Automated Testing of the Deep Space Network's Uplink Subsystem

William H. Duquette
Jet Propulsion Laboratory
William.H.Duquette@jpl.nasa.gov

ABSTRACT

This paper describes some of the lessons we learned in implementing two different Tcl-based test frameworks used to automate testing of the Deep Space Network's new Uplink Subsystem, and discusses the advantages and disadvantages of each. The first framework was external to the Uplink Subsystem's software and provided complete control of the environment; the second was integrated with the software and provided less control, but motivated more and better testing. Some implementation details of the second framework are discussed as well.

1. BACKGROUND

1.1 The Deep Space Network

The Deep Space Network (DSN) is NASA's primary ground system for spacecraft telecommunications. A world-wide network of antennas and related hardware and software, it consists primarily of an operations center at the Jet Propulsion Laboratory (JPL), three Deep Space Communications Complexes (DSCC's) in California, Spain, and Australia, and the voice and data networks which connect them. The DSN is primarily used for tracking NASA spacecraft, but also supports the European Space Agency (ESA) and others.

1.2 Spacecraft Tracking

Each DSN complex operates a number of dish antennas and related "subsystems". Each subsystem is a collection of hardware and software which supports one element of a successful track. The antenna pointing subsystem, for example, is responsible for keeping the antenna pointed at the spacecraft; the telemetry subsystem is responsible for decoding the spacecraft's downlink signal and forwarding the telemetry data to JPL. Some of these subsystems are associated with a particular antenna; others can be assigned to work with any antenna. Overseeing everything is a DSN operator sitting at a Monitor and Control (M&C) workstation.

First, an antenna and all necessary subsystems are "assigned" to track a specific spacecraft, and given over to a particular DSN operator. The operator must then configure and calibrate the subsystems to support the track. During the track itself, the operator monitors the assigned subsystems for hardware failures and other problems, and occasionally takes other steps, such as enabling the modulation of command data onto the carrier signal for uplink to the spacecraft. At the end of the track, the subsystems are unconfigured and returned to the pool for assignment.

1.3 The Monitor and Control Protocol

All communication between the operator and the subsystems is via the Monitor and Control Protocol. This protocol is implemented by a multi-threaded, socket-based API, and carries five basic kinds of message:

"Configuration Control Notices" are sent by M&C to the subsystems to assign them to support a track, and later to unassign them again.

"Event Notices" are sent by subsystems to M&C to notify the operator of progress made or of problems observed. There are a number of kinds of event notice, each with a specific purpose and level of severity, but fundamentally each is a text message intended to be read by the operator.

"Operator Directives" are requests for action sent by M&C to the subsystems. "Commands" would be the more normal term, but in this domain the word "command" is exclusively used to mean "data sent to a spacecraft." An operator directive is a text string consisting of the directive name and zero or more arguments. Operators have historically typed directives in by hand.

"Directive Responses" are sent by subsystems to M&C in response to operator directives. There are several categories of response, ranging from REJECTED to PROCESSING to COMPLETED. Colloquially, a directive is said to be accepted if the subsystem attempts to carry it out, and rejected otherwise. Each response includes a text message intended to be read by the operator.

"Monitor Data Segments" convey detailed configuration, status, and performance information. Each segment consists of one or more "monitor data items". Each item is named, and may be a text string, an integer, a floating point number, or one of several DSN-specific enumerated types. The subsystems assigned to a track use monitor data segments to communicate amongst themselves and with M&C; monitor data is also used to populate GUI displays on the M&C workstation. Monitor data is "published" by the subsystem, and may be "subscribed" to by any interested party.

All of these messages are either ASCII text or are easily converted to and from ASCII text; hence scripted control of a subsystem is a natural thing to do. The wrinkles are due to the asynchronous nature of the interface, and to the multi-threaded nature of the M&C API.

1.4 The Uplink Consolidation Task

Our project began in late 1997 as the "Command Replacement Task". The DSN command subsystem receives spacecraft command data blocks from the spacecraft's flight team, queues them up, and ultimately modulates them onto a subcarrier signal. Our job was to design and build a new command subsystem on modern hardware; the new subsystem was to be called the "Command Control Processor", or "CCP".

Over the years a body of reusable subsystem software had accumulated to support the building of subsystems, but as we were starting the DSN was just beginning to move to a new implementation of the M&C protocol described above. The new implementation was radically different from the old. As part of our job was to build the subsystem to the new standard, we were able to start from scratch and build a new subsystem infrastructure on top of the new M&C protocol.

