would not be executed and because EXMAP required testers to use hook data supplied by developers, the test team would never be aware of the missing hooks in the code.

The Cost of EXMAP

The cost of EXMAP to SBC incorporates more than the daily burden rate of machine time and person hours to support it. The cost of any failed testing model is the cost to fix/repair/replace defects released to customers. Figure 7 shows the cost of EXMAP as SBC's code releases became larger and more complex.

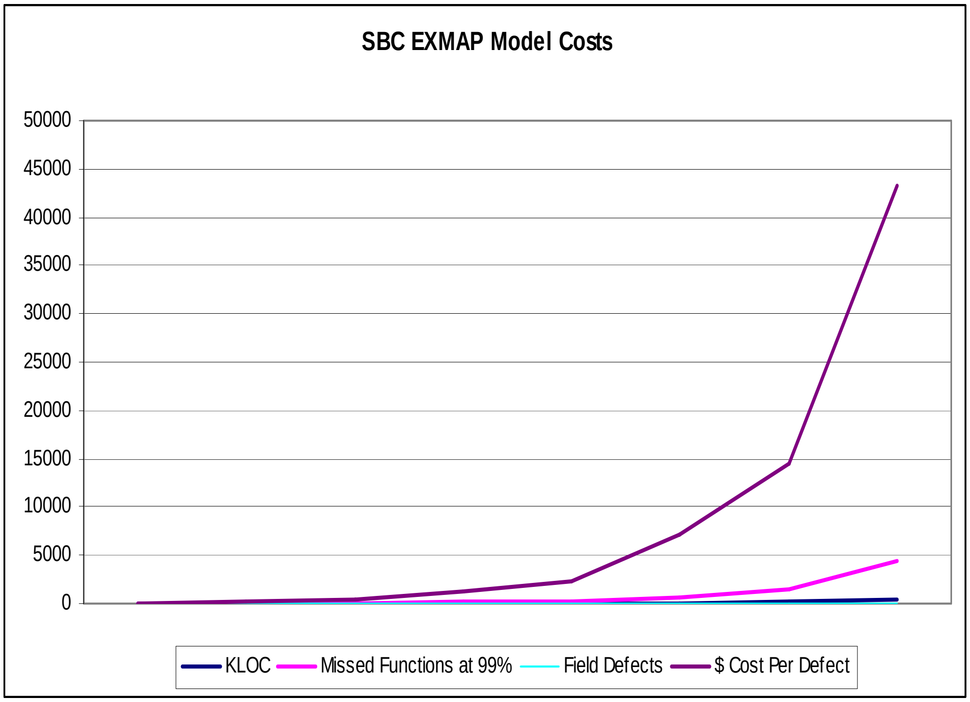


Figure 7 - EXMAP Cost per KLOC

Figure 7, assumes that each developer will be 99% accurate when placing hooks into the code and a single field defect will occur with every 100 missed function hooks. Each defect is estimated to cost $1000 to fix/repair/replace. Actual field defect cost rates are closely held financial information. The estimated $1000 per defect cost is extremely conservative. When a field defect is found, SBC will involve entire code and test team to create and validate the fix and the fix will be bundled and released as a new code level. A defect will consume resources from development, test, manufacturing, customer support, and management before being released to the field. Table 1, shows a conservative $1000 per defect and a theoretical 99% accuracy for placing coverage hooks. The cost of EXMAP becomes exponentially more expensive with each KLOC added to each release.

KLOC	Missed Functions at 99%	Field Defects	$ Cost Per Defect
1	10	0.1	$ 100.00
2	20	0.2	$ 200.00
4	40	0.4	$ 400.00
12	120	1.2	$ 1,200.00
24	240	2.4	$ 2,400.00
72	720	7.2	$ 7,200.00
144	1440	14.4	$ 14,400.00
432	4320	43.2	$ 43,200.00

Table 1 - EXMAP Actual

EXMAP alternative – User Experience Testing

By the late 1990's the EXMAP deficiencies forced SBC to abandon it and find an alternative. The growth and fragmentation of the different business organizations within the SBC hierarchy did not lend its self to adopting a universal testing model. Each development and test team adopted a coverage model that fit best. The primary driver behind all computing growth since the mid 1990's has been better performance at a lower cost. Market factors forced development and test teams to choose a coverage model that allowed them to churn code as fast as possible with the lowest burn rate. A majority of the SDC test and development teams adopted a long run user experience test coverage model. This model removes most performance and coverage metrics from the test environment. Code is tested in a black box fashion for the duration of the test. The concept driving user experience testing is creating a testing model that mimics customer use of the code. Because customer behavior is assumed by the testing model, any defects a customer would encounter in the field should be found during testing.

