Management of Test Case Aging: The Generation of Fine Varieties of Tests

Jon Hagar
Lockheed Martin Astronautics
P.O. Box 179
Denver, CO 80201
jon.d.hagar@lmco.com

Abstract: When a test plan first enters life, it can often be like a fine wine, still good, but not fully mature in character, depth and complexity of test cases and objectives. But like that fine wine, as a product and an organization's processes mature, so, too, should the test plan and associated cases. This presentation will report on an analysis of a suite of tests and methods that have matured over many years. Most software projects spend their "life" in maintenance and updates. During these activities a large percentage of money spent on the software will be consumed by testing. This presentation examines aspects of testing from initial through mature stages of an in-use software product. Analysis defines the impact of software trouble reports and change requests, including impacts from system usage on the testing. Percentage distributions between early test levels, objectives, and methods are compared with distributions from the organization as it evolves and matures. While data will be from a single product domain area, extensions and lessons learned to other software domains can be reached. This presentation reports ongoing development, so questions and items currently under study are also considered.

1. Introduction

Lockheed Martin Astronautics (LMA) in Denver Colorado has produced critical software systems for several decades. Production systems are embedded applications that must work the *first time* or hundreds of millions of dollars may be lost. These systems are typically very complex, consequently failures or errors could be introduced from many sources. These software systems have the following characteristics: real-time; spacecraft/booster flight control; minimal human intervention possible; and numerically intensive calculations of such critical items as, trajectories, flight dynamics, vehicle body characteristics, and orbital targets. Development programs are small, usually under 30,000 source lines of code, yet these programs are critical to the control and success of the flight system. Systems with software produced at LMA include the Titan and Atlas family of launch vehicles, upper stage boosters and spacecraft, as well as the associated ground systems. An example mission profile is depicted in figure 1. Production of software on many of these systems followed an historic and similar development process that has been, in part, responsible for each program's success. These processes include continuous improvement and evaluation efforts designed to make things better.

Being a government-military contractor requires a certain commonality and consistency of approach due to compliance with numerous standards. Software engineering efforts like the Software Engineering Institute's Capability Maturity Model (SEI CMM) are based on the idea that similarity and consistency of process over time and project are good. The CMM allows for orderly process change, and this paper examines how the testing evolves as a program continues in maintenance efforts with associated maturing of products. While the paper is based on observations from a narrow domain, it is reasonable to expect similar changes in test cases and processes in other software domains, since many of the practices are common to the industry test practices in general, e.g., unit, integration, and system level testing.

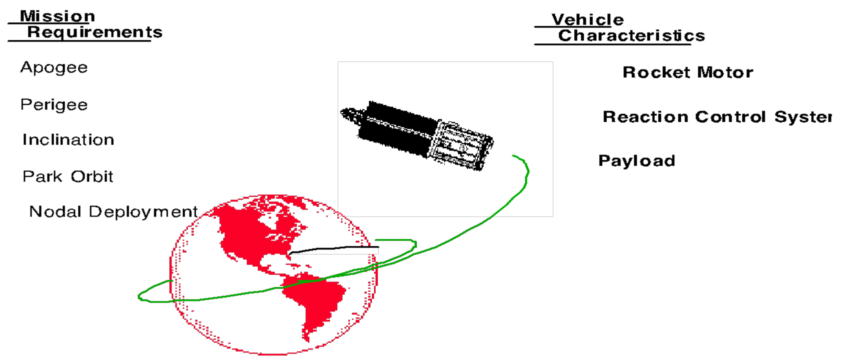

Figure 1 – Typical LMA System with Complex Software Requirements

This paper relates a generalized process that has been applied to numerous critical software programs at LMA, some experiences in testing, and the changes in testing over time. The basic processes of engineering and testing are introduced. Then the presentation outlines how testing is maturing and changing within these process. We find that, over time, testing goes through a series of maturity levels of an initial (infant), middle (teen), and maturing stage (adult).

1.1 Introduction: Lifecycle

In our basic engineering approach, the engineering disciplines are employed early-on during concept development and requirements analysis, which are often associated with a proposal or a new contract. These efforts start with requirements defined at a system level (System specification and/or Interface Control Document). Requirements are then allocated in a software requirements specification (SRS). All of these requirement-based documents are written in English with some use of mathematical expressions. For example, the stability control laws of a booster will be expressed in mathematical terms. The responsibility to produce the higher level requirements documents lies with the systems group, particularly in this case, where a control expert is required.

The SRS reflects a joint responsibility between systems and software engineering, with a software-system requirements engineer (or team) having direct responsibility for each allocated software requirement. Traceability between requirement documents is maintained within databases or traceability matrices. Engineering support tools, such as, RTM (Marconi Systems Technology) or SEDB (a custom LMA database program), are commonly used. In fact, database systems such as these are used throughout the life cycle.

