

# Testing Flight Systems With Machine Executable Scripts<sup>12</sup>

Don Gibbs  
Jet Propulsion Laboratory  
4800 Oak Grove Drive  
Pasadena, CA 91109  
818-354-2990  
Donald.E.Gibbs@jpl.nasa.gov

Brian Bone  
Jet Propulsion Laboratory  
4800 Oak Grove Drive  
Pasadena, CA 91109  
818-393-7405  
Brian.D.Bone@jpl.nasa.gov

*Abstract*—The MSAP project at JPL has been testing spacecraft avionics and flight software since 2005, in part using computer executable scripts. The scripts are document files of a common word processor and comply with the format of a traditional, formal test procedure common at JPL. These procedures use keywords to issue commands and evaluate responses, mimicking a human test operator. In effect, script lines are inserted into a normal procedure. Even though the executable structure of the procedures is limited to linear sequences of fairly simple operations, we have found significant value in certain test regimes given the repeatability, ease of execution, and readily understandable intent of these procedures.

(GDS). The GSE provides hardware and software interfaces to the avionics and in some cases simulation of certain components; the GDS provides the higher level capabilities, such as command generation and telemetry presentation. The ground system effectively surrounds the integrated avionics/FSW, providing all avionics power, telecommunications (uplink and downlink), and interfaces to onboard sensors and devices controlled by the avionics/FSW. Control and monitor activities of the ground system are done on a distributed collection of computers, but all activities can be done effectively at one GDS workstation given modern computer-to-computer communication. See Figure 1 for a simple context diagram of a typical test bed.

## TABLE OF CONTENTS

1. INTRODUCTION.....	1
2. AUTOMATED SCRIPT ATTRIBUTES.....	2
3. SELECTED TECT KEYWORDS.....	3
4. SCRIPT EXAMPLES.....	4
5. TYPES OF TESTING AMENABLE TO TECT.....	5
6. RESULTS AND CHALLENGING ISSUES.....	6
7. LOOKING FORWARD.....	7
8. ACKNOWLEDGMENTS.....	7
REFERENCES.....	8
BIOGRAPHY.....	8

## 1. INTRODUCTION

The Multi-Mission Systems Architecture Platform (MSAP) project at the Jet Propulsion Laboratory (JPL) is chartered to develop reusable, robotic spacecraft components. These components may be divided into the categories of flight software (FSW), avionics, and the ground system. The avionics, FSW, and ground system components form a catalog from which mission planners may choose selected items, or the entire suite. Extensive documentation helps guide mission planners deciding where to draw the lines between “shrink wrap” usage, further development of components, or building from scratch where none of the components come close enough to meet a need.

The ground system has two main components: the Ground Support Equipment (GSE) and the Ground Data System

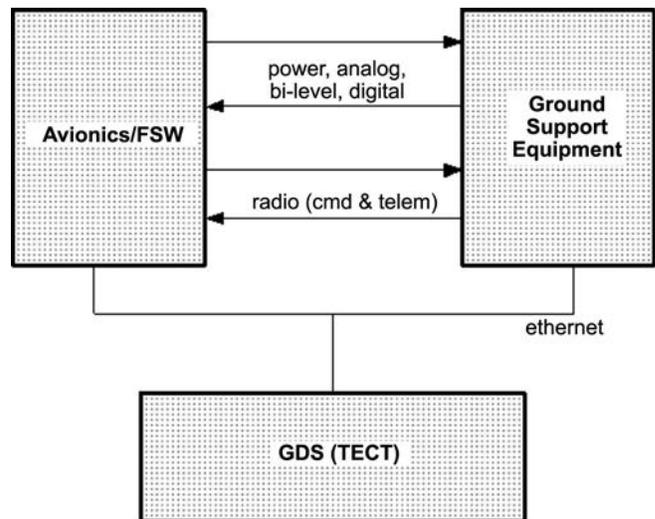


Figure 1 – Testbed Context Diagram

One of the features of the ground system is a script interpreter called the Test script Execution and Control Tool (TECT). TECT accepts procedures written as document files with a conventional word processor. A component of TECT called “document to script” (doc2scp) evaluates procedure documents, extracting embedded scripts from them. doc2scp extracts the scripts by looking for keywords which identify actions for the interpreter to take, including configuring the ground system, commanding the spacecraft, and evaluating telemetry. Thus, a procedure written for use with TECT may

<sup>1</sup> 1-4244-1488-1/08/\$25.00 ©2008 IEEE.  
<sup>2</sup> IEEEAC paper #TBD, Version 1, Updated TBD, 2008

perform autonomously any action available to an operator sitting at a ground system workstation.