The philosophy of long term user experience testing as SBC applies it is fundamentally flawed. Long run user experience testing does have a shorter duration and because no specialized skills are required to design the test, the personnel burn rate is lower. This does not consider the cost of a defect when found in the field. By 2002, the computing boom had slowed and mid to enterprise class customers were no longer willing to contend with defects. Computing at the enterprise level and below, had become commoditized and deep rooted corporate alliances where now being questioned over code quality. SBC struggled to maintain the customer base and grow market share while industry analysts dogged each wave of product releases. One of the primary contributors to struggling SBC code quality is over investment in user experience testing. The user experience testing model assumes that the customer behavior is predictable and can be contained in the model. Assuming the user experience model covers 95% of all the customers 50 customers in 1000 will encounter a defect. Based on the calculations form above this conservative estimate would cost $50,000. The dollar cost per defect for the user experience model does not include the cost of lost market share. A customer that encounters a single defect will likely run into more than one defect because the user experience model attempts to predict the user behavior. If a user encounters a defect they are likely operating outside the model boundaries and will encounter multiple defects. Multiple defects drive the mid to enterprise level customer to a competitor solution and shrink SBC's market share.

Chapter 4- Filling the EXMAP Void

The removal of support for EXMAP marked the end of a homogenous company wide coverage testing method for SDC. As each division within SDC devised its own coverage testing model and market demand for price competitive code releases resulted in customer experience testing. The failure of customer experience testing contributed to the shrinking market share and rising field defect rates and associated costs. As SDC continued to move forward with larger and more complex releases the need for coverage testing methods that do not require the overhead of EXMAP to maintain and is more thorough that user experience testing.

Solution Scope

SDC is a multinational development company supplying a full range of technology to all segments of the industry. The coverage replacement for EXMAP and the current customer user experience model being proposing is limited to midrange data storage device development and testing. The solution that is posed would apply across the SDC development environment.

Solution Requirements

EXMAP failed because is relied heavily on developers to place hooks correctly in the code. As code releases became larger the inaccuracy of hook placement was magnified until EXMAP became unreliable and costly to maintain. The user experience testing model relied too heavily on the test team replicating customer behavior to be practical. As the market changes the demand for high quality code releases will continue to increase. The coverage test tools used by SDC are not capable of providing customers with the low defect rates they demand. For a solution to be effective, defect rates are not the only issue that needs to be considered. The next code coverage solution must also take into account business aspects such as cost to develop/maintain and operation overhead. In a large company like SDC with a broad portfolio the transportability of the coverage tool across divisions must also be considered. For a solution to meet the current and foreseeable requirements of a coverage testing tool for SDC it must meet the following:

- No coverage hooks - Future SDC test coverage solutions can not rely on coverage hooks to be placed in the code. EXMAP failed because it relied on developers to place coverage hooks in the code as the developed it. Even if the SDC coders are extremely accurate placing hooks a 1% error rate translated into hundreds of possible defects reaching the field.

- Robust – SDC currently has multiple coverage tools in use across the company. SDC needs a tool that is robust enough to be deployed across the organization. The tool will need to be generic enough to be used on all products can provide specific coverage testing for the products it is used to test.

- No assumption based model – The user based experience model currently used by SDC attempts to predict the behavior of a customer. As the customer base grows the number of customers that fall outside the models coverage also grows. SBC must have a coverage model that does attempt to quantify customer behavior.

Statistical Specification Performance Testing

The requirements for a coverage testing tool apply to all software/hardware companies that are attempting to gain market share in the commoditized computing market. The market move from speed and capacity at the cost of quality to a market that demands higher quality has changed the demands placed on test groups. The demand for shorter testing cycles and higher quality has pushed the SDC testing organizations to the brink of failure. The reorganization of test cycle components and utilization of faulty test tools has resulted in lower quality products across SDC.