Additionally, at this early point, use of simulations and software tools are employed to analyze the requirements. A variety of custom-built programs and commercial products may also be employed. A common one at LMA is MatLab (Math Works). For example, math equations of the control laws for our booster system can be entered into a specialized simulation program and then subjected to a variety of input conditions to see how the equations (as a part of the overall system) will react. This allows for improvement or “tweaking” of the system. Approaches like this improve requirements and later in the lifecycle these same tools directly support testing.

While we do this early analysis and simulation, we have discovered that there are no perfect requirements. After initial concept, analysis, and decomposition of requirements, production of design and then code begin. In a classical waterfall model of the software life cycle, design continues until complete and then implementation or coding begins. In practice, we find there is usually a rush to get past requirements, get into design and even into implementation. In fact, some or all of these go on simultaneously to a certain extent. This is why the spiral or evolutionary life cycle models are now a standard at LMA. Engineers need to “get their hands on something” and in the case of software, that usually means code or logic that does something or can be executed. This is true whether we are dealing with systems or software teams. This iterative refinement goes on throughout the life cycle, so backtracking and revision of requirements at all document levels is (and required by our processes to be) an on-going part of development. We have started the use of design tools and auto-code generation that in effect link the design and coding processes.

Ultimately, a set of consistent products: requirements, design, and code, are produced by the development team. These have been produced over a number of iterations, reviews, and analysis as well as some development team testing and evaluation. During development and independent from development, formal test has been under planning and development.

1.1.1 Software Test - V&V

Our test process revolves around a series of stages or levels of testing: unit, integration, and system, where each of these may themselves be broken down and/or repeated during spiral development cycles. Test planning, design and implementation start concurrently with development. It is important to note these stages are not “stand alone” end of lifecycle points but in the spiral process we use, integrate and repeat these to varying degrees during cycles.

As part of our process, testing consists of a team of both software and systems engineers. Both, developers during early stages, and an independent test group, conduct testing. Unit, integration, and initial levels of functional tests are done by development staff and overseen by the test team. To ensure that software meets requirements, an independent test team does formal functional and behavioral testing. Formal here means that tests are written, controlled with independent quality assurance organizations, reported, and retained in historic archives. Functional is requirements based testing. And behavioral testing examines both the required and designed characteristics of the system. All tests, developer based, and independent, are subject to walkthroughs and/or team reviews both prior to execution. Team reviews are also used after testing, as results are approved and signed off. Test teams that have members who support all development lifecycles stages, have both advantages and disadvantages.

An advantage is that these engineers are responsible for defining testable requirements and designs in the first place. A requirement that is testable is better than one that is not. These engineers also understand what the system should be doing and so can define testing and stress testing quicker that an engineer with no history with the requirements. However, test prejudice and “blind-spots” on the requirements and software are concerns when using these engineers to support testing.

To compensate, the independent test team has responsibilities for the test planning, design, and execution. This additional staff is combined with the development engineers to form the flexible and evolutionary test teams. This combination of software and systems, enables a comprehensive V&V testing effort. This effort combines people, the verification and validation process, and test environment to show compliance of code to standards, e.g, software development standards, company standards, customer standards, but more importantly, to identify any anomalies in the software-system.

The different levels/stages of testing allow errors to be driven out nearest the lifecycle point where they were introduced. For example, we have incremental drops of software products, we will complete some aspects of testing for each stage depending on risk and functionality of the product.

Table 1 - Standard Sample Tools

Activity	Tool	Function	Benefit
Verification	Battlemap and Adatest	Coverage	Measurement of test
Verification with model – Full Modules	POSTII	System simulation	Assessment of data values testing
Validation	Test Environment – FAST	Execution of software	Assessment of software realistically

Developer based efforts using unit and integration testing accomplish verification. Verification shows compliance of the code to design, design to requirements, and even a binary executable configuration to its source files. We treat the higher level product as "truth" and test to show it is correctly transformed into the next level. Validation on the other hand tests that the requirements, design, or code does what "works" and is done at the system level. Validation is a much harder question and requires the human expert to quantify "works". For example, in validation, we look to see if the control system has sufficient fuel to perform the mission orbit conditions, given things like vehicle and spacecraft characteristics.

Our verification efforts concentrate on the detection of programming and abstraction errors. Programming faults have two subclasses of computational or logic errors and data errors. In Verification, we practice white box or structural testing to very low levels of the computer, including a digital simulator or a hardware system, such as, an emulator. At this level, verification testing is done to ensure that the code implements such things as, detailed software requirements, design, configuration controls, and software standards. This testing is usually done at a module-level or on small segments of the code which are executed somewhat in isolation from the rest of the system. For example, as shown in Table 1, we use the Battlemap [McCabe and Associates] and/or Adatest [IPL] tools to define our test paths, so that we get complete coverage at a statement and branch level. This type of testing is aimed at detecting certain types of faults and relies on the coupling effect in errors [Offutt-92]. A complication of this level of testing is the comparison to success criteria and the review of results. These are human intensive and time consuming although some use of automated comparisons based on test oracles has been achieved [Hagar-95].