After we'd released version 1.x of our new Command subsystem, and while we were working on version 2.x, our scope was expanded, and the task was renamed "Uplink Consolidation". In version 3.x, our software was to provide a consolidated monitor and control interface for the DSN Command, Exciter/Transmitter, and Ranging functions under the name of the "Uplink Subsystem", or "UPL". The Ranging function is used to measure the round-trip light time to the spacecraft, and works by generating a sequence of ranging tones which are turned around by the spacecraft and detected by the downlink ranging hardware. The Exciter/Transmitter function produces a carrier signal, modulates the Ranging and Command signals on to it, and steps it up to the required output power before passing it along to the antenna.

At the time of writing, we have finished the implementation phase for the first consolidated version, 3.x, and are preparing it for acceptance testing.

1.5 DSN Testing

Delivering a subsystem to the DSN involves four levels of testing:

- Unit testing by the individual developers
- Integration testing by the subsystem's test team
- Acceptance testing by DSN complex personnel, with help from the subsystem's test team
- Operational "soak" testing. This is a probationary period in which the subsystem is used for real operations, but is under close scrutiny.

Ideally, any automated test solution would apply broadly, allowing automation of both developer unit tests and the official test plans used for acceptance testing.

My responsibility on the Command Replacement/Uplink Consolidation task was the interface with the Monitor & Control subsystem, i.e., the code which received and responded to CCN's and operator directives, and which sent event notices and published monitor data. I soon realized that a suite of automated regression tests could save a great deal of time.

Because we were developing a new subsystem infrastructure as part of our overall effort, we produced a lot of library code, mostly written in C. Writing test programs for these libraries was generally quite straightforward. The difficulty lay in automating the testing of the subsystem software as a whole.

2. EXTERNAL TESTING

2.1 The Monitor & Control Shell

Prior to this task, I was part of the M&C subsystem team, responsible for developing GUI infrastructure. This involved using tempermental early versions of the new M&C protocol and API. To add in troubleshooting, debugging, and testing I wanted to create an interactive shell which gave access to the M&C API. The extensible scripting languages then available to me were Perl, Tcl, and Python. Perl I rejected out of hand as it doesn't provide an interactive shell. At that time (1995) Tcl was more widely known than Python, and Tcl's command syntax provided a more pleasing shell interface, so I selected Tcl, and created an application called *monsh*, the Monitor and Control Shell. When I left the M&C team to work on the Command Replacement task, I took *monsh* with me.

A digression: because of its interactive nature, *monsh* has remained my preferred tool for troubleshooting M&C communications problems; the effort spent developing and maintaining it has been worthwhile for this reason alone.

2.2 Monsh Implementation

For the most part, *monsh* is a straight-forward Tcl extension which wraps the calls in the M&C API and adapts them to Tcl style. The only interesting implementation detail is *monsh*'s handling of the M&C API's callback functions.

Rather than using an event loop, the M&C API is multi-threaded, using POSIX threads. The client registers callback functions with the API; when incoming messages arrive, the appropriate callback is called *in its own thread*. Consequently, it was necessary to multiplex these messages into the Tcl event loop for processing, as follows: when each message arrives, its data is copied into a new dynamically-allocated packet, and a pointer to the packet is written to a Unix pipe. The other end of the pipe is registered with a Tcl input file handler which reads the packet from the pipe, unpacks the data, and calls the user's Tcl callback code. This is all done in C.

Monsh can run as a GUI (*tkmonsh*), in which case the event loop is always available; in a typical test scenario it is more usual for a *monsh* script to send operator directives to the subsystem and enter the event loop just long enough to receive the response. *Monsh* provides the commands *mon_event_loop* and *mon_end_loop* to enter and exit the event loop as needed.

2.3 Naïve Test Scripting: The *CcpTest* Package

My first effort at scripted testing for Uplink was in support of my work on version 1.x of the subsystem, the "Command Control Processor", or CCP.

