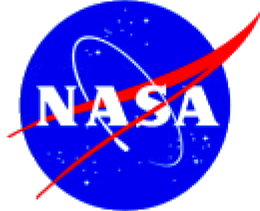
Alan Ogletree

Alan Ogletree is a lead software tester at United Space Alliance, LLC, in Houston, Texas. USA is the prime contractor for NASA's Space Shuttle program and has overall responsibility for the Shuttle's Primary Avionics Software System (PASS). Alan has spent the past 18 years working on the PASS, initially with IBM, and then with Loral and Lockheed Martin before joining the USA team in 1998. Alan holds a BS in Computer Science from LSU. He lives in Friendswood with his wife and three boys, and particularly enjoys time with his family, hiking, reading and being his kid’s principal.

Testing Large Mission-Critical Software Changes



Alan Ogletree
United Space Alliance
Space Shuttle Programs
PASS FSW

STAR*East* 2002

I. Objective

This paper is based on a recent experience implementing and testing a large new software capability in a maintenance organization which had not dealt with a large change in some time. The capability was called GPC Payload Command Filter (GPCF). While the task was completed successfully, it was not without cost in terms of schedule slips and personal angst.

The purpose of this paper will be to help the verifier learn from what was done right and what was done wrong, hopefully to avoid the pitfalls and emulate the successes. Specifically, the objective is as follows:

> *To provide guidance on how to successfully test a large new software capability using verification processes which have specialized over time to provide extremely effective results for relatively small changes.*

II. Space Shuttle Program Background

Shuttle Primary Avionics Software System (PASS)

The organization in which this large new development activity occurred is the United Space Alliance, LLC, (USA) Space Shuttle Primary Avionics Software System (PASS) Development organization. USA is NASA's prime contractor for Shuttle operations, performing the day-to-day management of NASA's fleet of Shuttles.

The PASS is a large, complex group of programs that run in a set of identical computers (General Purpose Computers, or GPCs). During ascent, entry, and critical on-orbit phases, the same software runs in up to four GPCs simultaneously. This is done to provide fail-safe, fail-operational protection against hardware failure. During on-orbit and less critical phases of operation, different PASS programs may be loaded into single computers within the set.

PASS has primary responsibility for critical flight operations. During flight, there are two major flight software programs that execute:

- *Guidance, Navigation, and Control (GN&C),* which runs in multiple redundant computers to perform the functions of navigation ("where are we"), guidance ("where do we need to go"), and flight control ("what do we need to do to get there").

- *Systems Management (SM)*, which runs in a separate computer to perform non-vehicle control functions – payload interface and management, communications, robot arm control, payload bay door opening/closing, parameter acquisition, fault detection and annunciation, and other vehicle system and payload monitoring functions.

The PASS changes that form the basis of this paper were made in SM.

PASS is of very high quality. To satisfy NASA's requirement for human-rated software that meets the highest safety and reliability standards, PASS evolved a software process that yields a highly predictable quality result. By executing the process faithfully to specified process standards, software produced by the process is predictably near zero defects. The dedication to producing quality products has resulted in its becoming a recognized leader in software quality. The product quality is at a level within industry known as world-class. For the past five years, the number of product defects per thousand lines of modified source code has averaged less than 0.1.

The PASS project was the first project assessed at a Maturity Level 5, the highest rating, of the Software Engineering Institute's Capability Maturity Model for Software®. The project received its ISO 9001 registration in 1994.

NASA's Space Shuttle Program

The Space Shuttle program is relatively old in that it has been around for nearly 30 years. The first Shuttle launch was in April of 1981. As of March, 2002, there have been 108 flights.

The Shuttle program has been remarkably successful. It has provided the nation with reliable human access to space for a variety of purposes. Through its 20 years of flight, it has been versatile enough to adapt to a constantly changing mission – including commercial satellite deploy, classified DoD projects, large-scale space-based scientific research, launch of interplanetary satellites, space station construction, satellite retrieval and repair, crew and logistics ferrying, and it has even been an instrument of international policy. The programs longevity and adaptability have validated the overall system design.

USA's PASS has supported all flights since the beginning of the Shuttle program, and there have been no critical PASS errors discovered in flight. Of the very few non-critical errors discovered in flight, none have resulted in the loss of a single mission objective or the shortening of a mission.

