The Evolution of a Test Process for Spacecraft Software

Deborah A. Clancy
Brenda A. Clyde
M. Annette Mirantes
The Johns Hopkins University Applied Physics Laboratory
11100 Johns Hopkins Road, Laurel, MD USA 20723-6099
Brenda.Clyde@jhuapl.edu
Annette.Mirantes@jhuapl.edu
Deborah.Clancy@jhuapl.edu

Abstract

The Johns Hopkins University Applied Physics Laboratory formed an embedded software group in 2001. In addition to defining the process for developing and testing flight software, the group had to quickly apply those new processes to a series of four different spacecraft missions from 2001 through the present. The paper describes the evolution of the software testing approach for embedded flight software during this time. After a brief description of the four spacecraft missions, this paper presents our initial approach to testing and how that approach changed with each mission, as our resources were overextended and our schedules were compressed. The final section of the paper presents the changes that we believe had the most significant impact on our testing efforts and our proposed test approach for the next mission.

1. Introduction

Like most starts at establishing an engineering methodology, our effort to establish an effective and efficient methodology for testing spacecraft software began as a set of guidelines and is continuing to evolve into a defined and effective process. After the success of several missions for NASA and the Department of Defense, in 2001 we determined that the Johns Hopkins University Applied Physics Laboratory (JHU/APL) Space Department software group was growing large enough that reorganization was necessary. As a result, the group was divided into three separate entities, each with its particular focus. The charter for one of these groups was the development of embedded software for unmanned spacecraft. The software development process at that time was a set of guidelines—recommendations with broad statements about how to develop and test flight software. Over the course of five years and four full spacecraft development efforts (some complete, some still “in the works”), those guidelines are becoming a defined, useable, and valuable process.

This paper focuses on our experience with the process for testing flight software. We currently have a process for testing flight software, but it is less cost-effective and efficient than we would like. To improve the effectiveness and efficiency, we did an evaluation using process metrics. We identified problems and areas where process improvements can enhance the efficiency and cost-effectiveness of flight software testing at JHU/APL. We studied the software testing process used on four recent and current missions and have used the results to recommend process improvements.

2. Mission overviews

Our test process was used during the flight software testing for four spacecraft missions. The missions are discussed briefly in the following sections.

2.1. CONTOUR

The COmet Nucleus TOUR (CONTOUR) was the first mission to be tested under our new process. CONTOUR launched July 3, 2002. The flight software architecture consisted of three Computer Software Components (CSCs): Command and Data Handling (CDH), Guidance and Control (GC) and Boot. These three CSCs were also the focus of the testing effort.

The CONTOUR architecture is one common to many spacecraft: one main and one backup processor for CDH, one main and one backup processor for GC, and shared Boot code. For all four missions, a 1553 bus centralized the interfaces to external subsystems, but custom interfaces were also used.

2.2. MESSENGER

The MErcury Surface, Space ENvironment, GEochemistry and Ranging (MESSENGER) mission began its testing effort mid-way through CONTOUR's testing. MESSENGER was launched August 3, 2004. The flight software architecture for MESSENGER diverged significantly from that of CONTOUR. MESSENGER had the CDH, GC, and Boot CSCs, but added a fourth CSC, Fault Protection (FP). The hardware architecture included a new processor and a new operating system. Unlike CONTOUR, the CDH and GC CSCs were located on one processor, and the FP software ran on two independent processors, one serving as backup. MESSENGER also included many new technology initiatives. Some were specific to software; others were both hardware and software. Some of these new features included a DOS-like file system for the solid-state recorder (SSR), the use of Consultative Committee for Space Data Standards (CCSDS) File Delivery Protocol (CFDP), a file-based communications protocol, an integer wavelet image compression algorithm, and a phased-array antenna control algorithm.

