Testing Critical Software: Practical Experiences

Jon D. Hagar
Lockheed Martin Astronautics
Denver, Colorado, 80201
hagar@den.mmc.com
(303) 977-1625

Greg Green
Lockheed Martin Astronautics
(303) 977-1763

Abstract: This paper presents our experiences in testing critical software that supports flight systems developed by Lockheed Martin Astronautics in Denver, Colorado. This approach has not been proven in an academic sense, but has been demonstrated over the years to result in software that successfully performs missions. It is based on teams comprised of the correct skill balance in software and systems engineering, as well as using a defined process.

Keywords: Flight Control, Test, Verification, Validation, Systems Engineering

1. INTRODUCTION

Lockheed Martin (formally Martin Marietta) Astronautics (LMA) in Denver Colorado has produced critical software systems for several decades. Production systems are usually one of a kind 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 (with small staffs), yet these programs are critical to the control and success of the flight system. Systems with software produced at LMA include the Titan 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. However, an important aspect of the overall process is the human factor, which *must* be considered.

Some authors and researchers would have you believe that the process is most important. You select the right or "latest & greatest" method; a good set of techniques; buy your support software (tools); add in schedule and budget to support these; and *presto,* you have a reliable and safe critical software system. From our experience, some of this is true, however, we advocate that you cannot overlook the human factor. *Good engineers who think are necessary* for the success of the process, and there is no substitute for human thoughtfulness. Additionally, no single engineer can understand all expects of these complex systems. This requires the use of teams with diverse skill sets, as well as a good process.

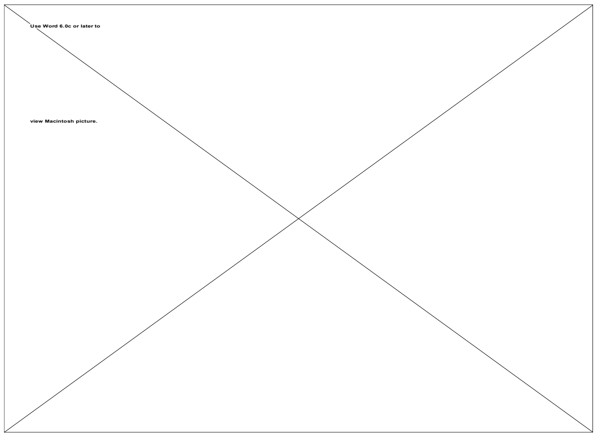

Figure 1 - Complex Software Requirements

This paper relates a generalized process that has been applied to several critical software programs at LMA, some experiences in applying the process, and the importance of the engineering team. We concentrate in the area of testing or Verification and Validation (V&V) of these systems, set in the context of the overall development concepts (processes) employed at LMA.

2. A COMMON APPROACH BASED ON A SYSTEM VIEW

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. In this paper, we've been asked to define how we work on critical software systems and relate what methods work for us. These questions have been asked by others and are of ongoing interest in LMA. Our answers, to date, seem to revolve around two important concepts.

1. Critical software must be developed and tested with constant consideration of the "system".
2. A standard, planned, and documented process carried out by responsible and knowledgeable humans is a requirement for success.

At LMA, we achieve success in system-software engineering by keeping the associated personnel of systems and software engineering involved *throughout* the development life cycle. This includes software testing/V&V which is a major topic in this paper. We feel V&V, from a systems perspective, is important to critical systems since we find between 25 to 50 percent (experience of authors and Beiser-84) of development costs are consumed in various software testing/V&V efforts, plus we notice many programs appear to encounter schedule, cost, or technical problems during the testing/V&V effort.

Our system engineering personnel are involved at the beginning of a project. For example, in the flight control software of a booster rocket. 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, there are mission performance requirements and vehicle characteristics which influence the software. As a minimum, the example software will employ the following kinds of systems engineers at the time of requirements definition: Controls, Mission Analysis, Guidance, and Navigation, and Electronics.

3. SYSTEMS ENGINEERING

In our basic approach, the above engineering disciplines are employed early-on during concept development and requirements analysis, which are often associated with a proposal for a new contract. These efforts start with requirements defined at a system level (A-level "spec" or System "spec", or Interface Control Document, according to military standards). Requirements are then allocated in a software requirements specification (or 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 requirements 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 (The 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.

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 can 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 ongoing part of development.

Ultimately, a set of consistent products — requirements, design, and code, is 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.

3.1 Software Test - V&V and Environment

Much of our test/V&V approach is common with industry, but the LMA test/V&V has some noteworthy aspects.

As part of our process, test engineers are a team of both software and systems engineers. We use the same types of engineers, and on some projects a few of the very same people that define the system requirements also implement the system software-level testing. This is the "formal" testing that is done to ensure that software meets requirements. (Formal here means that tests are written, controlled with independent quality assurance organizations, reported, and retained in historic archives.) A test team that has members from the requirements development staff has both advantages and disadvantages.