Nominal PASS Process

Since the early 1980's, PASS has been developed in software releases called Operational Increments (OIs). The OI includes a collection of software changes bundled together to support specific longer term Shuttle mission objectives, and comprises four serial life cycle phases:

- *Requirement development and approval* is typically a 9 month phase where an engineer comes up with an idea, submits a proposal to NASA, gets the concept approved, develops and modifies a set of requirements pages, gets community concurrence, and finally receives NASA approval to implement the change.

- *Software development* is typically a 10 month process where the requirements are implemented through a phased process involving functional design, detailed design, code, and internal development test.

- *Software verification* is typically an 8 month step where detailed and functional and performance tests are executed in order to fully test the software.

- *Reconfiguration* is typically a 9 month step in which the delivered software is configured for use in specific Shuttle missions flown using this release.

Because of the length of this entire process (36 months), several OIs are always in process concurrently (Figure 1), with a new PASS delivery approximately every 12 months.

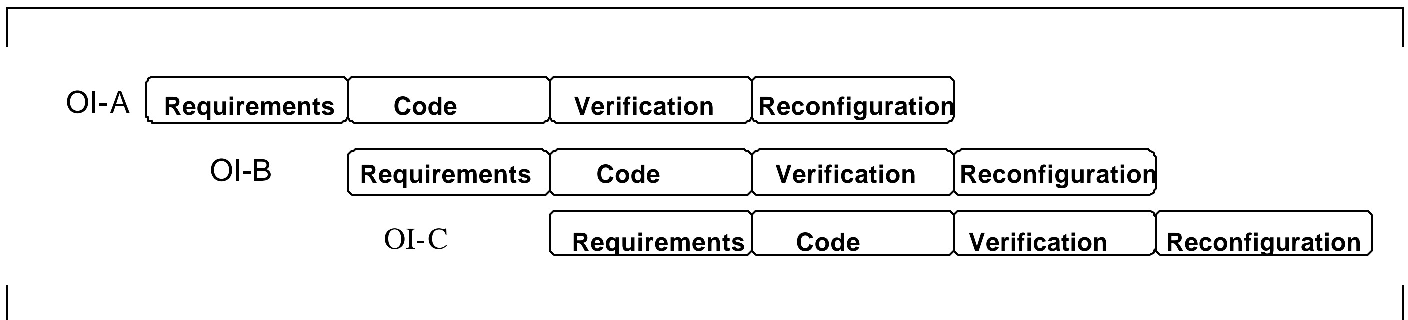


Figure 1. Concurrent Operational Increment Development

PASS Testing Process

The comprehensive PASS testing process has a long pedigree, is well established, and has proven to be extremely effective. It consists of multiple independent levels of test crossing several life cycle phases (Figure 2). (Besides these, there are additional levels of test performed once the PASS is delivered, but those are outside the scope of this paper.) The PASS testing process is tuned primarily for quality, however, and not for cost or schedule. This fact must be taken into account when applying what is presented here.