One available tool that meets the SDC requirements of not requiring code hooks, being robust and not predicting customer behavior is Statistical Specification Performance Testing. This type of testing relies of the specifications for a product to be well defined and available to the test team before test start. Statistical specification performance testing uses product specifications to define the boundaries of the coverage testing. The specifications used to define each boundary must be quantitative and be limited to defining a single aspect of the product performance. The primary assumption is that is all the boundaries are defined as specifications and the code contained within each specification control boundary will perform as expected in the field. This type of testing does not replace classical error path testing but can be used to augment it as the error response can be quantified and each boundary tested. Statistical specification performance testing requires the data sample size to be large enough to show the performance of the code and how closely it meets the specifications.

Design of Specifications

A specifications needs to provide reasonable feedback on aspects of a product that result in better performance or reliability. Using a quantifiable level of detail is critical to defining each specification and control. For example, a diagram of instruction timings for a CPU is not an adequate specification, although it is extremely detailed. CPU clock speed also is not an adequate point of measure measurement, although it is quantifiable and simple to summarize.

The primary element of a specification is that it should have built in control. After each specification that is going to be tested is chosen, other variables should be eliminated. If a variable can not be eliminated then more analysis of the specification control needs to be preformed. An example of eliminating variables outside the specification control is if comparing storage device speed, all tests need to be preformed on the same data files and same host machine. Comparing the read/write performance of Storage Device A that is connected up to a slow host machine, and Storage Device B that is attached to a faster host machine will not result in usable data about the storage devices. Conversely, performing testing on two different types of host systems can generate data used to characterize the storage devices performance if the specification controls are adequate. For example, Storage Device A has inefficient AIX attachment drivers, but Storage Device B has horrible LINUX attachment drivers. Isolating the storage device from the driver performance data is impractical, and since the device attachment drivers are possibly proprietary, it might also be impossible. If specification controls are not sufficient the generated test data is meaningless. Without defining the specification controls and eliminating the test variables adequately the storage device statistical performance test generated useless data when testing across two host platforms.

Data Analysis

Because test control specifications are a major influence in the GA of a product test designers can not be swayed by pressure to pass a product or alter data controls and variables. Table 2, shows the results to a statistical specification performance test for a storage device. The data is broken down between device and % performance to each control. Each device can perform to 100% of the specification of each control. The data in Table 2, assumes that all variables have been bounded by the control data and that enough data has been collected for each device to be statically meaningful. The results for each device are calculated from the entire data set for each control.

Example Control Data Combined Results

	%	%	%	%	%
I/O Device	Control 1	Control 2	Control 3	Control 4	Control 5
381	0.0	2.4	6.4	34.7	56.5
716	0.0	0.0	0.3	13.3	86.4
8810	0.0	0.6	8.9	27.2	63.4
8880	0.1	3.3	14.9	27.9	53.9
9038	0.2	1.1	6.4	42.1	50.3
9104	0.0	0.7	5.2	22.3	71.8
1931	0.1	2.2	3.5	2.3	92.0
2548	1.4	3.3	8.3	13.5	73.6
9078	0.0	0.8	7.2	26.3	65.7
9605	47.8	0.0	0.0	0.0	0.0
9032	1.3	1.5	3.6	7.1	86.5
9028	0.4	3.1	2.4	1.0	93.1
9029	0.0	0.0	0.0	0.0	100.0
1930	0.0	0.0	0.0	0.0	100.0
8813	0.0	0.0	0.0	0.0	100.0
9066	0.0	0.0	0.0	0.0	100.0
9080	0.0	0.0	0.0	0.0	100.0
9015	0.0	0.0	0.0	0.0	100.0

Table 2 - Generic Control Data Results

The control percent is calculated by how closely the device meets the specified control value. The example data shows that no single device achieved meeting each control specification. Some devices did achieve 100% satisfaction of the specifications but preformed poorly in all other control specifications. This indicates that the devices are not meeting the specified control values and an underlying defect is causing specification performance issues.