2.3. STEREO

Shortly after the test effort began on MESSENGER, the Solar-TErestrial RElations Observatory (STEREO) began its testing effort. STEREO is scheduled for launch in 2006. The STEREO flight software architecture was very similar to that of CONTOUR. It consisted of the CDH, GC, and Boot CSCs, but also included an Earth Acquisition (EA) CSC. The STEREO hardware architecture, like that of CONTOUR, uses one processor for CDH, one for GC, and shared Boot code, but, unlike CONTOUR and MESSENGER, STEREO had no backup processors. The processors and operating system were the same as those used on MESSENGER.

2.4. New Horizons

The final mission, New Horizons, started its test effort midway through the MESSENGER and STEREO testing efforts. New Horizons launched in January 2006. The flight software architecture and most of the hardware architecture for New Horizons are identical to those used on CONTOUR, although the use of several new technologies on New Horizons increased its complexity. The new technologies used included a Flash memory SSR, a thermal control algorithm, and over 30 different science data types for recording, verifying, and downlinking.

3. Mission differences and complexities

As the mission descriptions above indicate, the four missions we studied were not sequential; there was quite a bit of overlap during software testing. The differences among missions and their complexities varied. Table 1 lists various contributing factors to software complexity. Increased software complexity is associated with increased effort and complexity of testing.

4. Mission test processes

Starting with CONTOUR, we identified the need for more formal verification and validation and established a formal process. However, this process was beyond what was baselined in the CONTOUR proposal. The CONTOUR proposal did not include plans and funding for any independent software testing to verify functional requirements. Each subsequent mission had similar issues with the requirement for increased testing; of the four missions, only New Horizons included a small effort for this formal verification and validation testing in its original plan for the mission.

Table 1. Factors contributing to software complexity

Mission	Number of Flight Software Requirements	Number of External Interfaces	Number of Science Instruments	Lines of Code	% Software Reuse
CONTOUR	690	12	4	37893	30%
MESSENGER	1035	19	7	143121	30%
STEREO	1422	15	4	126054	15%
New Horizons	1074	12	7	145618	35%

4.1. Initial test process

The initial JHU/APL test process was based on the NASA Software Engineering Lab guidelines as specified in the *Recommended Approach to Software Development* developed by the NASA Goddard Space Flight Center [1]. CONTOUR was the first JHU/APL mission to apply this methodology. We implemented the process using dedicated test resources during the code- and unit-test phase of the development lifecycle. The efforts centered on training staff, establishing the deliverables, and verifying the functional requirements of the software. These efforts provided a foundation for future missions. Shortcomings to this process were identified. Changes were applied to the MESSENGER test process, which started the evolution. STEREO and New Horizons continued the evolution.

4.2. Evolution of the process

Subsequent missions tailored the process to address weaknesses identified in other projects as well as to meet mission-specific requirements for testing (Figure 1).

For CONTOUR, the ad hoc planning didn't take into account past experience or metrics since there wasn't sufficient data available. The planning allocated the work to the time available and may not have been sufficiently resourced to be completed on time.

For MESSENGER the process was tailored to use the "Test Like You Fly" (TLYF) methods described in "Test Like You Fly" Confidence Building For Complex Systems [2]. These methods demonstrated that the software functioned as required under realistic operational conditions and when placed under unusual stresses. To ensure repeatability of tests, MESSENGER automated all procedures and produced formal test plans for each CSC and Build pair. These test plans were formally reviewed in their entirety. MESSENGER was staffed with dedicated resources specifically for testing.