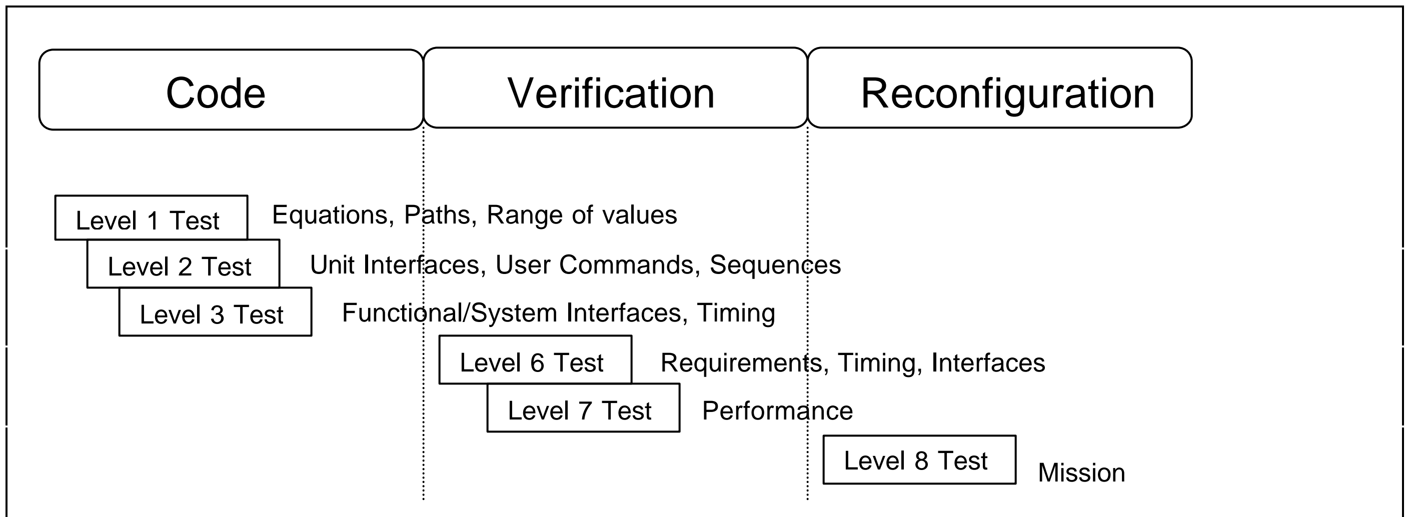


Figure 2. PASS Testing Levels

The testing phase specifically addressed in this paper is Level 6, or functional, detailed verification. For understanding the subsequent recommendations, it is important to understand that the Shuttle Level 6 verification testing is "white box". Since verifiers are called upon to ensure that every statement of code – even every segment of code within each statement – has been executed and verified, verifiers must develop an intimate understanding and knowledge of the code. Because of this, all verifiers participate extensively in all life cycle phases.

The details of the Level 6 testing process are much more complex than can be adequately covered here. Figure 3 presents a high level overview of the general tenets of the PASS Level 6 testing philosophy.

Detailed Verification Philosophy	**Brief Summary**
Assume Software Is Untested	Test analysts are to assume that any changed software delivered for verification is totally, completely, wholly and altogether untested.
Perform Independent Testing	Execution of the verification testing process is to be done independently of any and all other testing processes.
All Requirements Explicitly Tested	All defined requirements are to be explicitly verified.
Develop VTPs From Requirements	Verification Test Procedures (VTPs) are to be initially designed based on the flight software requirements to be tested.
All Changes Indexed to Requirements	All flight software changes will be identified as to the verification activity used to verify the change (e.g., inspection, test, static analysis, etc.).
All Code Explicitly Tested	Every line of code (or, more specifically, every segment of code) that has been changed must be explicitly executed.

Test Changed Functionality Only	Because all software has been previously tested, and because other levels of testing include regression testing of unchanged functions, analysts can assume that unchanged software functionality does not need to be retested.
Test Using Unmodified Software	All testing will be performed using actual flight computers with an unaltered flight software load.
All Changes Require Formal Closure	All changes to the software require formal closure by verification. (This means formal peer reviews, customer test reviews, and signed closure forms.) This closure should be done by independent test, analysis, and evaluation of the flight software.
Configuration Control	All Verification Test Cases (VTCs) are to be maintained under configuration control.
Use Resources Efficiently	Test plans and test cases are to be developed and executed in a manner which makes efficient use of computer and analyst resources.

Figure 3. Level 6 Testing Principles

Maintenance Phase

PASS was initially developed in the late 70's and early 80's and, as would be expected, is currently in a maintenance phase. The PASS continues to change frequently, but when it does, it is usually in small, self-contained pieces.

In this maintenance phase, PASS changes typically occur for one or more of the following reasons:

- *Refinements of existing systems*: As flight experience is gained and engineers become smarter with their systems, they often come up with improvements. For example, the robot arm control algorithms previously did not correct for arm hardware inaccuracies. So if the arm was commanded along a path, it would often diverge slightly, requiring the crew to manually correct the arm position. To fix this, the software was updated to include an auto-correction capability which kept the arm traveling along the commanded path.

- *Increased system reliability*: The Shuttle program is always looking for ways to make the overall system safer. A classic concern is abort survivability. Certain aborts are survivable only if the crew does exactly the right thing at exactly the right time. By automating as many of these procedures as possible, crew recognition and execution times are reduced, increasing the abort regions (length of time, number of scenarios, etc.) in which the crew and vehicle can survive.

- *Hardware upgrades*: With the age of the project, many hardware systems are becoming obsolete and costly (or impossible) to maintain. Additionally, many systems need to be updated to provide the benefits of modern technology, or to adapt to the changing role and mission of the Shuttle. An example of this is a recent upgrade in which the displays were updated from green monochrome displays to color displays, and many of the flight instruments were upgraded from mechanical devices to electronic displays.

A typical maintenance cycle (OI delivery) involves approximately 2-3% of the existing total lines of PASS code. This defines the "comfort zone" in which the project currently operates. The addition of major, large new capabilities in the software occurs relatively infrequently.