Chapter 5 – Statistical Performance Testing at SDC

The application of Statistical Specification Performance testing as an alternative to the current coverage testing model, user experience testing, was limited to a small <50 sample of SDC data storage devices. The data storage devices are established and previously released devices at the time of testing. No hardware changes were made for the duration of the Statistical Specification Performance testing. During testing the firmware code base on each device was a previously released level that had been evaluated and had been running in customer accounts for approximately 8 months prior to the start of the Statistical Specification Performance Testing.

Identification of test controls and variables

The identification of control specifications for the SDC data storage devices was straight forward. The physical storage devices had been in the field for over 3 years and represented the 2nd generation of the specific physical form factor used for the device type. A 3rd generation device had been release approximately 18 month prior to test start. Because the vintage of the physical devices the testing did not include physical testing. The Statistical Specification Performance Testing was limited to evaluating the firmware that resided on the storage devices. Any mechanical issues that did arise were accounted for in the data and assumed to be due to drive age. There was no possibility to benchmark the mechanical aspects of the drive due to vintage and all mechanical failures were scrubbed from the data. The intent of the Statistical Specification Performance Test was to evaluate the firmware and not to debug hardware issues of the data storage device.

The storage devices expected performance was well documented in both external and internal publication. The external publications specified performance data like capacity, speed and reliability. Internally published specifications detailed performance data that included error path information and degraded performance specifications. The specifications given in this document are only representative and are not the actual performance data for any SDC device.

The test control performance data was all related to physical device performance. The test controls can be classified into two categories; internally and externally observable. The externally observable test controls was performance data that could be observed from outside the data storage device. Bytes written, bytes read and capacity are representative of externally observable performance data. Internal test control performance data included error rates and incorrectly written data blocks. All internal performance data was collected in a device log page that could be parsed and the data read. The external data was collected via host data driver applications.

Application of Statistical Specification Performance Testing

Statistical Specification Performance Testing first required an application to be written that would collect the performance data for all the devices. This applications operated by using File Transfer Protocol (FTP) to capture performance logs from the host running the data drivers for each device and the device logs from each data storage device. Once the performance logs were collected from the host and the devices; the performance data was parsed out and uploaded into a DB2 database. This process could be automated, but for this first run test it was left as a manual process.

Once that data was uploaded into the database the data could be accessed by a standard database query. Each database entry was for a complete data run of a device. Table 3; shows a truncated database entry for a singe device.

Device	Run	Wrt GB	RD Gb	Error rate 1	Error Rate 2	Stops	Data skip	Error Wrt	Error Rd	Prem wt	Perm rd	Temp wt	Temp rd	Device error
659	DataRun1_ASME___I1_970D	33.8	498.3	24,925,533	40,597,758	0	4	0	0	0	0	0	0	0

Table 3 - Parsed Data Table

The header information was added to each data query and is not contained in the actual database. The parsed data was used to quantify performance for each device.

Performance Data Analysis

After the data was collected, parsed and extracted from the database it would be ready for analysis. Once the data is parsed for each device; the data is placed in a spreadsheet for analysis. A spreadsheet is the best option for SDC because it is a tool the test team was familiar with and it allowed calculations to be preformed on rapidly. The output from the spreadsheet was broken into two sections. The first section was an overall summery of device data and the second was a breakdown by device.

SDC Summery Data

The data summery chart shows how much was read and written, and the calculated error rates. Table 4, shows the overall summery for the group of SDC data storage devices used in testing. The data summery chart shows data broken two sections. The bytes section shows how much data was processed collectively for all the devices in the test group. This section is exclusively externally observable and does not require calculations beyond totaling that bytes processed for each device. The rates section uses data parsed from the data device log pages and is calculated. This section calculated each rate based on the number of occurrences for each event divided by the amount of data processed. The column labeled “SPEC” defines what the control specification value is.