For STEREO the process was tailored to use more of a mix of dedicated and part-time resources. The test plans were incrementally generated and reviewed. The goal of incremental generation and review was to reduce the cost of generating and maintaining test documentation. Test cases were prioritized to ensure that the more complex software functions were tested first. TLYF methods and Fault Protection testing were also applied upon completion of requirements verification. Most STEREO test cases are being automated, and STEREO is performing some

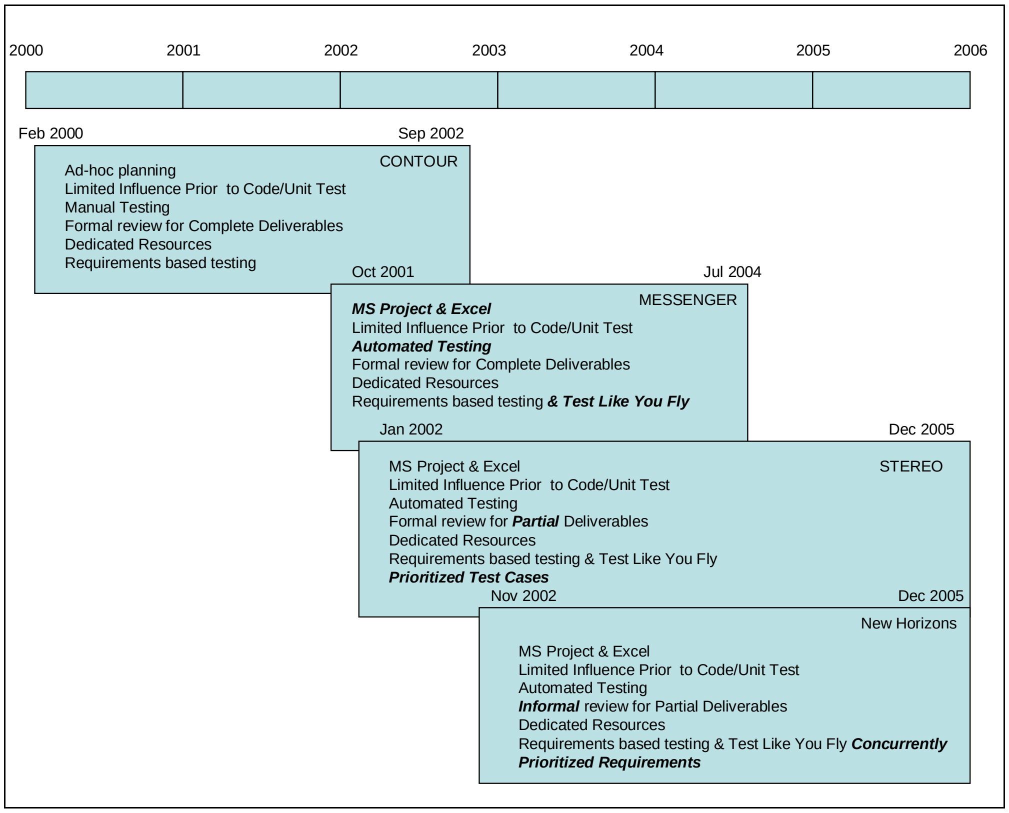
2000
2001
2002
2003
2004
2005
2006
Feb 2000
Sep 2002
CONTOUR
Ad-hoc planning
Limited Influence Prior to Code/Unit Test
Manual Testing
Formal review for Complete Deliverables
Dedicated Resources
Requirements based testing
Oct 2001
Jul 2004
MESSENGER
MS Project & Excel
Limited Influence Prior to Code/Unit Test
Automated Testing
Formal review for Complete Deliverables
Dedicated Resources
Requirements based testing & Test Like You Fly
Jan 2002
Dec 2005
STEREO
MS Project & Excel
Limited Influence Prior to Code/Unit Test
Automated Testing
Formal review for Partial Deliverables
Dedicated Resources
Requirements based testing & Test Like You Fly
Prioritized Test Cases
Nov 2002
Dec 2005
New Horizons
MS Project & Excel
Limited Influence Prior to Code/Unit Test
Automated Testing
Informal review for Partial Deliverables
Dedicated Resources
Requirements based testing & Test Like You Fly Concurrently
Prioritized Requirements

Figure 1. Evolution of the software testing process over the four missions

requirements verification through white-box or developer tests.