TECT's development began at JPL in 1996 by Levesque *et al.* [1] in an attempt to shorten spacecraft system testing schedules, lower costs, and improve quality through test script automation. This level of automation implies not only the ability to execute a defined list of operations, but also to access feedback, evaluate it, and act on it.

TECT's native scripts are based on Tcl, a general purpose programming language, and this alone is sufficient to provide robust expression of test control. Tcl also provides TECT with easy access to other networked computers, such as the GSE, which implies the ability to send commands. Access to spacecraft downlink telemetry, however, is a different story. Access to the telemetry flow required a considerable investment in GDS systems programming. This reprogramming effort allowed TECT to sample individual channels from the spacecraft telemetry as it flowed through the GDS. Real time telemetry sampling gave TECT the ability to conduct closed-loop testing under the control of a Tcl script.

TECT supported SeaWinds/QuikSCAT and SIRTf before getting adopted by MSAP.

In a completely separate development at JPL, another test script tool developed by James Roberts and Michael Hasbach was created for use on the Mars Exploration Rover (MER) mission. This tool was aimed squarely at automating traditional test procedures, not scripts written in a widely used programming language. Test procedures were normally printed out on paper and then conducted by a human operator, who configured test equipment, issued commands, and recorded results, step by step. In order to speed up repetitious manual tests, a script interpreter was written. The MER scripting tool was able to "execute" traditional scripts (with slight extensions) – to the delight of those poor souls who, due to limited test hardware resources, were occasionally condemned to the 3<sup>rd</sup> shift! While this tool suffered from a well understood limitation in the quality of accessed telemetry, it was still applicable to many tests. (The major deficiency was due to its sampling method, which could not guarantee capture of transient values.) Still, it gave script writers the ability to issue commands and compare real time telemetry against predicted values in a straightforward way.

Though the development efforts for the MSAP and MER tools were separate they shared the same core goal: automated test execution based on a procedure not necessarily a script (in the sense of Tcl or Python). Up to this point test engineers had been responsible for upkeep of a test procedure and often times a corresponding test script. If not watched vigilantly this dependence between procedure and script could lead to a configuration management mess. The MSAP approach eliminated this potential by doing

away with the corresponding stand alone script since the script is embedded into the procedure, now the single source document.

## 2. AUTOMATED SCRIPT ATTRIBUTES

At the outset of MSAP, the systems and test engineers identified automated script based testing to be worth investigating, even if the team members had differing opinions as to its precise application. Rather than build a new tool from scratch, it was natural to consider extending TECT (with its exhaustive telemetry access) to accept traditional test procedures a la MER. This section describes the list of attributes MSAP's test and systems engineers determined TECT should have to meet MSAP's scripted testing needs.

Without any adaptation to MSAP's needs, TECT had several desirable capabilities, viz., TECT could

- (1) Issue commands to the spacecraft (and the support equipment), and
- (2) Intercept and evaluate spacecraft telemetry.

Unfortunately, TECT's only scripting interface at time was the Tcl programming language. While someone trained in a programming language could generally be expected to master the essentials of Tcl in a few weeks or months, the team wanted to make the scripting accessible to a wider range of engineers and scientists. With that motivation, a series of meetings between MSAP testers and TECT developers ensued, in which the following attributes were added to the two above.

- (3) Comprehensive test bed control
- (4) Optional operator intervention (full automation supported)
- (5) Comprehensive test log generation
- (6) Relaxed skill base for script authors (implies scripts can be reviewed by non-programmers)
- (7) Common word processing format for writing and editing script source documents
- (8) Machine execution of script source documents, allowing for machine translation from a common word processor format into one more readily executed
- (9) Nested scripting (scripts may call other scripts)
- (10) Conditional script execution based on telemetry evaluation (branching and looping)

A frequently stated goal for TECT was that it allow a systems engineer (or instrument specialist) to write original test scripts without first bothering to become a professional programmer or an expert in test bed operations. A consequence of this goal was it also let procedures be reviewed by an individual with no scripting knowledge in a format that is easily readable.