Test:	**SDC Generic Summary**		
CODE Level:	**R123456**		
Date:	**2006-01-01**		
(bytes)	SDC Data Device		
Mb WT	60957.6		
Mb RD	207849.3		
		Cycles	1280
Total Mb processed	268806.9		
(rates)		SPEC	
Skip Data	195.38	< 0.8	0.00
Data Write Stop	131.0	< 197	0.97
Error 1	2.2E+005	5E+006	22.48
Error 2	1.9E+004	1E+005	5.14
Permanent Errors			
Data In	30478.8	1E+005	0.33
Data Out	51962.3	1E+006	1.92
Temporary Errors			
TEMP_WT	224.1	100	0.45
TEMP_RD	831.4	250	0.30
Total Perms	6		

Table 4 - SDC Specification Summery Report

If the calculated rate falls outside the specified rate limit it is highlighted in red. Any red highlighted data shows that a control specification is out of spec and a possible defect exists. Data that is not in specification would tell the test engineer that more investigation is needed. The test engineer would then use the device breakdown section of the spread sheet to see more detail than what is available from the data summary chart.

SDC Device Breakdown Data

Table 5, shows the device breakdown data that corresponds to the summery data presented in Table 4. The performance for each device is shown for each control specifications. Like the Summary Data, any values are out of specification. The specification values are shown above the actual device performance values. The data shows the specific performance for each device. Using this data it is possible to for the SDC test engineer to pinpoint what control specification is out of design and on which device and how far it is out of specification. This will allow the test and development team to focus on the specific code segment that the control data corresponds to.

SDC Generic Device Breakdown

Code Level R123456

Date: 2006-01-01

Device	RUN	Mb WT	Mb RD	Total Mb	Radar Rates							
					> 3E2	> 3E2	> .5	< 166	> 10	> 10	> 3E4	> 1E5
					Type1_ERRS	Type2_ERRS	Skip Data	WRT_SKP	Temp Wrt	Temp OUT	Perm IN	Perm HDW
381	ABABRB00I	2130.4	4687.5	6817.9	8.7E+005	1.5E+006	532.6	---	426.1	4687.5	---	---
716	ABABRB00I	1706.8	5118.5	6825.3	9.0E+005	1.9E+006	---	128.7	1706.8	1023.7	---	---
8810	ABABRB00I	2990.4	5984.0	8974.4	5.2E+005	1.1E+006	996.8	128.4	66.5	352.0	2990.4	---
9030	ABABRB00I	3845.8	8119.5	11965.3	1.7E+006	9.7E+005	1281.9	128.4	274.7	8119.5	---	---
8880	ABABRB46_I	3848.1	6839.9	10688.0	1.1E+006	3.4E+005	296.0	128.4	1924.1	977.1	---	---
9038	ABABRB46_I	2560.5	5551.9	8112.4	9.3E+005	1.8E+006	2560.5	128.4	426.8	5551.9	---	---
9104	ABABRB46_I	3851.1	6842.6	10693.7	9.6E+005	1.5E+005	550.2	128.4	770.2	3421.3	1925.6	---
1931	ABCWRB00	778.6	10084.2	10862.8	4.0E+004	8.5E+003	389.3	155.1	29.9	916.7	---	10084.2
2548	ABCWRB00	615.5	10287.8	10903.3	7.2E+005	1.8E+006	615.5	128.4	---	1469.7	---	---
9078	ABCWRB49I	2521.5	4586.9	7108.4	2.3E+005	1.9E+005	10.3	129.5	504.3	509.7	2521.5	4586.9
9032	ABCWRBRH	765.4	9969.5	10734.9	5.8E+005	1.1E+006	382.7	129.1	382.7	766.9	---	---
9028	ABCWRBRH	621.0	12065.4	12686.4	1.2E+006	9.2E+005	---	128.7	310.5	12065.4	---	---
9029	ABWWB00A	696.1	13498.3	14194.4	7.5E+005	2.5E+004	---	128.6	696.1	2699.7	---	---
1930	ABWWB00A PP	1250.4	22646.0	23896.4	6.7E+005	2.4E+004	416.8	128.8	1250.4	514.7	---	---
8813	ABWWRB00I	6873.7	21033.7	27907.4	4.1E+006	8.5E+006	2291.2	128.1	---	---	---	21033.7
9066	ABWWRB46I	3764.5	11392.6	15157.1	7.7E+004	4.3E+004	1254.8	131.5	342.2	172.6	---	---
9080	ABWWRB49I	7728.7	22740.8	30469.5	1.9E+006	8.0E+006	429.4	128.2	3864.4	5685.2	---	---
9015	ABWWRBRH	6439.2	18430.3	24869.5	3.4E+006	5.0E+006	2146.4	128.2	---	---	---	---