For New Horizons the tailoring process used more of a mix of dedicated and part-time resources. Most New Horizons test cases are automated. TLYF methods are being used concurrently with requirements verification. Requirements were prioritized to ensure that the highest-priority requirements were tested first. The test plans were incrementally generated and reviewed informally. The formality of the reviews was significantly reduced to reduce the cost of maintaining test documentation.

4.3. Current test process

The test process has evolved; it currently consists of 10 steps. Figure 2 shows the current process flow. The efforts now include steps to support increased planning and scheduling to better monitor the test progress, reduced requirements for documentation formality and maintenance, expanded use of TLYF methods, and risk-based prioritization for testing completion. The 10 steps are listed below:

STEP 1 – *Plan the Test Effort.*
STEP 2 – *Evaluate the Requirements.*
STEP 3 – *Create the Test Plan Outline.*
STEP 4 – *Define the Test Cases.*
STEP 5 – *Review Pieces of the Test Plan.*
STEP 6 – *Implement the Test Scripts*.
STEP 7 – *Execute the Test Cases.*
STEP 8 – *Create and Track Defects.*
STEP 9 - *Maintain and Report Test Status.*
STEP 10 – *Create Test Report Summary*.

5. Problems

After work began on MESSENGER, many staff both inside and outside of the embedded systems group believed that the testing process was not cost-effective or efficient. In July 2003, a working group was established to discuss test process improvement. The group was a mixture of managers, developers, and testers. The goals were to use our experience with CONTOUR and the ongoing MESSENGER and STEREO projects, and to help the current programs finish on schedule and plan strategy for New Horizons and future programs. The list of problems below was identified.

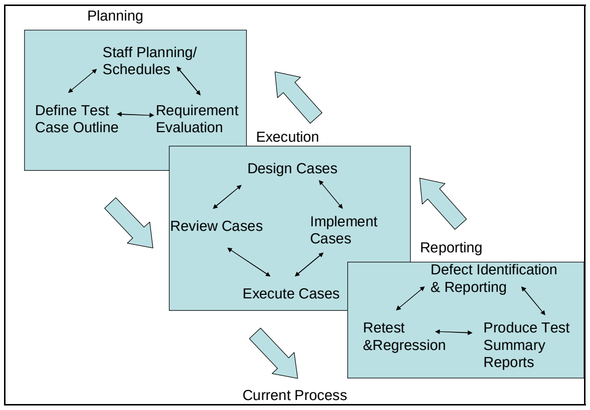

Figure 2. Current software testing process

Problem 1: Contents of flight software builds often change unexpectedly, which can affect the test team. Recommended Mitigation(s):

a) Keep test team informed of changes.
b) Assess impact of change to test team.
c) Focus testing on functional areas rather than builds.
d) Delay testing of functional areas that are not complete.

Problem 2: Requirements are often difficult to test using “black-box” methods and may contain too much design. Recommended Mitigation(s):

a) Consider alternative test methods (e.g., inspections, analysis, inference, and others).
b) Consider having development team verify these requirements during unit and integration testing.
c) Change requirements to remove design detail.
d) Assure that requirements review panel contains test representative.

Problem 3: Flight software and testbed documentation not available when needed. Recommended Mitigation(s):

a) Prioritize the needs of the test team.
b) Put milestones in schedule for required documentation.
c) Don’t deliver build without required documentation.

Problem 4: Late changes to requirements can affect test team. Recommended Mitigation(s):

a) Keep test team informed of changes (use Change Request system).
b) Assess impact of change to test team before approving it.

Problem 5: Lack of communication between development and test team. Recommended Mitigation(s):

a) Encourage communication (joint meetings).
b) Co-locate development and test team.
c) Assign single lead for both testing and flight software.

Problem 6: Little reuse of test plans or procedures between missions. Recommended Mitigation(s):

a) Assure test team has artifacts from previous missions.

b) Take steps to maintain mnemonic naming conventions between missions.

c) Assess impact to testing when making changes to heritage software.