The initial goal was to support exhaustive testing of the subsystem software, i.e., all nominal scenarios plus all failure scenarios resulting in 100% code coverage. Consequently, test scripts needed to be able to do the following things:

- Configure the subsystem's initialization files and local directory space before invoking the subsystem.
- Invoke the subsystem software as needed.
- Simulate the M&C Subsystem, including the sending of Configuration Change Notices (CCNs).
- Send operator directives and verify the category and text of the responses.
- Verify event notices and monitor data values.
- Scan and verify entries in the subsystem's debugging log.
- Simulate communication with other subsystems, as needed, notably M&C and Exciter/Transmitter.
- Terminate the subsystem software as needed.

In my initial implementation, the test software consisted of two layers on top of *monsh*. The first was a package called *MonTest* which implemented a generic M&C test harness based on taking action and then waiting in the event loop for a response. Given a list of monitor data item names and values, for example, the *monVerify* command verified that each item's published value was as specified, retrying periodically, if necessary, until the values were correct or a set timeout expired. Similarly, the command *sendDir* sent an operator directive to the program being tested; *expectResp* waited for the response, which had to match a specified pattern. Scripts written using *MonTest* don't provide a GUI; instead, individual *MonTest* commands enter and leave the event loop as necessary. *MonTest* is still used today to test some of our low-level library code.

The second layer was an *ad hoc* collection of code specific to our subsystem; it simulated the M&C and Exciter/Transmitter subsystems, configured the CCP's initialization files, invoked the subsystem before each test and terminated it afterwards, and in general provided a convenience layer on top of *MonTest*. This package was called *CcpTest*.

We'd discovered that it could be dangerous for multi-threaded programs to fork-and-exec other tasks (under POSIX Threads, at any rate). However, it was much safer if any forking was done prior to creating any new threads, and so *CcpTest* used a two-level approach, as follows:

- Each *CcpTest* script ran as a separate invocation of the *monsh* interpreter.
- First, the script set up the subsystem's environment as desired.
- Next, it invoked the subsystem software.
- Next, it wrote the body of the test to a temporary file, and invoked *monsh* to execute it.
- The child script performed a series of tests, logging all activity and results to standard output.
- When the child script terminated, the parent script captured the output and scanned it for failures.
- The parent script then wrote the complete log and the results to standard output.

I wrote several dozen detailed test scripts using this approach. Over time, a number of disadvantages became evident. Because the test scripts controlled the subsystem environment, they were remarkably fragile. These were early days, and the contents of the subsystem initialization files were updated regularly, as were many other things. Consequently, the test scripts were frequently broken by changes having nothing to do with what they were testing, and required frequent updating. Human nature being what it is, this meant that they didn't get run very often.

Next, each test script contained ten to fifteen lines of boilerplate code that were almost (though not quite) the same for each script--ten to fifteen lines were both necessary and obscure.

Those developers I tried to interest in using *CcpTest* for their own unit testing were put off by the steep learning curve. None of them were Tcl programmers, nor did most of them need to test the ins and outs of subsystem invocation and failure as I did. Our lead tester was interested, but writing scripts was sufficiently difficult and his time sufficiently constrained that nothing happened.

2.4 Better Test Scripting: The *ccp_test* Tool

About halfway through the development of version 1.x I tried a slightly different approach in an attempt to resolve some of these problems. The result was both easier to use and less flexible; where *CcpTest* was capable of invoking and testing any desired program, the new approach was focussed on testing the CCP subsystem software only.

First, the collection of *ad hoc* code loosely named *CcpTest* was extensively revised, refactored, and redesigned for clarity and consistency. Second, a tool called *ccp_test* was written to encapsulate all of the ugly details of getting a test script up and running. The input to *ccp_test* was a Tcl file that defined a series of independent test cases. *Ccp_test* was responsible for invoking the subsystem software as needed, and for executing each test case, in sequence or in random order, once or repeatedly. It was a great improvement, doing away with most of the ugly boilerplate and easing the learning curve somewhat. It didn't solve the most serious problem, the fragility of test scripts. Nevertheless, *ccp_test* was used by our lead tester to do operator directive syntax checking as part of normal integration testing. It was never used for acceptance testing.

As before the result of the fragility of the test scripts was that I didn't run them all that often so that I wouldn't have to update them all that often. Moreover, I never completely finished porting my old scripts over to the new framework.

2.5 Subsystem Evolution

When version 1.x of the subsystem software was complete, we began development of version 2.x. It was at about this time that we were tasked to do Uplink Consolidation in our version 3.x, and as most of the 2.x changes didn't involve me I spent most of my time working on new infrastructure to support our vastly expanded set of requirements for 3.x. Some of this infrastructure went into the 2.x version to make it more robust and to save time later on, and the related architectural changes broke *ccp_test*'s handling of subsystem invocation and termination. The code tested by my existing test scripts remained largely unchanged, however, and being busy with other things I allowed *ccp_test* to remain broken until late in the development cycle.