What we got was a tool that would accept MS Word® document files with embedded keywords, corresponding to the actions TECT should take. The keywords are loosely “verbs” followed by objects, where meaningful. In general, any line in the procedure that does not begin with a keyword is discarded by TECT. As long as keywords cannot reasonably be confused with ordinary word usage, TECT scripts can be made compliant with an organization’s traditional standards for test procedure formatting; in fact, existing test procedures, originally written to be run manually, can be retrofitted with keywords and run by TECT.

While not exactly a script attribute per se, test scripts enhance an MSAP catalog of offerings. Not only would a customer have avionics, FSW, and ground support, but the test scripts provide a immediate way to perform extensive exercises. And, review of these scripts would help give a sense of routine test operations.

### 3. SELECTED TECT KEYWORDS

In order to better explain the script examples in the next section, consider some common TECT keywords. The “>” (“greater than”) character is part of the keyword and serves to distinguish a keyword from a common English word appearing in the non-script part of a test procedure. Keywords can be followed by arguments.

#### *Uplink>*

This keyword sends a command to the spacecraft via the ground support equipment.

#### *Check>*

This keyword is used to compare a predicted value against the current value of a selected telemetry channel, relying on real-time access to channelized telemetry within the GDS as it arrives from the spacecraft. The *Check>* keyword expects to have a single named channel and a predicted value supplied in the script line, e.g.,

```
Check>  
  command_number _____ (0x0010)
```

The value inside parenthesis is called the predict. TECT records the result of the comparison in the test log.

Here, the script expects the channel to have the hexadecimal value “10” (decimal 16). If the last telemetry value for this channel was ‘0x10’ then the script will continue immediately; if not, the script will linger until the first channel update matches the predict or the timeout interval expires.

A basic paradigm for our scripts is the command/check pair where the check can wait a short while for the expected results of the command to become apparent in the telemetry.

#### *LatencyTimeout>*

The flight system (FSW and avionics) introduces a latency in the downlink and if a script compared a telemetry channel against its predict immediately after issuing a command, the script would likely report a misleading mis-compare. This latency is accommodated with *LatencyTimeout>* keyword which takes one argument, the maximum number of seconds to wait while checking for a match between a telemetry channel and its predict.

#### *InitCheck>*

This keyword takes no argument and effectively synchronizes every channel appearing in the script with the last telemetry value the GDS received. This operation has most benefit for those channels that deliver an incremented value.

#### *AskOnCheckFail>*

After sending a command, scripts normally check if the actual telemetry matches a predicted value. If there is a match, the script simply proceeds to the next test step. If the predict and actual do not match before the timeout expires, then the script can do one of two things, it can either pop up a message requiring operator input before proceeding or it can silently proceed to the next step. The script determines this behavior with a TRUE or FALSE argument to the *AskOnCheckFail>* keyword. FALSE allows the script to run without operator intervention. Either way, TECT will note the mismatch in the log.

#### *Wait>*

This keyword takes one argument, the number of milliseconds to pause the script.

#### *Comment>*

This keyword puts a text string into the log for purposes of documentation.

#### *Call>*

This keyword calls another TECT script.

*System*>

This keyword calls a (unix file system) executable file, such as a shell script.

#### 4. SCRIPT EXAMPLES

Given the brief introduction to some of the more common TECT keywords, the two simple scripts below should be quite understandable. The two scripts form a pair: a supervisor script which calls a subordinate script.

These examples consist of keywords only, unlike a real script, which is made up mostly of “boiler plate” and richly formatted test procedure, suitable for manual execution. Again it bears repeating that scripts are nothing more than conventional test procedures with embedded keywords.

The FSW I&T scripts generally assume a supervisor/subordinate relationship where the supervisor script sets up a test, calls one or more subordinate scripts to perform the nominal tests, and finally tears down a test. Each subordinate script operates in a modular way, leaving behind as few side effects as possible and removing any files or other artifacts it creates. In contrast, the Systems V&V scripts tend to be stand-alone documents in which the script executes its own activities mixed with prompts directing the operator to perform well defined tasks. No example of such a script is provided in this paper.