Table 5 - SDC Device Breakdown Data

Statistical Performance Test Results

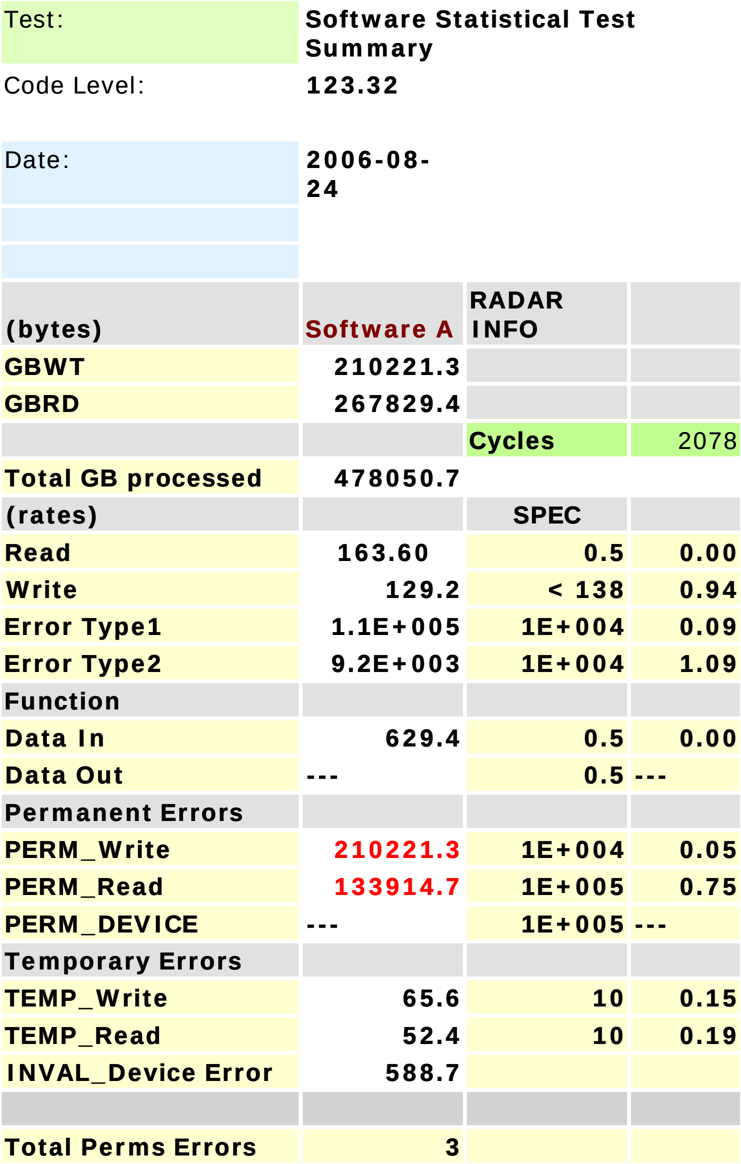
The data storage devices that were used for the statistical performance testing were all at GA level firmware and hardware release levels. The SDC storage device firmware had been tested previously with user experience coverage testing and other testing methods. The storage device firmware and has encountered defects not detected during the user experience testing. All field escape defects were related to data errors when writing to the device. These errors were not detected in the user experience test because the test model was not updated correctly to bound the new data write error recovery sequences that were introduced in the last release level of firmware.

The statistical specification test model did not detect the data write error recovery sequences defect in the device firmware initially. The mechanism required for the firmware error recovery defects to be encountered needs a device to perform marginally. The error recover sequence is not entered until the device writes data that does not match checksum. Over the course of the statistical performance test three devices started to perform marginally. Once the devices started to perform in a degraded manner the firmware defects occurred. These defects did not cause a halt in data flow or cause the device to crash. In Table 5, devices 8810, 9104 and 9078 show the firmware defect. Perm write was out of specification indicating the device had encountered a problem during a data write recovery sequence. After more investigation the data revealed that the statistical performance test had found the firmware defects missed by the user experience testing model. Because the defects required marginal device behavior and the defects were not catastrophic that the user behavior test did not encounter these problems. The time required to execute the Statistical Specification Test was 2 weeks shorter than the User Experience Testing Model.