Then version 3.x development began in earnest, and the architectural changes broke *ccp_test* again. One simulated subsystem no longer needed to be simulated; and as the subsystem

became much larger and more complex, the fragility problem became correspondingly more difficult. I made several abortive efforts to update *ccp_test* for version 3.x, but motivation was lacking. While *ccp_test* added value, it was unwieldy and much of that value was eaten up by the constant rework.

And then the death knell rang. *Ccp_test* used an M&C subsystem simulation to test the subsystem's reaction to configuration change notices. At about this time, a policy change came down that indirectly (but effectively) banned the use of such simulators on the main LAN at a DSN complex. As designed, *ccp_test* could never be used for acceptance testing.

3. INTERNAL TESTING

I've referred to my previous efforts at scripted testing of our subsystem software as "external testing" because the test software was completely independent of the subsystem software itself. For several reasons, I decided to try a completely different approach:

First, control of the subsystem initialization files can be useful for unit testing, but isn't required for integration or acceptance testing: the initialization files and environment must be set correctly at subsystem installation and shouldn't be changed thereafter.

Second, control of subsystem invocation and termination is useful for testing certain specific failure modes, but isn't required for the vast bulk of conceivable test cases.

Lastly, a scripting mechanism that could assume that the subsystem software was successfully configured and invoked would be freed from the fragility problem. Environmental changes required by software changes would have to be resolved before the scripting mechanism was even available, thus removing this concern from the individual scripts.

An internal scripting facility would run as part of the subsystem software. Consequently, it would not be able to control subsystem initialization or invocation…but at the same time it would be always available whenever the subsystem software was running, and would be controlled using the same interface as the rest of the subsystem software. The moral was clear—by relaxing my two most stringent requirements, I should see an increase in usability and stability.

3.1 The Legacy AutoTester

The infrastructure used with the previous version of the M&C API had included a scripting facility, the "AutoTester", with which scripts residing on the subsystem's disk drive could be invoked and controlled somewhat interactively via operator directives. In at least one case, the AutoTester had been used to automate the bulk of a subsystem acceptance test. This was a model worth examining.

It soon became clear that simply porting the AutoTester to our platform was out of the question. It was closely tied with the obsolete subsystem infrastructure, and made a number of assumptions (such as the use of shared memory for storing monitor data items) that were invalid for our architecture; many of these assumptions were evident in the scripting language itself. Moreover, its scripting language was the typical result of trying to write an *ad hoc* extension language that's as easy as possible to parse and execute line-by-line, with ugly control structures bolted on afterwards. Now that many worthwhile extension languages are available, it looked especially bad.

However, the AutoTester's command set and operator directive interface had proven themselves useful for subsystem testing and automation, and were clearly worth emulating in a better language.

At this point in our development, our project was using two scripting languages: Perl and Tcl. Of the two, only Tcl was designed from the ground up for embeddability; moreover, thanks to *monsh*, we had experience with Tcl's C API. So it was an easy decision to base the new facility on Tcl.

3.2 Uplink Software Architecture

The Consolidated Uplink Subsystem consists of a Solaris workstation called the Uplink Processor Assembly (UPA) and a number of hardware boxes. In this paper, the term "Uplink Software" refers only to the software that runs on the UPA.

The Uplink Software is a distributed application consisting of a number of application programs, or "tasks" built on top of our new subsystem infrastructure, the Uplink Common Software. Generally speaking, each task is responsible either for managing one external interface, or for coordinating the work of other tasks. The tasks communicate among themselves by means of messages sent across Unix-domain sockets. Internally, the execution of each task is controlled by a *select()*-based event loop similar to Tcl's own.

3.3 The Uplink Scripting Engine

The internal scripting facility could be built into an existing task, or implemented as a new task. We chose to write a new task, the Uplink Scripting Engine, for these reasons:

Code Independence: Because the Scripting Engine was to be used to test the Uplink subsystem's software, it should, so far as was possible, rely only on independently-tested infrastructure code and not on subsystem application code.