Verification testing detects compiler-introduced errors, as well as human programming faults. Our test aid programs (tools) and computer probes allow the measurement of various types of program code coverage (statement up to logic/data paths). Success criteria are based on higher level requirements in the form of English language specifications and/or design information, as well as an engineer's understanding of how software should behave. Verification is conducted primarily by software engineers or computer scientists with some aid from other members of the whole team, such as, systems engineers. This is possible because the higher level "requirement" that is being verified to is taken as whole—complete, and good. Transformation of requirements-to-design, design-to-code, code-to-executable, and hardware-to-software interfaces, *all* can experience deductive errors that may result in failure. Verification at LMA has found anomalies in and is targeted at each of these development steps.

Verification by development staff continues during integration testing and what we call full module testing. Full modules are integrated units of code that perform a function or correspond to an integrated object. Units of code are integrated and test as a whole during this testing. Testing exercises the interfaces between units of code. As an option during integration, we use computer simulations to analyze functionality. This can serve as oracles for later testing. Each simulation or model is specifically designed to concentrate on one error class (deductive or abstraction) and function of the system (control, guidance, Nav, etc.). These simulations are higher order, non real-time models of the software or aspects of the system, usually executing on a process other than the target computer. At this level, our simulations are design-based tools, and they simulate aspects of the system but lack some functionality of the total system. These tools allow the assessment of software for these particular aspects individually.

The simulations are done in both a holistic fashion and on an individual functional basis. For example, a simulation may model the entire boost profile of a launch rocket with a 3-degrees of freedom model, while another simulation may model the specifics of how a rocket thrust vector control is required to work. This allows system evaluation starting from a microscopic level up to a "macroscopic" level. Identical start-up condition tests on the actual hardware/software can be compared to these tools and cross checks between results made. Often aspects of the actual code and algorithms are incorporated in these full module test tools. The results from these runs and tools can then be used in higher levels of testing and analysis.

Verification tests the code, design, and requirements at a low level. Test results are reviewed and approved by teams, but these efforts by themselves are not sufficient. Validation continues where verification leaves off

Validation is conducted at several levels of "black box" or functional testing. We test the software extensively in a realistic, hardware-based, closed-loop feedback, test environment. The other validation level is requirements-based analysis by systems engineering to assess the correctness of the requirements themselves. This paper does not consider validation by the systems engineers and the associated system modeling.

In the major aspect of validation, we develop a comprehensive test environment. This is very important in our experience, and we attempt to replicate some or all of the actual hardware of the system whose software we are trying to V&V. These environments can be very expensive to create (cost figures are directly dependent on the complexity and size of the system) but are the only way to test the software in a realistic environment. Some of our test facilities at LMA include ground operation systems, ground cabling, and vehicle configuration. However, there are aspects of the critical systems that cannot be fully duplicated in a test environment and thus must be simulated.

Typically these test environments use supporting computers, workstations, and programs that replicate the functions that a completely hardware-based test system cannot. There are always questions of fidelity and accuracy of these models, and we have had problems in these areas in the past that have resulted in lost time and efforts. Consequently, we take great care in the test environment set-up area.

Validation testing on the hardware-based test bed is done in nominal (expected usage) and off-nominal scenarios (stress and unexpected usage). This "real world" systems-based testing allows a fairly complete evaluation of the software even in a restricted domain. In addition, unusual situations and system/hardware error conditions can be input into the software under test without actually impacting hardware. For example, we can choose to fail attitude control thrusters, so that the control software we are testing is forced to react to a set of hardware failures. Validation testing is aimed at "breaking" the software to find errors, even more than it is at the nominal test cases, which seek to show the software is working. Failures in software at this "system" level receive the most visibility and publicity, and we seek to have 100 percent mission success. [Howden 91] argues that the goal in V&V is not correctness but the detection of errors. We agree with this and practice testing consistent with it. Each of the tools shown in Table 1 has been successful at detecting errors that would have impacted system performance. Thus, they are credible test aids.