The supervisor script in Example 1 first identifies itself in the test log then calls a subordinate script to capture configuration information under which the test is run, such as command dictionary version, GDS software version, etc.

Then it calls another script to load and initialize FSW, which among other things will leave the spacecraft clock at a near zero value and the downlink telemetry rate at 10 bps. The AskOnCheckFail> keyword conditions TECT to continue executing without operator intervention in the case of a mismatch between a predicted telemetry channel value and its actual value within a specified time limit; LatencyTimeout specifies that time limit (in seconds). Next, the supervisor calls a series of subordinate scripts to carry out the nominal tests. Finally, the supervisor calls a Perl script to create a terse summary of the test log and declares its end in a log comment.

The subordinate script in Example 2 first places a self-identifying comment in the test log. The next keyword (uplink) will issue a command to FSW to set telemetry’s downlink rate to 5000 bps, then the script waits 40 seconds to allow for the expected latency as telemetry clears out any backlogged data. Once the telemetry reaching the GDS is current, InitCheck> gets a snapshot of the most currently instances of each telemetry channel that appears anywhere in the local script. At this point, the entire system is ready to begin testing. The next comment simply makes clear that the nominal test is about to begin. The next keyword sends a FSW command to set the spacecraft clock to an absolute value. Then the Check> keyword verifies a series of telemetry channels. First it will synchronize on the counter which keeps track of the number of dispatched commands and then confirm the clock took on the new value. The next command/check pair of keywords does a similar confirmation for a “no operation” command (which performs no actual function other than exercising the command subsystem). And finally, the last keyword makes a test log entry announcing the end of the subordinate script.

```

Comment> Test_Supervisor: Example 1
Comment> Collect testbed config info
Call> GSE_GDS_Revs.scp

Comment> Load and init FSW
Call> load_init_FSW.scp
Wait> 7000

Comment> Configure TECT
AskOnCheckFail> False
LatencyTimeout> 30

Comment> Commence testing
Call> SubordinateTest_001.scp
Call> SubordinateTest_002.scp

Comment> Extract abstracted test log (*_etl).
System> cd `todays_TECT_log_dir`; etl &

Comment> End of Test_Supervisor: Example 1

```

### Example 1 – A supervisor script, Test\_Supervisor.doc

```

Comment> SubordinateTest_001: Example 2
Comment> Configure FSW
Uplink> SET_DWN_RATE,5000
Wait> 40000
InitCheck>
Comment> Begin SubordinateTest_001

Uplink> SET_CLOCK,5000
Check>
command_number      _____ (+1)
sc_clock            _____ (> 4999)

Uplink> NO_OP
Check>
command_number      _____ (+1)

Comment> End of SubordianteTest_001: Example 2

```

### Example 2 – A subordinate script, SubordinateTest\_001.scp

## 5. TYPES OF TESTING AMENABLE TO TECT

MSAP FSW is developed, integrated, and tested in a conventional way. After FSW unit testing is complete, it is integrated with the rest of FSW and this integrated build is tested ensuring the new code behaves as expected without undesirable side effects. FSW Integration and Test (I&T) works as a gate keeper, ensuring the integrated FSW is “good enough” before releasing it to Systems Validation and Verification (V&V) for their testing. From a requirements perspective, the requirements verified by the Systems group decompose into the requirements tested by the FSW group, hence FSW I&T tests lower level requirements before

letting the Systems testers have at it. Both organizations use their own TECT scripts.

To date, we have used “production line” TECT procedures in three ways:

- (1) Low level requirements (FSW I&T)
- (2) Mid level requirements (Systems V&V)
- (3) Command regression testing (FSW I&T and Systems V&V)

The first type of procedure, the low level requirements test procedure, is characterized by intricate detail required to set up test conditions and then to exercise FSW through some precise required behavior. FSW I&T tests emphasize a “lights out” style of fully automated testing without operator intervention where failure in one part of a test does not necessarily preclude successful testing elsewhere. Automatically generated test reports end with a summary of each test section for quick review.