III. GPCF Project Background

The Exception

Despite the fact that major new capabilities are rare – especially in the Systems Management (SM) functional area – this occurred recently in the PASS program. The capability to be added was called GPC Payload Control Filter (GPCF).

GPCF is part of a larger cost savings project that is designed to streamline payload development by allowing the Shuttle payload customer to develop his own software on a Cargo PC[sm] laptop. Because the Cargo PC has a lower reliability requirement than the GPC, the Cargo PC must communicate with the payload through the GPCF software when it needs to perform hazardous operations. The GPCF software authenticates and filters hazardous payload commands before forwarding them to the payload. This provides the requisite safety features while allowing greater flexibility in the payload software development process. The following diagram presents a high level overview of the GPCF system architecture.

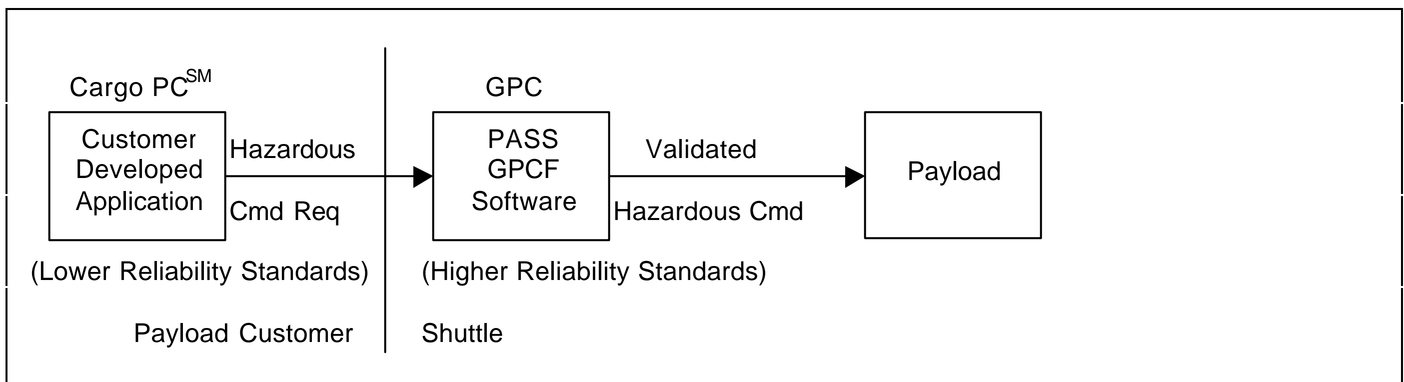


Figure 4. GPCF System Architecture

(Note that the PASS GPCF software portion is the subject of this paper.)

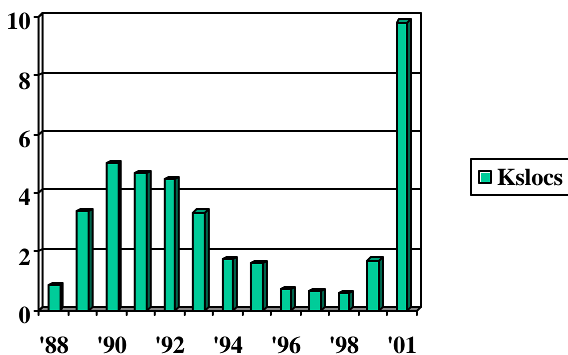


Figure 5. SM Code Modification History

This was the largest SM code change in more than a decade. It was approved in November of 1999. Figure 5 shows that the GPCF code development ('01) was at least twice the size of any SM code change implemented in the past 13 years, and 500% larger than any set of maintenance updates performed in the past 7 years.

The project committed to delivery of this system in February of 2001, fully tested and error free.

The Result

Due to a number of factors, the project was unable to keep the original schedule commitment. This ultimately resulted in a two month change to the final software delivery date. (This schedule was negotiated with affected stakeholders some months in advance of the planned release date in order to assess the impacts, if any, to the Shuttle program.) Additionally, the code that was delivered to the verification testing organization had a larger than anticipated number of errors present. (Fortunately, a large number of these errors were detected in time to fix and re-verify them before final software release.)

Based on some serious soul searching which occurred after the belated delivery of the software, a number of suggestions were brought forth as to how the project could have better prepared for and managed this large new software task. This "soul searching" took the form of a comprehensive project-wide team which examined all contributing factors that fed into the GPCF schedule problems and instituted a series of process improvements and risk mitigation plans which have been reported to NASA and are currently being tracked. From this information, among other sources, six thoughts or lessons learned have been distilled which should help software project planners make the mountains flat, the paths straight and prepare the way for a successful large new software change.