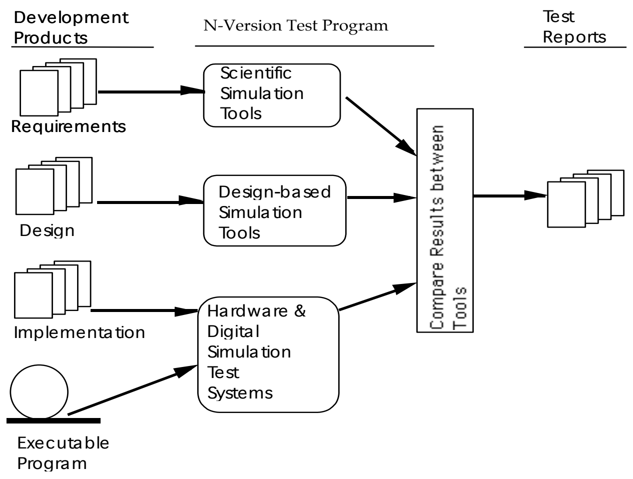
Development Products
N-Version Test Program
Test Reports
Requirements
Scientific Simulation Tools
Design
Design-based Simulation Tools
Implementation
Hardware & Digital Simulation Test Systems
Executable Program
Compare Results between Tools

Figure 1.1.1-1 - Test Tool Levels

This software is responsible for a variety of critical functions, all of which must work for a booster system to go from its power-up state (usually on the ground) to some final orbit condition. The software interacts with hardware, sensors, the environment (via the sensors and hardware), itself, operational use timelines, and possibly humans (if a ground command function exists). As shown in figure 1.1.1-1, there are mission performance requirements and vehicle characteristics, which influence the software. As a minimum, the system test/validation process will employ and be reviewed/approved by the following kinds of systems engineers: Test engineers, Software, Controls, Mission Analysis, Guidance, and Navigation, and Electronics.

2. Test Evolution

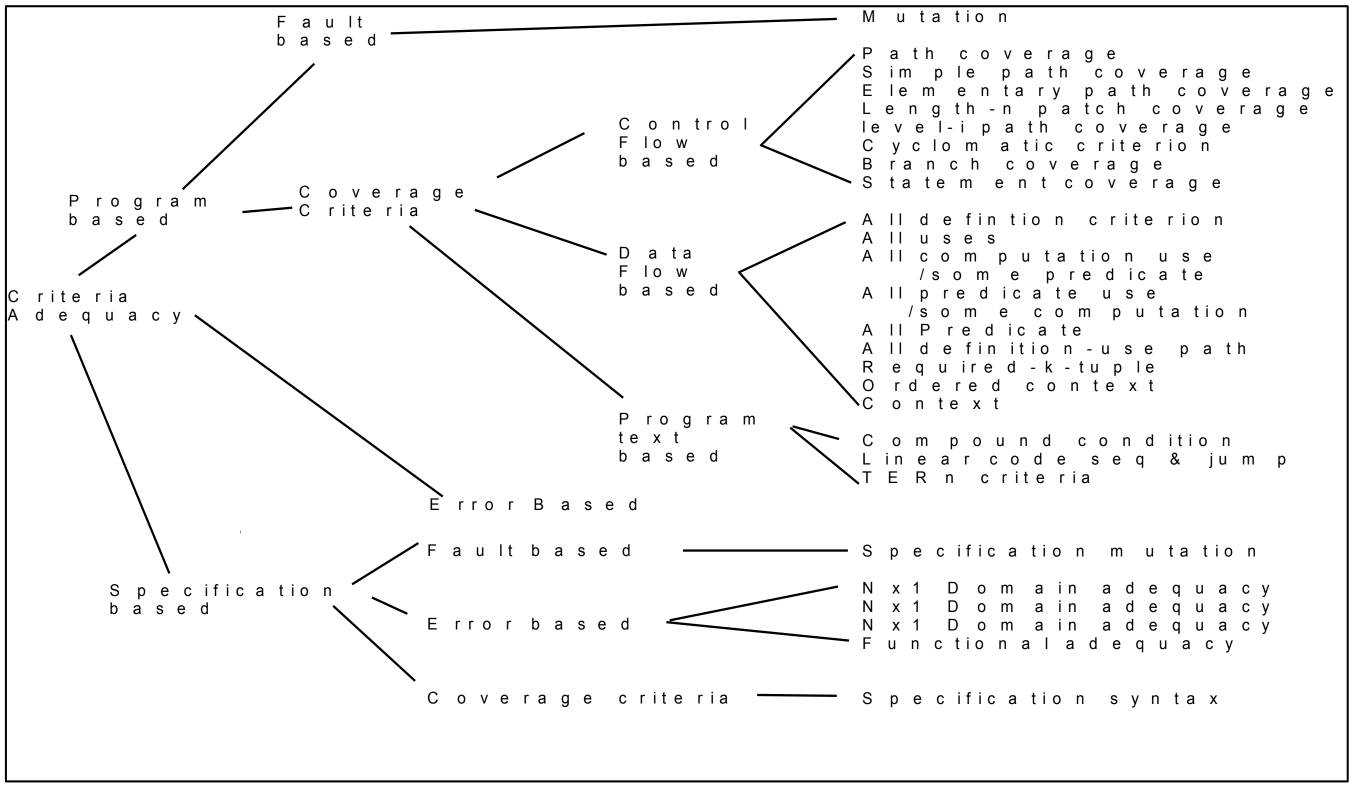

Figure 2-1 Classification of Test Data Adequacy Criteria

The first part of this presentation outlined our basic engineering and test processes to set the context for how our testing matures. The following sections outline how some of the specific test techniques and test cases have been observed and measured to change over time. It is important to note the on-going and product (flight) products. Not all results and data are finalized.