An advantage is that these engineers are responsible for defining testable requirements 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, prejudice and "blind-spots" on the requirements and software are a real problem for the "reuse" of these engineers.

To compensate, additional engineering staff, yet with the same skills, are used to support V&V. This additional staff is combined with the "reused" engineers to form the test team. On other projects the test team is completely independent from the development staff, but the skill base (or types of engineers) is the same. Additional to the skills base, projects should have on staff experienced or senior engineers, particularly in testing/V&V, as has been observed by [Deutsch 88 (and others)]. These requirements are considered when building a test/V&V team. This combination of software and systems as well as experienced and inexperienced engineers, 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, but more importantly, to identify any anomalies in the software-system.

Many of our V&V testing concepts originated in Air Force programs that needed to achieve high reliability. We defined V&V in section 1 and here we expand the definition with some examples.

Table 1 - Standard Sample Tools

Activity	Tool	Function	Benefit
Verification	Battlemap	Coverage	Measure - ment of test
Scientific Validation	Booster Utility Program	3-degrees of freedom simulation	Assessment of data values testing
Validation	Test Environment	Execution of software	Assessment of software realistically

In verification, we test to show 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". This 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 tool [McCabe and Associates] 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 [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 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. 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 or design-based analysis programs or what we call "scientific validation" tools. Validation has a larger (than verification) concentration of manpower, time, and testing. This is because the validation question ("Did we build the right software?") is much harder to answer. Validation staff are mainly systems engineers, and their skills mirror development.

In scientific validation, we use extensive computer simulations to analyze requirements and serve as oracles for the actual "black box" results of the software. Each simulation or model is specifically designed to concentrate on one error class (deductive or abstraction) and the level of the system. These simulations are higher order, non real-time models of the software or aspects of the system. Our validation efforts start at what would be considered integration testing within the industry. 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 (e.g., Booster Utility Program), 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. We use the scientific validation tools as oracles for the test environment as well as "stand-alone" analysis aids.

In the other 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 are done using the 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.

V&V/testing on the hardware-based test bed is done in nominal and off-nominal scenarios (stressing testing). 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. Thus, after requirements based "acceptance" testing, which we must do, the concentration of our testing is directed at finding the significant and catastrophic failures that result in mission loss or unacceptable degradation of system performance. Failures in software at this "system" level receive the most visibility and publicity, and we seek to have 100% 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.

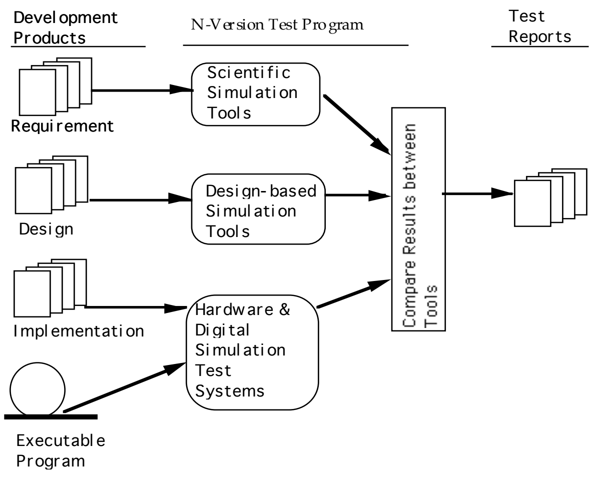
Development Products
N-Version Test Program
Test Reports
Requirement
Design
Implementation
Executable Program
Scientific Simulation Tools
Design-based Simulation Tools
Hardware & Digital Simulation Test Systems
Compare Results between Tools

Figure 2 - Test Tool Levels

3.2 N-Version Testing

Our approach can be likened to a multiple version (or N-version) based testing methodology. This is because we have simulations and partial implementations of the system which have been generated separately from the software development team and are used in our V&V. Figure 4 depicts the different levels commonly found on our projects. We have the hardware test environment at the bottom of the figure that was previously talked about. We also mentioned above that we create design based simulation tools that replicate a function or groups of functions of the software. And finally, at the top we have the software programs that are based on requirements including possibly several different levels of requirements documentation. In our view, these form the equivalent of an N-version programming system. This is because these tools are compared at different points to each other. In one of our project areas, we have a minimum N=3, and a maximum N=8 during all test phases. Thus, we compare and check points of at least “N” different programs and/or levels. These numbers of versions represent a typical test program for us and has remained constant over the years, though individual tools and checks have changed.