Problem 7: Developing automated procedures is time consuming and requires documentation that might not be available.
Recommended Mitigation(s):

a) Reconsider the use of automated procedures.

b) Automate only when it makes the effort more efficient (e.g., regression tests that will be run many times or long-duration tests that can be run during off-hours).

c) Doing interactive testing may encourage more effort to "break the software."

Problem 8: Feedback from test plan reviewers and Independent Verification and Validation (IV&V) may lead to greater testing than we can afford. Recommended Mitigation(s):

a) Must assess impact to cost/schedule of this feedback.

Problem 9: New test team members must climb steep learning curve *before* they become productive. Recommended Mitigation(s):

a) Make greater user of experienced flight software developers to test.

b) Consider possible use of joint development/test team to foster greater communication and training of new team members.

c) Rely less on contract employees in the future.

Problem 10: Testbed user interface is too complicated and sensitive to change. Recommended Mitigation(s): None identified.

Problem 11: Lack of testbeds is an issue for the test team. Recommended Mitigation(s): None identified.

Problem 12: Testbeds do not contain the functionality that is needed. Recommended Mitigation(s):

a) Need to identify missing functionality and work with testbed team to assign priorities.

Problem 13: Testing common code across processors has not been straightforward. Common code may not have common requirements. Common code may be slightly different from processor to processor. Recommended Mitigations(s):

a) Flight software team needs to develop command and telemetry interfaces to common code.

b) Common code should be integrated by the same person to assure common approaches.

These recommendations were given to the test leads for integration into the test process for all of the ongoing missions. Some of these recommendations were implemented, but the expected improvement in cost and schedule didn't materialize. Another series of working group meetings was held in early 2004. This working group identified new problems, areas that were still problems, and made recommendations for mitigations. These are listed below.

Problem 1: It is very time-consuming to test all requirements equally by automated script. Recommended Mitigation(s):

a) Use automated test scripts for
- System setup scripts
- Common scripts that will be used by many testers
- Instances when an artifact is required
- To create regression and long-term tests
- To make use of testbed "off-hours" (test can run unattended)

b) Do not use automated test scripts for
- Tests that won't be part of regression or long-term tests
- Early builds when database and software is immature
- Negative testing that will not be repeated

c) Future mission should also use some type of prioritization by risk.

Problem 2 (old see Problem #2 above): Requirements contain too much design information, too much detail; requirements that are un-testable; entire areas of functionality with missing requirements. Recommended Mitigation(s):

a) Revisit requirements during the design phase.

b) Review requirements during code reviews and during test planning.

c) Provide training for writing better requirements.

d) Consider having development team verify these requirements during unit and integration testing.

Problem 3 (new): Requirements management tool being used is complex and costly for requirements maintenance. Recommended Mitigations(s):

a) Alternative methods of tracking discussed, but no change recommendation made.

Once again, however, the efforts didn't yield the expected improvements in cost effectiveness and efficiency. We found it very difficult to change an existing process for an ongoing program. The primary focus was on changes that could be implemented within the test process, since changing the development process requires buy-in from a larger team. Only very small adjustments to the test process could be made.

6. Evaluating the process

Thus, we learned that to succeed in improving the cost-effectiveness and efficiency of the test process for future missions, a more objective approach was needed. We therefore evaluated the test process by looking at software process metrics. We first looked at metrics for Defect Removal efficiency, but found that this metric could not be computed reliably because of weaknesses in gathering data on when defects were discovered and by whom. We next examined the percent of effort of the life cycle. The results of this metric for the four missions are shown in Table 2.

The baseline expects 25% of the flight software lifecycle to be devoted to system testing. This data shows that three of the four missions were within their allotted percentage of the lifecycle. However, that does not imply that they were within budget and schedule. In fact, the cost data in Table 3 tells a different story. Although actuals from previous missions are used to estimate future efforts, we continue to exceed planned costs.