To categorize the changes in the test process we are using classification criteria as seen in figure 2-1, which is taken from [Gardiner 99]. Process evolution has been one of the larger areas of change. We have progressed from older custom build tools to generic commercial products. While doing this we have taken the best from the old and added the new.

2.1 Verification: Unit level

Unit level testing initially consisted of achieving statement and branch coverage. This initially was done with custom-built tools, test cases, and a fair amount of pain. This limited the number of test case instances that could be run (table 2.1-1). Further, during maintenance, only changes tend to be unit regression tested, i.e.; a whole suite was not always run.

We have recently progressed to using commercial tools with a higher degree of automation and a larger coverage (see maturing data of table 2.1-1)

Table 2.-1 Unit Testing

Effort:	**Criteria (right side)**	**Tests - Sample 1**	**Sample 1 Measures**	**Tests - Sample 2**	**Sample 2 Measures**
Initial	Statement & Branch	38	Code units - 6 Cy peak - 16	20	Code units - 8 Cy peak - 10
Maintenance	Statement & branch of only changes during regression	12 * 4 (repeated 4 times)	Units - 6 Cy peak – 16 Changes - 4	9	Units - 8 Cy peak – 10 Changes- 1
Maturing – new development	Cyclomatic number and ordered context coverage	332 (large data structures)	Units - 3 Cy peak - 2	70	Units - 8 Cy peak - 12

Notes: Cy – McCabe's Cyclomatic complexity number.
Code units – number of units of code (separate compilation) in sample.
Changes are counted by individual “function” and not total lines of code changed, e.g., a fix or change will impact a single function of the unit of code but may impact more than one line of code.

Observations:

1. For the mature regression testing process to be used during maintenance, all of the unit tests will be rerun. This is an improvement over the regression methods initially used.

2. The more mature unit test process increases the level of coverage to include cyclomatic complexity which includes the other lower levels of coverage. We have also expanded our coverage into the data flow based criteria. In earlier stages data was only selected to drive control flow and a few “interesting” data cases.

3. The test automation allows a larger number of tests because of features like automated success criteria checking and metrics that before had to be “hand” generated.

4. What is not clear is the impact to error density in later life cycle stages, i.e., is more really better. Defect trends and density are currently be monitored; though they are not ready for reporting in this paper.

5. While not listed in the table, each of the methods had limited specification based coverage criteria of the specification's syntax.

6. Additionally, most tests set cover compound conditions (limited).

7. During earlier program efforts, maintenance testing was primarily aimed at the software fix or change (e.g. a smoke test). This did include limited regression testing, but not all the unit tests were executed. This was due to nature of the changes and to conserve time/effort. A more comprehensive approach to regression is to execute all tests, which is what will be done in the maturing approach. The more mature methods will mitigate some of the regression risks of partial execution.

2.2 Verification: Module and integration testing stage

Module testing in the initial stages was based on the execution of the integrated package (series of functionally related units) and/or simulation models as described in section 1.1.1. This approach introduces coverage of specification, error based, functional coverage (see figure 2-1), by executing the integrated package over a number (large number) of specification based data conditions. This technique was used in both initial and maintenance efforts. It proved effective in that it found errors before we entered system testing and lost "visibility" into some code functions.

These methods are continuing as we mature, however we are increasing number of tests in the integration level by adding test cases from the unit level. We have found that unit test information can be integrated with drivers to test a whole series of units in the integrated module/function. This can be done quickly and results automatically compared. Also, additional integration data sets are being added that cover cases that cross unit boundaries. This appears to increase out level of coverage, but we have not gone far enough with this maturing to see:

1) Are more or different errors being found than historically?
2) Does this complement or impact historic efforts?
3) What are the cost impacts?

2.3 Validation: System stage

Software system level testing matures in the following ways.

1) Initial numbers of tests are larger, to get to a baseline product; then as the product enters maintenance and use, the numbers of tests are fewer (regression problem);
2) During initial testing of a new products, the test plans "grows".
3) The nature of the test set changes, as tests come to reflect the nature and problems of the product.