IV. Lessons Learned

(1) Influence the Requirements

It may seem out of place for an item related to requirements to be mentioned first in a paper on testing. However, good requirements are foundational to any project. In PASS, verifiers are involved in all software changes from the beginning. This has the benefit of helping the verifier understand what he will be testing and forcing him to begin early to formulate a plan for how he will test the forthcoming changes. Additionally, the project experience the verifier has, even if his history is only in verification, is valuable, and should be applied to all life cycle phases. Here are two specific ways in which, in retrospect, it would have been beneficial to influence the requirements.

Cut functionality. If it becomes evident that verification (or the project as a whole) is being squeezed (schedule, budget, etc.), the proper route is to cut functionality, not quality. The obvious time to recommend a reduction in functionality is while the requirements are being written, but before they have been approved. Take the hard line early. Remember that although writing flashy requirements is fun, eventually someone has to build and test them. Go through a process of categorizing requirements in terms of minimum requirement vs. highly desirable requirements vs. "nice to have" requirements. And if a cut must be recommended, thought will have already been given to identify where those cuts need to take place.

For example, the approved GPCF requirements allowed for the possibility of 31 Cargo PCs^{sm} (15 running concurrently) on 2 ports executing up to 15 payload applications simultaneously. Realistically, for a new Shuttle application, a first use of the system will probably use 1 Cargo PC on 1 port running a minimal number of payload applications. In retrospect, it would have been more sensible to start with this minimum requirement and build in capability as needed. Instead, the requirements specified a feature-rich system and the project ended up with some extremely complicated code (which is unlikely to be used anytime soon.) Rather than seeking a phased implementation, this change called for an "all or nothing" approach.

One word of warning: it is sometimes difficult to tell from examining the requirements what will be difficult and what will be manageable. In GPCF, when it became apparent that the requirements were

getting large, a list was made of functionality that it was thought might ought to be deleted. (The case was not presented forcefully, however, and little functionality was deleted.) But in retrospect, only about half the capabilities thought to have been difficult turned out to actually be difficult, and many of the things not considered for deletion (because they "seemed" simple) ended up being the actual bugaboos that toasted our buns.

If project team members feel strongly that they are being over-committed and that reasonable cuts can be made to the requirements, make a case and propose them. The worst the requirements authors can do is say no.

Contain requirements creep. When a set of requirements are first submitted, they are usually tightly connected and coherent. However, as reviewers suggest changes, and these changes are incorporated into the requirements, it can:

- Result in patchwork, inconsistent requirements,
- Have unintended side effects, and
- Result in a system that is much harder to test.

Each of these problems was encountered during GPCF. As the requirements were modified over a four-month period before they were approved, many requirements crept in that ultimately made the testing job much more difficult. Of course it is acceptable to fix errors in the requirements, but the "hey, can't we do this" and "why don't we add this" additions, especially if added in a piecemeal fashion, can lead to problematic requirements.

For example, GPCF originally had a capability that was common across 2 displays. However, as requirements were inserted, these capabilities diverged – not intentionally – resulting in the displays working differently. This meant that a common test scheme for both displays had to be split into a unique test scheme for each display.

Another example involved status data that was received from the Cargo PC^{sm}. Initially the status data was intended to only be displayed, but as the requirements were modified, this status data ended up being modified by the software in certain cases. Then, as coding began, more scenarios were uncovered that had not been considered, resulting in a final product (which matched requirements!) that was inconsistent and, occasionally, incoherent.

Remember, requirements creeps are bad people!

(2) Influence the Code

Again, the tester has an important role to play in influencing the code. In a "white box" testing environment, the testability of the code is an important issue. However, the verifiers failed in several instances to recognize and act on issues that could have made the testing experience more agreeable.

Two examples of code issues that were not satisfactorily addressed by verifiers during the coding phase were:

- *Independently coded "duplicate" functions* – There were a number of cases where similar functions were coded in substantially different ways by different coders. For example, there are two data tables in GPCF that have the same types of operations performed on them – additions,

deletions, modifications, etc. However, these operations are coded in significantly different ways for each table. This complicates testing in that, not only do the verifiers have to understand these two different methods, but they have to gather different sets of data to test each table. And instead of using a single test philosophy for both tables, they had to develop a separate testing philosophy for each table.