Conclusion

The Statistical Specification Test was successful in detecting defects that were not detected in the current SDC user experience testing model. As the SDC mission continues to grow the need for testing models like the Statistical Specification Test will grow also. The Statistical Specification Test succeeds where EXMAP failed. The specification test model used to verify the SDC data device firmware did not require the development team to place coverage hooks in the code like EXMAP. Removing the need for hooks allows the statistical performance test to accurate independent of KLOC size. The statistical performance test is also robust enough that it can be deployed to any product that has defined specifications and does not have a defined user behavior model like the current SDC coverage test. The integration of the Statistical Specification Testing to the SDC test portfolio will help SDC to grow its market share by reducing the number of field defects, improving product quality and maintaining release schedule integrity.

APPENDIX A: TYPICAL STATISTICAL TEST MODEL RESULTS



Test:	**Software Statistical Test Summary**		
Code Level:	**123.32**		
Date:	**2006-08-24**		
(bytes)	**Software A**	**RADAR INFO**	
GBWT	**210221.3**		
GBRD	**267829.4**		
		Cycles	2078
Total GB processed	**478050.7**		
(rates)		**SPEC**	
Read	**163.60**	**0.5**	**0.00**
Write	**129.2**	**< 138**	**0.94**
Error Type1	**1.1E+005**	**1E+004**	**0.09**
Error Type2	**9.2E+003**	**1E+004**	**1.09**
Function			
Data In	**629.4**	**0.5**	**0.00**
Data Out	**---**	**0.5**	**---**
Permanent Errors			
PERM_Write	**210221.3**	**1E+004**	**0.05**
PERM_Read	**133914.7**	**1E+005**	**0.75**
PERM_DEVICE	**---**	**1E+005**	**---**
Temporary Errors			
TEMP_Write	**65.6**	**10**	**0.15**
TEMP_Read	**52.4**	**10**	**0.19**
INVAL_Device Error	**588.7**		
Total Perms Errors	**3**		

Error Rates by test device

Code Level 123.32

Date: 2006-08-24

					Radar Rates											
				Specification->	> 1E5	> 1E3	> .45	< 130	> 2.4	>600	> 8.4	> 9.3	>50	> 4E4	> 1E3	> 1E6
Device	**RUN**	**GBWT**	**GBRD**	**Total GB**	**Error type 1**	**Error type2**	**SW**	**Invalid wrt**	**Data in**	**Data out**	**Temp Write**	**C_TRD**	**Inval RD**	**Perm Write**	**Perm Read**	**PDEV**
809	RWCDJEHMMOTIONEC__I1_729	157.2	156.2	313.4	**3.2E+004**	**5.8E+005**	**---**	**129.8**	**---**	**---**	**---**	**78.1**	**---**	**---**	**---**	**---**
455	RWCDJEHMMOTIONEC__I1_729	157.8	155.1	312.9	**1.8E+004**	**3.7E+005**	**---**	**131.1**	**---**	**---**	**---**	**155.1**	**52.2**	**---**	**---**	**---**
51	RWHDJE01INTCJ2EC__I1_729	23337.2	44600.5	67937.7	**1.5E+005**	**1.1E+004**	**103.7**	**129.1**	**---**	**---**	**58.3**	**66.9**	**246.2**	**---**	**---**	**---**
55	RWHDJE01INTCJ2EC__I1_729	9312.9	17997.7	27310.6	**1.6E+005**	**6.5E+004**	**300.4**	**128.9**	**358.2**	**---**	**95.0**	**47.1**	**1011.5**	**---**	**---**	**27310.6**
57	RWHDJE01INTCJ2EC__I1_729	11120.6	21678.3	32798.9	**2.2E+005**	**9.4E+005**	**171.1**	**128.6**	**---**	**---**	**97.5**	**1083.9**	**400.0**	**---**	**---**	**---**
60	RWHDJEHMINTCJ2EC__I1_729	9652.6	19104.8	28757.4	**1.8E+005**	**5.9E+005**	**79.1**	**128.6**	**---**	**---**	**53.0**	**616.3**	**239.6**	**---**	**---**	**---**
57	RWHDJEHMINTCJ2EC__I1_729	0.0	0.0	0.0	**---**	**---**	**---**	**---**	**---**	**---**	**---**	**---**	**---**	**---**	**---**	**0.0**
54	RWHDJEHMINTCJ2EC__I1_729	12410.0	23814.0	36224.0	**2.5E+005**	**3.0E+005**	**107.9**	**128.6**	**---**	**---**	**60.2**	**175.1**	**262.5**	**---**	**---**	**---**
805	RWHDJEHMINTCJ2EC__I1_	11119.6	22519.2	33638.8	**1.8E+00**	**9.0E+002**	**72.7**	**138.0**	**---**	**---**	**7.7**	**7.0**	**225.8**	**11119.6**	**22519.2**	**---**