2.3.1 Decreasing tests as a product matures

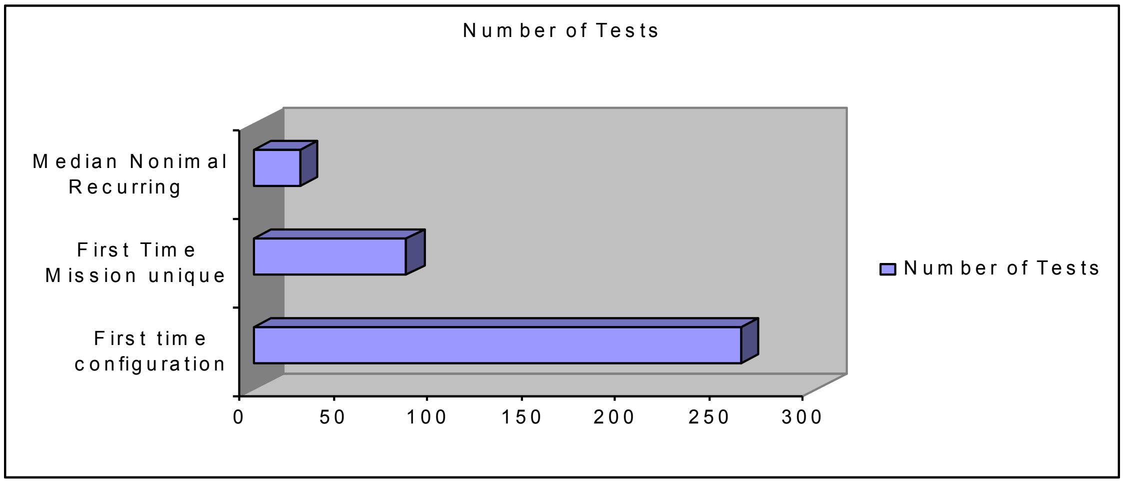

Figure 2.3.1-1 Tests decrease over time (more mature usage cycles are on top)

Figure 2.3.1-1 shows the “shrinking” of the tests that are executed per product usage (release). The bottom two bars show a new version of the product with major changes and new functionality. It required extensive testing prior to first use. This was done in two phases. The first configuration covered the generic functionality of the system. Following this, a first usage (mission) resulted in about 25 percent fewer tests, with the tests aimed at this particular usage, i.e., not all generic functionality was tested. Finally, historic data shows that once a software version goes into maintenance (small or no logic changes but new data/parameters), the number of tests fall to about 10 percent of the first time configuration. Note: the 10 percent number is a median value taken over 30 software releases.

The majority of the 10 percent number are regression in nature. We define regression suite as tests that are aimed at the old functions, minus any removed features, plus the test of any new features or fixes. The make up of a typical regression suite of tests is based on a nominal mission run and one or more stress cases. If these regression tests are not sufficient to cover the new/change functions, then additional tests are added to the test plan. The assessment of the tests are needed and made during software review board meetings and include the development, test, and systems staff associated with each change or data set.

Additionally, the 10 percent of tests include one or two tests drawn from the historic suite of tests (lower two bars) that test functional areas of the system. The historic suite is made up of all tests ever run at the software system level including the first time and mission based tests. With this approach over time, all the “classes” of tests are cycled through and executed. This method of adding new, regression, and historic tests has turned up problems in the regression suite, code, and/or test environments. This approach of cycling through tests represents a maturing of the test planning process (it was not done initially).

2.3.2 Maturing within a test plan cycle

The maturing of the test plan can be seen as more tests are added to it. Figure 2.3.2 –2 shows this. During a major product upgrade, the test plan started at 195 tests. By the end of the testing cycle, 362 tests had been completed. The tests were executed, analyzed, errors detected and based on these errors, product changes made to the software. Not all the increases in the number of tests were the result of errors or changes in the code. About 20 percent of the increases appear to be due to tests aiding the understanding of the staff. As the staff understood the software and system better, they then wanted more tests to check of features and behaviors of the software. This increase can be viewed as maturing the test plan. The rest of the increase is due to changes in the software (regression tests). The breakdown of these changes, due to errors, can been seen in figure 2.3.2-2. The majority of new tests were associated with design issues. This was reflected in the design/purpose of these tests. The next largest number of tests was associated with requirement changes, and very few tests were associated with other sources.