- *Unnecessary Complexity* – The Shuttle software is just plain complex, and there's not much that can be done about it. However, in the case of GPCF, a design architecture was chosen which resulted in an unnecessarily complex software module. By one measurement technique (McCabe's) this module had a cyclomatic complexity of 3 times greater than the most complex existing PASS software module. Although in this particular instance the verifiers concurred with the design, it is generally considered unwise to attempt to correctly code and test a module of high complexity.

So, while a verifier performs only an advisory role in code design, it is important for him to raise issues related to testability. While there may be legitimate resistance to many of these suggestions – they affect the schedule, they cause additional work for the developers, they complicate the code, they add inefficiencies to the code, etc. – it is still important to make these suggestions. And if a tester can make a strong enough case, it might get done.

(3) Consult Established Expertise

Even though the SM area had not had a large change in 13 years, other sister areas in the PASS organization had. Specifically, the GN&C area had recently implemented a large change associated with GPS navigation. The SM area failed to take full advantage of that expertise – learning from GN&C what they did right and (more importantly) what they did wrong.

In any maintenance project there is an accumulated wisdom that may not be written down anywhere, but exists solely in the minds (and filing cabinets) of those who fought the early wars. The problem is that these people have often migrated to management and staff positions that, unfortunately, don't always delve into the technical, detailed, daily workings of the project. As such, they might not become aware of a misdirected path taken until too late. So this expert involvement needs to be sought out early

Following are two examples of issues about which the verifiers should have consulted these experts earlier in the project:

- *Architecture* – As mentioned earlier, the high level design that was selected ended up being quite difficult to implement and test. This design resulted in a very large main program which, when compiled, was too big to fit within a storage allocation block on the Shuttle mass memory device. This resulted in a time-consuming effort to split this module into smaller pieces.

  One of the comments from the oversight findings on the GPCF development experience was this:

  > *There was not a process requirement for a broader project review of the architectural design choices made for a large, new function.*

  A review of the high level design early on by the project experts could have averted some mistakes that proved costly down the road.

- *Costing* – Because of the size and risk associated with the project, project expertise was involved early on and consensus was achieved on the proposed cost and schedule. However, as the project progressed and cost was refined upwards, the project failed to keep the original experts adequately informed. As such, project management only became aware of a growing cost-schedule disconnect after when some corrective activities could have been initiated.

(4) Don't Procrastinate

By the time this project got to the verification phase – some seven months after the first line of code was committed to paper – everybody was concerned. It was no surprise to the verification team members or to the verification management that this project was much bigger than originally thought. There was concern that since the development was exceeding cost, the same might happen to the verification effort. As such, much planning and pre-verification work was done.

Verifiers have the advantage of having the opportunity to accumulate quite a bit of experience about what's happening before the code is released for verification. If they've been proactive, they have participated in the requirements process (and understand what the code is supposed to do). They have participated in the code process as a reviewer, as well, so they should be intimately familiar with the code. So, when it comes to verification time, the verifier should know exactly what's out there, what has to be done, and how to do it.

So take advantage of the position that verification has in the flow – the last step – to get a head start on verification. While others are doing code, the verifier should be preparing for test. When the verifiers see panic setting in among their development brethren, that's a signal to start preparing now! Every tester has other tasks besides testing. Get management's permission and defer or offload some of those tasks, so that the focus can be on the upcoming verification effort.

Our experience has shown that the most effective way to get ahead of the curve is to invest effort developing time-saving tools. Beginning as early as the requirements generation phase, the verifiers began developing custom tools in 3 basic areas. These classes of tools should apply to any testing project, not just ours.

- *Test data generation*: If it's difficult to generate test data, then it becomes much more difficult to create the bizarre and weird scenarios that are the bread and butter of testing. The verifiers created a tool that allowed test data to be specified in an easy, sensible format. This saved countless hours as the verifiers refined and updated their test data during verification.

- *Test script generation*: The mechanics of writing a test script can be quite time consuming, depending on the simulation environment that is supported. For GPCF, the verifiers recognized early that changes to the simulation environment would be needed to facilitate effective testing, so they pushed for these early. Additionally, tools were generated with converted the test data into the multiple formats that were required for testing.

- *Data reduction*: How does a verifier know if the test worked or not (or even if he provided the correct inputs) unless he looks at data? Data, data, data! It can be overwhelming. Tools to reduce data from columns and columns of numbers into readable formats, with changes or other significant features highlighted, can make a vast difference. Tools were developed which not only reduced the data to easily readable formats, but also automated some of the rote analysis of data that was be required. Not only did this save time, but it allows the verifiers to uncover errors in

the software that would otherwise have been almost impossible to detect by a someone using conventional manual analysis techniques.