The second type of procedure, the mid level requirements test procedure, has a similar level of detail (sometimes including instrumentation not accessible by TECT scripts). System V&V procedures differ from the low level “batch” style of execution, however, with intimate operator involvement. Scripted prompts allows the System V&V scripts to weave together operator activity with scripted commands.

Both types of tests rely on the procedure to capture the testing details, including rationals. Systems V&V test procedures place responsibilities on the operator to judge the correctness of at least the higher level behavior in a step-wise fashion while the low level procedures defer operator judgment until after the test is complete.

The third type of test, regression testing of FSW commands, provides a quick way to determine if a change to FSW has introduced any defects. These tests tend to use minimal correctness checking, similar to the subordinate example script above.

## 6. RESULTS AND CHALLENGING ISSUES

We have found that it takes about as long to write a TECT procedure (with embedded scripting) as it does to write an equivalent manual test procedure. Most of the TECT keywords have an identical manual step in a test procedure; in fact, it is this equivalence that keeps TECT procedures easy to read. The keywords peculiar to TECT, i.e. those keywords that would not appear in a manual-only procedure, do not appear frequently and so are easy to deal with: `InitCheck>` and `LatencyTimeout>` are placed near the beginning of a procedure; `Call>` appears only in the top level (supervisor) scripts. (There is a mechanism that allows us to escape into pure Tcl code, but that technique has been used very sparingly and is not discussed here.) All in all, it takes little extra time to include the script specific lines to a traditional test procedure.

Finding and removing flaws from manual procedures and TECT scripted procedures require about the same effort. TECT procedures require a few trial runs - tantamount to debugging a simple program. Manually executed test procedures require an alert operator to redline errors in the procedure. So the costs to develop a mature procedure are roughly equivalent.

The `Call>` keyword might be an underrated capability. Given that someone has produced a library of mature TECT procedures, the `CALL>` keyword puts that library at your disposal. While manual procedures can direct the operator to perform activities from another procedure, this is not the norm; it is more typical to cut and paste entire sections of test procedures so there is a single “as run” paper document to serve as a controlled record. So far, our library of TECT procedures has been a significant enhancement mostly for quality control, i.e., collecting test configuration for every test and storing it in the test log.

Script execution takes about 25% to 50% of the time required to execute and evaluate an equivalent manual test procedure. (Scripts would run faster if not for the latency of telemetry to arrive at the GDS.) Scripts never overlook a test step, but then again, scripts never get curious and probe spacecraft behavior in an ad hoc way.

Engineers learn more about the spacecraft and test bed from manual test procedures since they are forced to pay attention to every step, but scripts produce useful results immediately. Of course, TECT procedures can be understood without needing special programming skills so the scripts form a basis from which empirical tests can be launched.

Given MSAP’s small budget and short life span, most of the script writers have been engineers with substantial programming skills. However, a few systems engineers have written significant test scripts to good effect. The early indications are promising.

We have run into a few challenging issues, however.

We have yet to resolve the conflict between simple keywords (understandable by non-programmers) and conditional control flow as commonly available with general purpose programming languages. TECT is based on Tcl and we have the ability to “escape” into purely Tcl code, but this defeats the goal of requiring a relaxed skill base. We are tending toward use of libraries to do the heavy lifting. A library function can be as complex as necessary – opaque, even, to non-programmers – as long as it can be described in “black box” terms understood to a non-programmer. So far, the higher level scripts use only linear control flow (no loops and no branching).

The comprehensive logs generated by TECT are too detailed and verbose for quick analysis. We wrote a filter to reduce the verbose log to a terse summary which is easy to understand if the script is available side-by-side. That filter can be included near the end of the TECT script. (Look for the “`etl`” system call at the end of the supervisor example script.)

Evaluation of test results can demand sophisticated analysis not easily expressed in algorithmic form. Simply put, some analysis done easily by eye can be difficult to program. This

is a tradeoff between fully automated testing and more operator intensive testing. It reduces to a question of budget, frequency of testing, and difficulty of analysis. At this time, our more sophisticated test procedures are relatively easy to write, infrequently run, and require human analysis after the test has collected all the data.