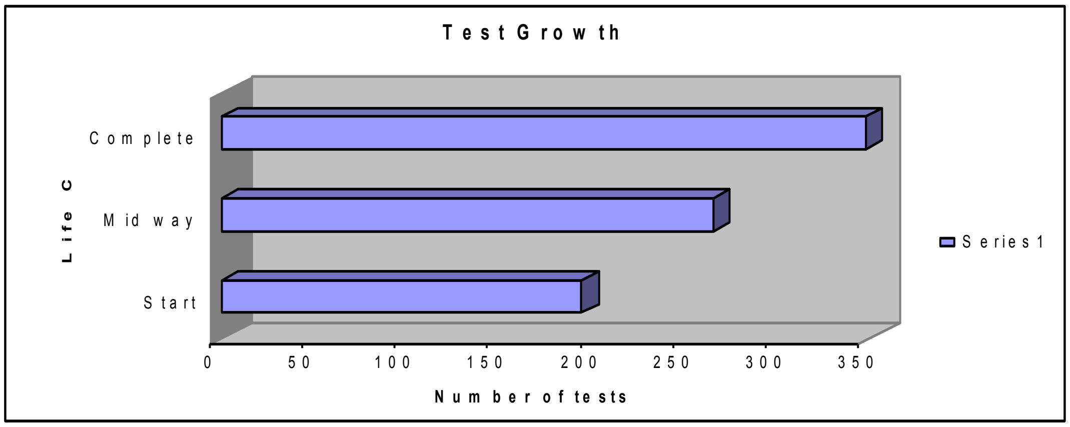

Figure 2.3.2-1 Growth of testing within a test planning cycle

The majority (85 percent) of the added tests were just refinements of existing test cases. The typical changes were to introduce new test environments (data variables or commands) or refine of test sequence procedures. The test cases matured by including different stress cases or numbers that impacted the execution of the software and addressed functionality of the system that had not previously been tested. During the study test cycle, approximately 20 "new" tests were defined during this "maturing" process.

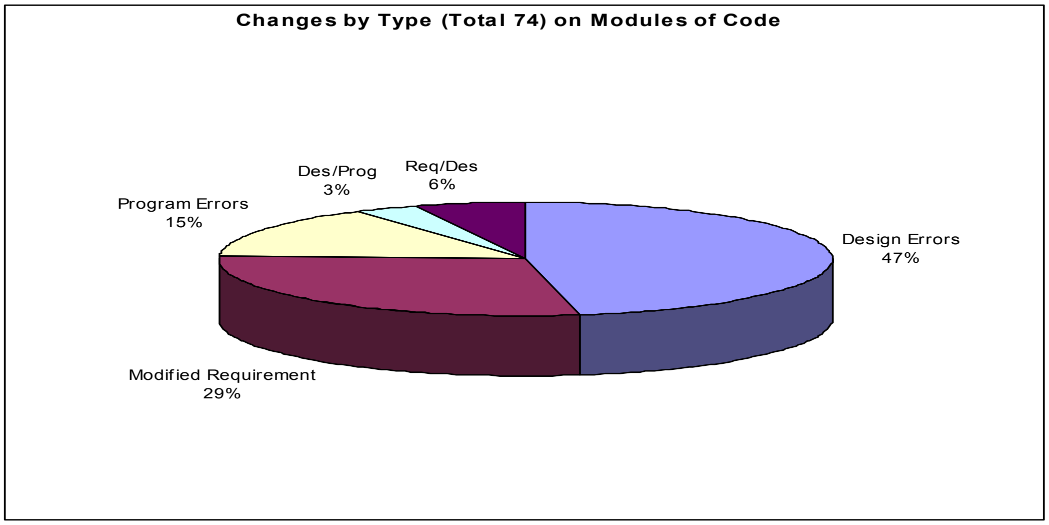

Figure 2.3.2-2 Product/Lifecycle Error Trending

2.3.3 Aging of the Test Plan Data base

The historic test plan database appears to have been in place over about 10 years. There are about 310 tests in our historic base, with 35 of these tests being "nominal". This means that the major (over 265) of tests cover some special function, error, stress, dispersion, or system condition. Figure2.3.3-1 shows what happened during one analyzed test period (covers several test cycles and efforts). The majority (90%) of the added 40 tests were new or modified off-nominal tests One can see from this increase, that as it continues over time, the test plan data base will become more and more weighted toward these off nominal tests.

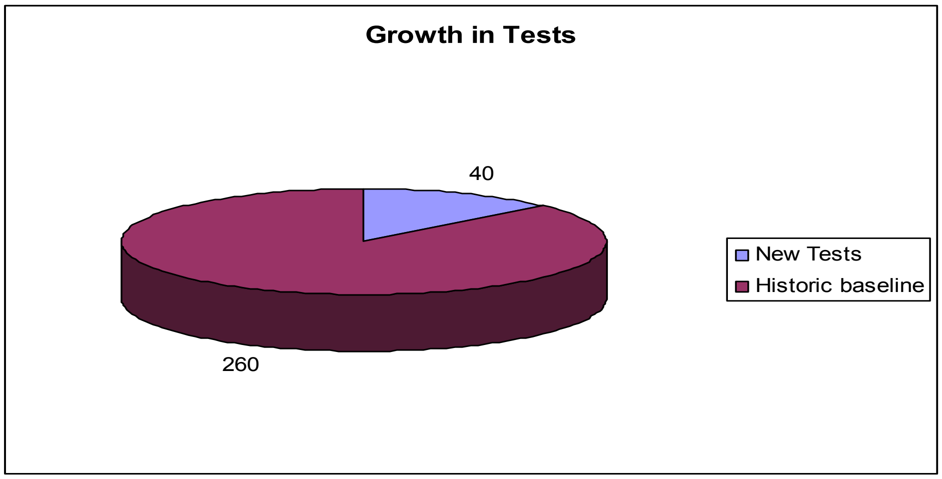

Figure 2.3.3-1 Growth of tests

From this, one question might be, how frequently are the "off nominal" tests being executed, with the concern being that if these are executed more frequently, they would tend to grow because of the frequency of execution. However, at least one nominal test is scheduled for every test cycle and is executed as part of every regression sequence. Nominal tests account for one third of the total tests that are run, even though they account for only about ten percent of the test base. Further, about 55 percent of the non-nominal tests are executed only once or twice during at test plan cycle.