Realistic Testing: The main function of the Scripting Engine is to exercise the Uplink Software by means of M&C messages, especially operator directives and responses. For end-to-end testing, these messages should come from outside the task that handles them, and should be handled identically to messages coming from outside the subsystem.

Failure Recovery: Because the Scripting Engine is primarily a test tool, fatal errors in the Scripting Engine should not be allowed to affect the operation of the remainder of the Uplink Software. Being a separate task, the Scripting Engine can be brought up and down without hazard.

4. ENGINE IMPLEMENTATION

The following sections will discuss the features and implementation of the Uplink Scripting Engine.

4.1 Operator Control

Scripting is controlled by the **ACTL** operator directive. The acronym **"ACTL"** stands for something like "Automation ConTrolL"; it was chosen for consistency with the legacy AutoTester.

ACTL <scriptname> [<arg> [<arg>…]]

Invokes the named script. The script is found by searching for a file called "<scriptname>.tcl" along a list of script directories. When the script is invoked, the variable *argv* will contain a list of the arguments (if any).

The operator may invoke only one script at a time; the script must terminate before the operator may invoke any subsequent script.

The Scripting Engine sends event notices to M&C at script invocation and termination; the name of the current script is published as a monitor data item for display.

ACTL RESM [<arg> [<arg>…]]

A script can request operator action by sending an event notice to M&C and suspending its execution. After taking the action, the operator uses **ACTL RESM** to cause the script to resume execution. When execution continues, the variable *rargv* will contain a list of the arguments to **ACTL RESM**, if any.

ACTL END

Terminates execution of the current script whenever it next suspends (e.g., to wait for a directive response).

ACTL RESET

Resets scripting by killing the Uplink Scripting Engine task, which will be restarted automatically. This is a drastic step, but protects against scripts that never suspend, like this one:

```
while {1} { }
```

4.2 The Scripting Language

The Uplink Scripting Language is based on the standard Tcl 8.0.3 interpreter. Some of its commands are implemented in C and loaded into Tcl interpreters as needed; others are defined in a Tcl package called *UlcSe* which is *package require*'d by each created interpreter

The implementation of the Uplink Scripting language depends heavily on careful control of access to the event loop. Consequently, the *after* and *vwait* commands are disabled.

The following commands are modified from their normal Tcl definition:

exit

Normally *exit* terminates the program, which in this case would be the Uplink Scripting Engine itself. In this application, *exit* terminates execution of the current script, returning control to the caller, which may be the Scripting Engine or a parent script. This was define by redefining *exit* as follows:

```
proc exit {dummy} {
    return -code error
}
```

The script invocation code catches the error, sees that there is no error message, and presumes that it's a normal termination. Execution then continues in the parent.

bgerror

This command is redefined in the usual way to output background errors to the Scripting Engine's log.

The following is an incomplete list of the Uplink-specific commands.

susp ?<message>?

Suspends script execution, sending the <*message*> to the M&C operator as an event notice. If <*message*> is not given, a standard message is used. The message will usually tell the operator to take some action, followed by sending the **ACTL RESM** directive. The script can request operator input by telling the operator to send **ACTL RESM** with particular arguments.

call <script name> ?<arg>…?

The *call* command is designed to allow test scripts to call each other safely, with minimal worry about interference through shared variable names. The command searches for the named script just as the **ACTL** directive does, and invokes it in a new slave interpreter (see 4.3, Managing Script Execution). The current script is suspended while the called script runs to completion; then execution of the current script is resumed. If the called script throws an error or a fatal test failure, then all script execution is terminated and control returns to the Scripting Engine itself.

As with the **ACTL** operator directive, *argv* contains a list of the remaining arguments, if any.

include <script name>

Searches for the named script just as the *call* command does, and sources it into the current interpreter. This is the normal way for authors of test scripts to load any libraries of test code they might have written. Packages in the *TCL_LIB_PATH* can also be *require*'d, but few test script authors are Tcl programmers, and using packages would require them to learn an additional mechanism.

wait <seconds>

Pauses script execution for the specified number of seconds. The argument may include a decimal fraction, so it's possible to wait for a fraction of a second.

fail <message>

Signals a test failure. A script may choose whether or not failures should terminate execution; either way a failure message is logged and sent to M&C. If failures are fatal, then *fail* is equivalent to *error*. Failures are fatal by default.

od <directive text>

Sends an operator directive to the subsystem, and waits for it to be accepted. The test fails if the directive is rejected.

odrej <directive text>

Sends an operator directive to the subsystem, and waits for it to be rejected. The test fails if the directive is accepted instead.

dr <response pattern>

<*response pattern*> is a *string match* pattern. This command attempts to match the last received directive response text against the pattern. The test fails if it doesn't match.