6.1. Test team composition

Team composition is critical to performance. The cost data in Table 3 caused us to focus on staffing metrics. Test team composition is one area where improvements can be made to achieve cost-effectiveness. The staff experience chart in Figure 3 shows that the use of part-time and contract staff is not cost-effective for three main reasons: (1) Part-time and contract staff must tackle the steep learning curve associated with testing embedded software in a complex testbed environment. Once gained, this knowledge is not being retained for use on future missions. (2) Part-time staff requires more coordination, management, and testbed resources to ensure that they are spending adequate and effective time on the program. (3) Finally, part-time and contract staff are not always available when needed.

Table 2. Software process lifecycle partitions

	Planning Reqs	Prelim Design	Detailed Design	Code & Unit Test	System Test
Baseline	11.0%	14.0%	20.0%	31.0%	24.0%
CONTOUR	9.5%	6.6%	18.8%	43.0%	22.0%
MESSENGER	11.8%	17.0%	10.0%	41.2%	19.9%
STEREO	17.3%	6.4%	8.4%	43.2%	24.7%
New Horizons	7.7%	7.6%	13.1%	38.9%	32.7%

Table 3. Percentage relative to estimate of costs for software testing

Mission	Percentage Relative to Estimate: (Actual – Planned)/Planned $\times$ 100
CONTOUR	62%
MESSENGER	93%
STEREO	69%*
New Horizons	47%

*** Based on effort to date.**

7. Test Improvement Model assessment

To provide an independent assessment of our test practice, we used the Test Improvement Model (TIM) as outlined in *TIM – A Test Improvement Model* [3]. This model has two major components: a framework and an assessment procedure. The framework has five levels (0 being the initial non-compliant level) and five key areas. Detail information on this model can be found in reference 3. In general, this framework looks at five key areas – organization, planning and tracking, test cases, testware, and reviews. The assessment is done by assigning a value based on how well we implement the items identified for each level of each key area in the framework. The assessment allows the analyst to assign a current level in each area. The four levels for each area, valued from lowest to highest, are baselining (1), cost-effectiveness (2), risk-lowering (3), and optimizing (4). Each mission's state of practice was determined using the TIM assessment procedure. Figures 4-7 show the TIM assessment for each of the four missions. People involved with each mission's test function contributed to the results given in Figures 4-7.

7.1. Organization

All programs fulfilled the TIM criteria for baselining, although STEREO and New Horizons did lose a number of their core, experienced testers as

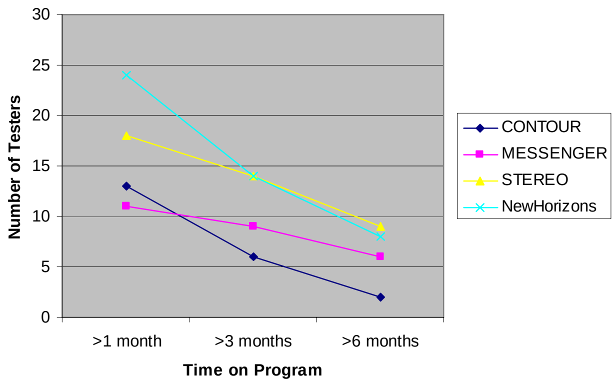

Figure 3 - Test staff experience.