One can surmise that the testing is fairly well distributed, but because errors and features tend to “lie” in the off nominal areas of the design/code, the tendency will be to see increases in these tests. Also, even though many requirements deal with "off nominal" cases (thus validation tests would tend to reflects this), there still is growth above this in the number of "off nominal" tests as the test plans mature.

3. General Observations

1. Regression cases typically are within the existing portfolio of tests, though specific data and use-cases needed to be refined.
2. Rotation through the suite of tests has a tendency to be good in that it exposes errors that might not otherwise be seen.
3. Majority of system-software validation tests are aimed at non-nominal test scenarios. This appears to be a continuing shift over time, i.e., more off nominal tests are created which bias the total count over time.
4. Item 3 appears to be related to the concept raised by people like Boris Beizer regarding what he calls “the pesticide paradox”. Briefly, test may remove one or more errors, but running the same test will not remove errors that have already been “killed” and successfully removed, i.e. the tests become ineffective. And related to this, errors that remain, get harder to kill. We see that data base of testing shifts increasingly toward runs that are aimed at “special” cases (rerunning the same cases over and over did not buy us anything). Further looking at the tests we do rerun, we noted that almost all of them included new values and data (in fact they were not exact reruns). This increased the likelihood of avoiding aspects of the pesticide paradox.
5. Testing moves more toward “earlier” and high levels of coverage with the use of tools that enforce rules. The desire is that this will find errors earlier. (Note one “technique” not discussed in this paper is the use of peer review and structured walkthroughs. While some

consider inspections an aspect of verification and related to testing, we have not detailed them, since it was not within the scope of this paper. However, Peer review and inspections are a key to LMA processes).

3.1 Future Efforts

As an ongoing product area, we practice optimization and strive for continuous improvement. Testing is an important area to look to. It represents between 15-to-50 percent of our budgets (depending on the project). The following items continue to be researched:

1. Our unit testing has increased the numbers of test cases we execute. Will this really decrease errors in later tests and development cycles?
2. Use of unit and integration automation has been seen to improve (speed) regression during the spirals of development. Will this continue to be realized during maintenance?
3. Does the improvement in Unit and Unit-Integration testing, really improve the overall error trends and "speed" (nearness to point in which they were introduced) with which they are found when they are most cost effective to fix?
4. Test growth in system test/validation area appear to be associated with increasing numbers of off nominal tests. This was measured during one major update cycle. A) Is this historically true for all test cycles and B) will this trend continue give some of changes listed in this paper under the maturing test process?

4. Summary

Earlier test plans seem to been aimed at basic coverage and nominal testing. The focus on testing for errors has driven the trends towards off nominal test cases. There were data gaps encountered in this study from extremely early test plans, so we do not have complete statistical analysis. Some anecdotal evidence was gathered by talking with long-term program members, and the data did not conflict with any that the data provided from later efforts. It does appear test plans and cases go through a maturing process, and testers would do well to consider the types of maturing changes outlined in this presentation. The result can be better test plans, schedules, and development efforts.

Early test plan efforts can be summarized as lower levels of coverage. The middle efforts appear to be characterized by the growth of the numbers and types of tests. Finally, maturing plans seem to have a duality. The numbers of tests may very well increase, and these tests have higher levels of coverage. This appears to be associated with improvements in methodology and tooling. The other nature is that the numbers of tests decrease once the errors have been "driven out" (during maintenance). This results from a decrease in the number of tests associated with Validation. The long-term impacts of these trends have not been determined, nor has the impacts of increasing the numbers of tests in areas like Verification/unit testing.

Reference

S. Gardiner (ed), Testing Safety-Related Software, 1999.

J. Hagar and J. Bieman, "Adding Formal Specifications to a Proven V&V process for System-Critical Flight Software", *Workshop on Industrial-Strength Formal Specification Techniques*, April 95.

W. Howden, "Program Testing versus Proofs of Correctness," *Journal of Software Testing Verification and Reliability*, Vol. **1**, Issue **1**.

A. Offutt, "Investigations of the software testing coupling effect", *ACM trans. of Software Engineering and Methodology*, Jan. 1992.