N-version programs have been criticized for not significantly aiding in reliability [Knight and Leveson - 86]. Knight’s and Leveson’s experiments indicate that different teams tend to make the same mistakes in “independent” versions of software. The underlying assumption of N-version is that independently coded programs fail independently and Leveson’s experiment proved this to be incorrect in the general case. This has been, and continues to be, a concern to us. However, our use of the N-Version program-based testing has been successful in detecting errors. And we and others like [Houden] judged the value of tools on their ability to detect errors.

For example, one program area can be characterized as:

- long term - over 15 years
- multiple numbers of missions - 13 flights
- critical and complex - responsible in real-time for all flight guidance, navigation, and controls activities of booster system
- size - nearly 10,000 lines of source code.

A team has tested and analyzed this system in a redundant Independent V&V effort (IV&V). This effort has yielded over 1,500 errors. Error counts are spread out over many revisions of the software and higher error counts, as expected, occurred during the early development phase of the program. Approximately 1 percent of these errors, if undetected, would have significantly affected mission performance or mission success. This data is typical of our other program areas. Some of these errors have been found by the N-Version testing approach in all phases of the software’s maturity. The approach has found the following specific kinds of errors:

- compiler-generated logic problems
- differences between requirement-specification and code equations
- differences in logic between requirements and control logic
- differences in numeric representation of floating point values
- differences in requirements-to-design deduction
- problems between development products and test products
- incorrect data load and initialization values
- differences in tools between development and test groups.

Most of the 1500 errors were uncovered early during initial testing. But the comparison process shown in figure 4 has continued to expose software anomalies even after 10 years of testing. Most of these in the later years have been classified under the last two of the above factors.

The production of the N-Versions of software tools serves the following functions:

- as oracles to aid the human in judging when a software unit under test has worked by providing comparison data points;
- as error identifiers when differences between versions occur, they must be resolved; and
- as part of a process that requires the human engineer to rigorously think about the problem that is being solved.

Thus, while N-Version Programming is in question, our use of different programs in testing has detected a variety of errors. We would argue that the statement for N-Version should be "that independently coded programs fail independently sometimes". And this is sufficient to allow their use as test tools, if we recognize their limits, i.e., not all errors or failures will be found.

We employ supporting methods to ensure some of the problems of N-version are partially mitigated. We use numerous cross compares where differences must be resolved. Tools or simulations are developed in a controlled software development process and are subject to peer reviews to add an improvement over just having a human read or review the code or products. Thus, we have always looked for improvement. This has included increasing levels of automation and data "inter-changers" (between programs), and limited application of formal methods [Hagar-95]. We recognize the limits, and would not claim N-version based testing is a complete answer in and of itself, however, it is one "trick" in our bag for testing and has been successful at aiding in error detection on real software versus academic experiments.

4. EXPERIENCES, ADVANTAGES-DISADVANTAGES

The basic approach to critical software outlined in this paper has many advantages. The first advantage is that it works. The basic approach has been in use and evolving since the first space flights. It has been documented in our company standards. Further, the approach is adaptable. Variations on this theme have proved possible over the years. Finally, the approach eliminates some types of problems and has checks and balances that successfully detect problems introduced during development. Additionally, since the software and systems engineers are a composite of people who both defined the initial system and/or have similar domain knowledge of what the software should be doing from a concept stand point, we can detect faults in the software that are associated with missed or poorly abstracted software requirements. This is accomplished by detailed inspection and review of analysis results from the automated tools and software under test runs. This is a subjective activity, but we establish as much objectivity as possible with rigorous controls over the test process.

Despite the advantages, things are not perfect. We have already outlined the issues of N-Version. Our approach has dependency on humans. In a purest world of "process, process, process", this dependency is disconcerting to some in the software community. However, we feel less worrying should be done about removing every aspect of the "human equation" and more interest should be directed toward the things in software engineering that get done and those that produce working systems.

As noted in the previous discussions, the process is dependent on a thorough, consistent development process as well as engineering judgment. Requirements are sometimes open to interpretation and interactions between subsystems can be difficult to predict. To compensate, extensive testing/V&V is required as well as picking the right people to minimize the chances of errors.

Availability of skilled software and systems engineers continues to be a challenge for the industry. This is the well-known "software crisis".

The evidence that our approach works in a large part is anecdotal. We find problems. We fix problems. We fly systems. The software and associated systems work. Does this say we have an optimal solution to building systems? Perhaps more importantly, what is the optimal solution? Does our approach work in the more academic theoretical world? Are there better "mouse traps" that will work for us, our process, and engineering staff? We have metrics now, but will they allow us to answer these questions and know if we actually are getting better? There still seems to be a lot of questions unanswered.

References

B. Beiser, "Software System Testing and Quality Assurance", Van Nostrand Reinhold, 1984

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**.

J. Knight and N. Leveson, "An experimental evaluation of the assumption of independence in Multiversion programming", *IEEE trans. of Software*, SE-12, Jan. 1986.

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