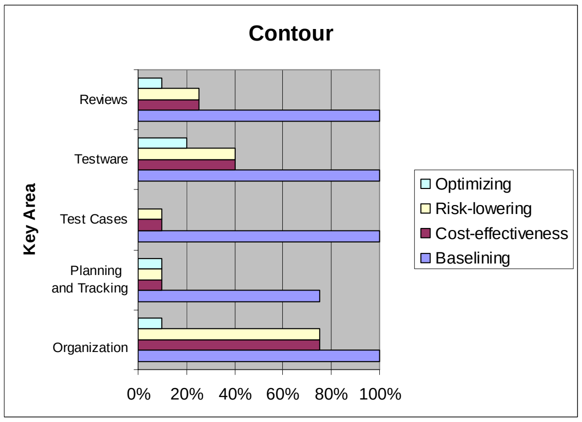

Figure 4. TIM ratings of the CONTOUR mission

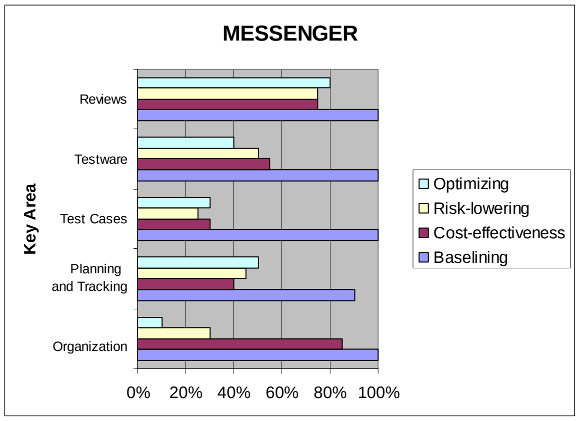

Figure 5. TIM ratings of the MESSENGER mission

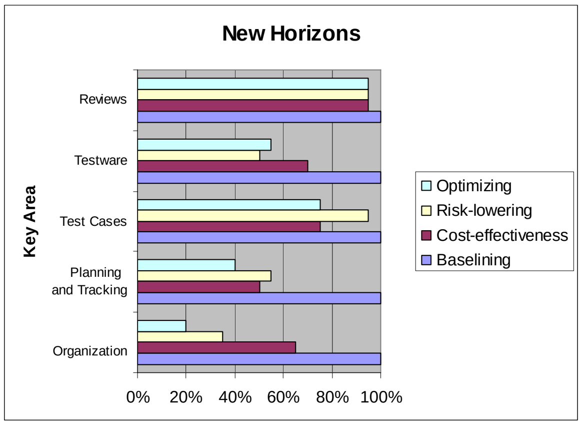

Figure 6. TIM ratings of the New Horizons mission

program priorities were realigned by management. To move the next program into the cost-effectiveness level and above, we need to focus on training and building a core group of full-time testers. Improved communication between the test and development teams needs to be developed.

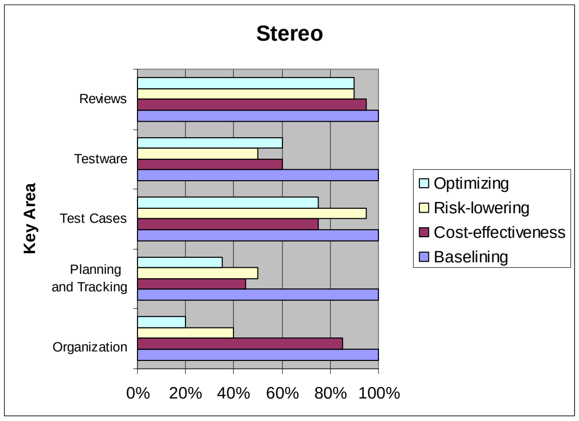

Figure 7. TIM ratings of the STEREO mission

7.2. Planning and tracking

The first two programs, CONTOUR and MESSENGER, lacked some documented standards, but the subsequent programs were able to fulfill the criteria for baselining in this area. To achieve cost-effectiveness, the future test programs need to have flexible planning and a standard method for tracking. The TIM authors propose that planning should be evolutionary. This would help some of the problem areas: changing requirements, changing functionality, and availability of hardware resources.

7.3. Test cases

All programs fulfilled the TIM criteria for baselining. To achieve cost-effectiveness, future programs need to develop the ability to allow testability to influence requirements and design. These programs will need to evaluate the testability of requirements and design and factor them into the overall software architecture and requirements definition. The test cases area did progress some into the risk-lowering level. Each program took steps to rank the criticality of either requirements or test cases or both. Significant progress has already been made toward achieving this level.