## 7. LOOKING FORWARD

### *Conditional Branching Tests*

The one requirement we have not yet achieved is the ability to do conditional branching and looping with TECT keywords alone. Part of conditional branching is the notion of a return status from subordinate scripts. Combined, they inspire the C programming style of conditional branching based on the result of a user defined function.

We have not yet advanced on these issues due mainly to budgetary considerations. The existing TECT capabilities have allowed us to write many useful procedures. The lack of even crude conditional branching does not preclude us from writing many more. Hopefully, this requirement will not remain dormant indefinitely.

### *Avionics Simulator*

As stated earlier, TECT procedures (or manual procedures for that matter) must be run on the avionics to detect flaws in the procedures themselves before they can perform their nominal function in a reliable way. The avionics are expensive and therefore a relatively scarce resource.

MSAP is developing an avionics simulator that will run on a generic workstation, such as a single board computer, and will accept FSW. Further, the simulator will interact fully with the ground system, which means it can be exercised by a test operator running a manual procedure or a TECT procedure. A correctness test of the simulator is that it behave exactly like FSW running on the avionics itself. This implies that any test procedures succeed and fail exactly the same way.

This simulator represents a major productivity boost for procedure development. This simulator will permit TECT procedure development to scale well with respect to the number of engineers since they can polish the procedures on relatively inexpensive workstations before reserving valuable time on the real test beds. This is an especially significant issue during several “crunch” stages of a project’s life when test bed utilization approaches 100% on a 24x7 basis.

### *Institutional Directions*

The Mars Science Lander (MSL) is currently the dominant project under development at JPL. MSL’s test bed group has taken a different scripting direction from MSAP’s and

created a scripting tool with the acronym MTAK, which is based on the Python language. MTAK scripts are written exclusively in Python and make use of a library of Python functions to do many of the same things that the TECT keywords do. MTAK puts more expressive power at the disposal of a script writer owing to its direct use of a general programming language (Python), but restricts its script authors and reviewers to those with some programming skills.

MSL has introduced several innovative features to MTAK, one of which is a sequencing engine with fine time resolution running within the GSE. MTAK scripts effectively program the sequence engine to manipulate instrument simulators (different from the avionics simulator) running within the GSE. Since these simulators provide input to avionic sensors, this opens up the arena of scripted fault injection.

Given MTAK’s strengths, it may be that MTAK will eclipse TECT, but that does not necessarily imply the end of TECT procedures. Although there is currently no way to run TECT scripts in the MTAK environment, it should not be a major challenge to modify the doc2scp program to translate TECT procedures into Python scripts.

## 8. ACKNOWLEDGMENTS



The authors would like to thank Tom Fouser and Leticia Montanez for project support in developing TECT scripting capability for MSAP; thanks also to Carol Glazer for similar support from Ground Data Systems. Danny Lam, the previous FSW I&T Lead, encouraged TECT deployment. Joe Diep and Mike Gan adapted and extended TECT for our use. Carolina Barltrop helped define our requirements. Liz Johnson was an early non-programmer user. Cindy Huynh (a spacecraft test veteran) has been a prodigious script author. Melody Safavizadeh is a new team member who has provided value questioning the ways that old-timers do things. And special thanks to Roger Klemm for some valuable review and comments.

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement

by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

## REFERENCES

- [1] M. Levesque, J. Louie, A. Guerrero, "Test Execution Control Tool: Automating Testing in Spacecraft Integration and Test Environments," 2000 IEEE Aerospace Conference Proceedings, Volume: 2, 389–395, March 18–25, 2000.

## BIOGRAPHY

**Brian Bone** began working at JPL in 2001 after obtaining a BS in Mechanical Engineering. Before joining the Multi-Mission System Architecture Platform (MSAP) he supported the Mars Exploration Rover (MER) and Cassini projects. In

2005 he received his MS from the University of Southern California. Concurrently in 2005, he started supporting MSAP in an integration and test role. He has since moved on to support the DAWN and most recently Juno missions.

**Don Gibbs** is the Flight Software Integration and Test Lead for the Multi-Mission System Architecture Platform (MSAP). He has been at JPL since 1983, first as a software engineer in the Deep Space Network and then as a test team member of numerous flight projects. He earned his BS in Information and Computer Science from the University of California at Irvine (1978) and his MS in Computer Engineering from the University of Southern California (1996).