It is likely that one of the reasons so many errors made it past the internal development testing phase and into the formal verification phase was the lack of tools on the development testing side. Coding input commands in hexadecimal, having a very limited set of test data, and having to look at pages and pages of raw test output makes it very difficult to find errors. If the verification testing team had been in the same boat, it's almost certain that either the verification effort would have either run past schedule or have been much less effective than it was.

In addition to the use of testing tools, verification activities were sequenced during the 6-month testing phase in order to try and get the most benefit out of the early tests, specifically in terms of trying to identify any "show stoppers" or other major problems as early as possible. By performing these key verification activities early, the verifiers were able to quickly establish confidence in the basic soundness of the system and commit to an early field release of the system for lab testing with an actual Cargo PC^{sm}.

(5) Realize Everything Can't Go Right…

Despite the best-laid plans, things are not always going to go right. In the case of GPCF, quite a few bold assumptions were made up front, and the success of the schedule depended on them being right. In retrospect, the project was perhaps a bit too success-oriented (i.e., optimistic) in its planning.

Some of the assumptions made, which didn't pan out, were:

- *All new hires would be trained quickly*. In a large project such as GPCF, current staffing levels proved inadequate and new people had to be hired. Unfortunately, many were hired too late to provide sufficient training. (In a complex project like PASS, it usually takes several years before someone is truly productive.) Because of this, the productivity effect on the team from these new hires was not as great as had been hoped. For a superior process to produce superior results, quality, adequately trained personnel are essential.

- *People from other areas would be available and would be fully functional*. GPCF was able to borrow some experienced folk from other sister organizations. While generally successful, in some cases the specific people requested (who had the exact skills requested) were not available. Also, while the people who were able to be used were experienced in PASS, they were not specifically familiar with the SM software, and so required some level of oversight. Although people from other areas were indispensable to the project, they were not as productive as had been hoped.

- *Changes would not expose system limitations*. Since GPCF was simply considered a larger than usual maintenance task, it was expected that the existing architecture and support environment would support the change. Therefore, no prototyping activity was performed. Unfortunately, the size of the change exposed limitations on a number of the project tools – compiler, display code generator, linkage editor, and the code path analysis test tool – that had to be worked around.

- *Everything would work out.* Each maintenance change needs to be evaluated realistically, especially if it is different (i.e., larger, more complex, shorter template, etc.) than the normal maintenance change. Complexity does not increase linearly with project size. A history of success and high quality can lead to an over-confident "been there, done that" attitude, but it

should be emphasized that the history of success does not guarantee a similar future. When this attitude occurs, it makes it very difficult to cause an organization which is accustomed to things going right to recognize that things are going wrong.

Inadequate margin was built into the schedule to recover in case serious problems were encountered. This was recognized from the beginning, but the project believed the risk was acceptable.

(6) …But Sticking To The Proven Process Does Pay Off

As much as this paper tends focus on what went wrong, there were significant successes as a result of this effort.

The first flight from this new PASS system to use GPCF has yet to fly. Since this software will continue to be used in progressively more complex missions for years to come, the ultimate quality of this software is yet to be proven. However, internal statistics and limited use in NASA test labs indicate that the quality of the delivered product seems to be in line with the quality of the existing software.

One temptation which was never seriously considered was the temptation to cheat the detailed steps of the testing process. Although the software was delivered late for testing, the project determined to maintain the time required to adequately complete the verification testing, even at the expense of the overall project schedule. Many verification organizations get squeezed out of their required verification schedule time when code is delivered late, but in taking the political "hit" to move the schedule – and guarantee the time required to ensure a properly verified product – a project will ultimately benefit by having a higher quality product. As mentioned earlier, this particular PASS verification process has been in place for decades and has a proven record of success. Where the verifiers did make reasoned exceptions to the process, they were documented (as waivers) and project management concurrence was obtained.

Through each step of the development, although the error counts were high, the proper processes and procedures were followed. Additionally, statistical analysis was used to project the number of errors that would be found during testing, and actual numbers came remarkably close to the projections (Figure 6).