7.4. Testware

All programs fulfilled the requirements for baselining. However, to progress to the next levels, resources will need to be applied to evaluate and develop or purchase test tools to gain cost-effectiveness and to lower risk.

7.5. Reviews

This is an area where we are close to the optimizing level. Each program was successful in building on the previous work for each level. To complete the optimizing level for the future, investment needs to be made in training, and we should examine the possibility of adding different review techniques to our skill sets.

8. Ten lessons learned

The following are ten lessons learned from the working group results, actual experience, the review of the metrics, and the TIM assessment. They are in no particular order.

1. **Invest in testing –** Train a core group of testers, invest in tools, and integrate testing earlier into the development process (better communication).
2. **Increase focus on cost-effectiveness –** In the TIM assessment, all programs made some attempt to fulfill requirements in the risk-lowering areas, but not as much focus was placed on cost-effectiveness. Our business makes us risk adverse, and risk management is built into our processes. However, a better balance is needed between cost and risk in our testing effort.
3. **Use metrics for process improvement –**Metrics need to be used to improve the process for future missions. For example,
    a. Show cause and effect of late delivery of a functionality.
    b. Show overlap between deliveries and planned regression tests.
    c. Analyze the impact of late or incomplete software deliveries more effectively.
4. **Manage resources purposefully –** Commit resources to testing and stand behind the commitments.
5. **Scope the effort** – Have a plan that does not treat all requirements equally.
6. **Involve the development team –** Make verification a joint effort between the development and test teams.
7. **Leverage all test efforts –** Leverage the testing efforts of other teams to reduce duplication and increase effectiveness.
8. **Track changes to test documentation and test tools –** Enter Change Requests (CRs) for changes to test plans that result from reviews and from actual implementation of the tests. The documentation is changed to reflect the as-built software, but the changes aren't tracked.
9. **Plan for test case re-use –** Re-use case designs and scripts mission to mission and across CSCs.
10. **Plan for "waiting" time –** More effectively use time spent waiting for software deliveries and testbed resources.

9. Proposed test process improvements for future missions

1. Involve experienced testers heavily in requirements review.
2. Review requirements to assure testability and to filter out extraneous design information.
3. Perform risk analysis to determine whether functional areas to be tested via scenario testing or traditional requirements-based testing.
4. Build up scenario-based tests iteratively and incrementally.
5. Maintain all test plans in Requirements Tracking Tool, link them to requirements, and capture test methods.
6. Use scenarios as the basis of regression tests; use automation for cases to be included in regression; and do the remaining testing manually.
7. Build and review test cases and documentation incrementally. Track changes using a COTS version control system.
8. Gather and use metrics throughout the test cycles to allow dynamic process adjustments when needed.
9. Plan for re-use when developing test cases, and look for ways to simplify them.
10. Use verification matrix and test methods to identify where other test efforts can be leveraged.

10. Conclusion

Our flight software testing process was originally just a set of guidelines – recommendations with broad statements about how to develop and test flight software. Over the course of five years and four full spacecraft development efforts (some now complete, some in process), those guidelines have become a defined, useable, and valuable process. This process can be improved by continuing to identify weaknesses and implement changes to address the weaknesses. This paper highlighted some weaknesses and provided guidance on changes that can be made to improve the process. The process should be periodically reviewed and improved.

11. References

[1] Software Engineering Laboratory, *Recommended Approach to Software Development*, NASA Goddard Space Flight Center, June 1992

[2] M. N. Lovellette and Julia White, "'Test Like You Fly' Confidence Building For Complex Systems," *IEEE Aerospace Conference Proceedings*, 2005

[3] Thomas Ericson, Anders Subotic, and Stig Ursing, *TIM – A Test Improvement Model*, URL: http://www.lucas.lth.se/events/doc2003/0113A.pdf