	729				4											
67	RWWAJE01DURBEC____I1_729	4518.9	3514.6	8033.5	1.0E+005	2.2E+004	2259.5	128.5	---	---	376.6	140.6	---	---	---	---
73	RWWAJE01DURBEC____I1_729	4016.4	3012.2	7028.6	5.6E+005	3.8E+006	2008.2	128.2	---	---	286.9	---	---	---	---	---
50	RWWAJE01DURBEC____I1_729	4518.9	4518.9	9037.8	2.5E+005	7.3E+005	2259.5	128.4	---	---	376.6	1506.3	---	---	---	---

TABLE OF FORMULAS

Test Effectiveness

(1- (Field Escapes / (Test Defects + Field Escapes)) * 100 = ____%

(1 – (Field Escapes / KLOC)) * 100 = ___%

Cost of Test Ratio

Test Cost $ / Pre-GA Test Defects = ___K$ cost per test defect

Test Cost $ / KLOC = ___K$ cost per KLOC

Overall Test Duration

((Projected duration – Actual duration) / Projected duration * 100 = ___%

Traditional Execution Capability Projection

((Actual Engine Rate – Projected Engine Rate)/Projected Engine Rate)*100 = ___%

Actual Execution Capability Projection

((Projected Version Duration- Actual Version Duration)/Projected Version Duration) * 100 = ___%

List of References

Boris Beizer, Software testing techniques (2nd ed.), Van Nostrand Reinhold Co., New York, NY, 1990.

Elaine Weyuker, The Cost of Data Flow Testing: An Empirical Study, IEEE Transactions on Software Engineering, v.16 n.2, p.121-128, February 1990.

Hong Zhu , Lingzi Jin , Dan Diaper , Ganghong Bai, Software requirements validation via task analysis, Journal of Systems and Software, v.61 n.2, p.145-169, March 2002.

Duran J. W. and Wiorkowski J. J., "Quantifying software validity by sampling," <i>IEEE Trans. Reliability</i>, vol. R-29, no. 2, pp. 141-144, June 1980.

Musa J. D., "A theory of software reliability and its application," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 312-321, Aug. 1975.

Whittaker, James A. What Is Software Testing? And Why Is It So Hard?, IEEE Software, v.17 n.1, p.70-79, January 2000.

Horgan, Joseph R., Saul London, Achieving software quality with testing coverage measures, Computer, v.27 n.9, p.60-69, September 1994.

Miller, Keith W., Noonan Robert E. ,Park, Stephen, David M. Nicol , Branson W. Murrill , Jeffrey M. Voas, Estimating the Probability of Failure When Testing Reveals No Failures, IEEE Transactions on Software Engineering, v.18 n.1, p.33-43, January 1992.

New Economy. University of Toronto (2006). http://home.utm.utoronto.ca/~mckee/Index.html.

Cheung, R. C., "A user-oriented software reliability model," <i>IEEE Trans. Software Eng.</i>, vol. SE-6, Mar. 1980.

Piwowarski P., Ohba M., Caruso J. , Coverage measurement experience during function test, IEEE Transactions on Software Engineering, v.15 n.1, p. 287 - 301, 1993.