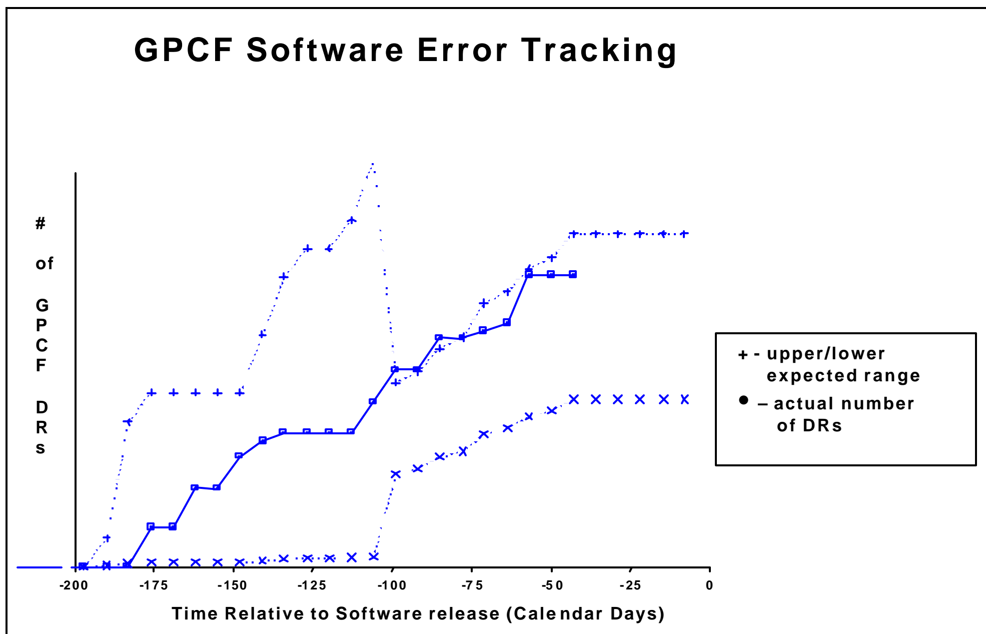


Figure 6. GPCF Software Error Tracking

So, by following the details of the established process, even though cost and schedule estimates were constantly under pressure, the verifiers were able to take a large, unfamiliar change and successfully verify it from within a maintenance-oriented process.

V. Conclusion

From a project perspective, the entire GPCF development experience was a significant planning and technical deviation. The life cycle processes, which have been tuned to produce reliable, on time code in relatively small packets, was severely overloaded. Although verification of the GPCF code was completed with initially encouraging results, only time will tell if that process too was overloaded. This is a concern for the PASS project management and additional risk mitigation efforts are being conducted to go above and beyond the usual product quality assurance activities.

Despite this, the apparent success of the verification effort can be attributed to several items:

- The verification process had access to established expertise and effective resources (both in place and borrowed).
- The verification team planned early for size and complexity outside of the familiar range of experience.
- The verification effort was front loaded to get most benefit from early test activities.
- The verification team had the advantage of witnessing GPCF project issues as they were happening.
- The project anticipated the likely effect of GPCF issues with additional risk mitigation steps. The earlier a project can adjust and replan, the better its chances of recovery.
- The project wisely preserved the required testing schedule, with original buffer, despite pressure to “make up the difference” .

So the thrust of this paper is to advise the verifier to use caution when taking an existing small-change maintenance process and using it on a large change. Take into consideration the six “lessons learned” presented in this paper:

- Influence the requirements
- Influence the code
- Consult established expertise
- Don’t procrastinate
- Realize everything can’t go right…
- …But sticking to the proven process will pay off

And, above all, avoid complacency and over-confidence.

VI. Additional Information

This paper is based, in large part, on information gathered and synthesized by John Magley in the project team oversight findings.

The USA home page can be found at http://www.unitedspacealliance.com.

The USA logo is a Registered Trademark of USA, LLC.

Cargo PC[sm] is a Service Mark of USA, LLC.

Capability Maturity Model for Software® is a Registered Trademark of the Software Engineering Institute at Carnegie Mellon University.

The following two papers are excellent sources of additional information on this subject:

- C. Billings, J. Clifton, B. Kolkhorst, E. Lee, and W.B. Wingert, "Journey to a Mature Software Process," IBM Systems Journal, Vol. 33, No. 1, 1994, pp. 46-61

- Carnegie Mellon University, Software Engineering Institute (Principal contributors and Editors: Mark C. Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis), The Capability Maturity Model: Guidelines for Improving the Software Process, ISBN 0-201-54664-7, Addison-Wesley Publishing Company, Reading, MA, 1995, pp. 93-121