This command is always used together with *od* or *odrej*, as follows:

```
od "MOD CMD E"   ;# Enable modulation
dr "*enabled*"   ;# Test the response
```

mon <item name>

The Scripting Engine receives a complete copy of all monitor data items published by the Uplink Subsystem; a script can access the value of any item by passing its name to this command:

```
set status [mon Status]
```

verify <condition> ?wait <seconds>?

This is a generic testing command, usually used in conjunction with the *mon* command. By default, it simply verifies that <*condition*> (a Tcl expression) is true, calling *fail* if it is not. If the *wait* <*seconds*> clause is added, then it will wait for up to the specified number of seconds for the <*condition*> to become true. For example,

```
# Verify that the subsystem is not
# configured for service.
verify {"Waiting" == [mon Activity]}

# Configure the subsystem for service
# with spacecraft 99
od "CNF SCN=99"

# Wait until the subsystem has been
# configured. Fail if it hasn't
# successfully configured within 30
# seconds
verify {"In Service"==[mon Activity]} \
    wait 30
```

event <event list> ?wait <seconds>? ?with <commands>?

Each event notice has a unique integer ID. This command waits until an event notice is received that has an ID in the <*event list*>. By default it waits indefinitely; if the *wait* <*seconds*> clause is include, it waits for up to the specified number of seconds, and fails if a matching event notice hasn't yet been received. For example,

```
event {2001 2002} wait 30
```

Sometimes the relationship between an event notice and the actual event that triggers it is problematic. Consider the following case:

```
# OD sends event 100, then responds
od "CNF"

# Always fails
event 100 wait 30
```

The *event* call always fails because the event notice is received before the *od* call returns. The *with* <*commands*> clause handles this case:

```
event 100 wait 30 with {
    od "CNF"

    verify {
        "In Service" == [mon Activity]
    } wait 30
}
```

In this case, *event* executes its body, which may contain any desired commands (including nested *event* calls), and will end successfully if a matching event is received at any time from the start of executing <*commands*> up to 30 seconds after it's done executing <*commands*>.

The Uplink Scripting Engine understands many other commands, including commands to send event notices, write to the log, unpack *argv* and *rargv* in convenient ways, and so forth, but these are the commands with interesting implementation details.

4.3 Managing Script Execution

Tests scripts will be written at different times, by different people, all of them making different assumptions and few of them (initially, anyway) familiar with Tcl. If all test scripts were run in a single interpreter, it would be easy for one script to leave garbage behind in Tcl's memory that could distort the execution of a subsequent script. Therefore, each script must be run in its own slave interpreter. The corollary is that scripts cannot share data with each other through Tcl variables, but this turns out not to be a problem. For this application, the essential data is the state of the Uplink subsystem itself. Scripts can examine the subsystem directly, by querying monitor data or sending query directives. Moreover, the effects of running a script should be apparent as changes in the subsystem's state. (This is called positive closed-loop control, and it's a requirement placed on all DSN subsystems.) Thus, there is little need for two scripts to share data in any other way, except by the limited means of passing arguments from a calling script to a called script.

If a script needs procedure definitions defined in another file, it can *include* that file; but the included definitions will be valid only in that script.

The Scripting Engine creates and initializes a master Tcl interpreter at start-up. It is used only for creating slave interpreters to execute scripts invoked via the **ACTL** operator directive. Similarly, when a script uses the *call* command, a slave interpreter is created to execute the new script. In both cases, the following Tcl code does the work:

```
proc ::UlcSe::SlaveSource {
    script arglist filename
} {

    # FIRST, Create the slave and load
    # it with the Scripting Language
    set slave [interp create]
    PushSlave $slave

    # NEXT, pass the args to the interp
    $slave eval [list set argv $arglist]

    # NEXT, invoke the script.
    set msg ""

    try {
        $slave eval source $filename
    } catch msg {
        # Do nothing
    }

    # NEXT, Restore the old one.
    PopSlave
    interp delete $slave

    # NEXT, report errors or return normally
    if {"" != $msg} {
        if {[string match "in *:*" $msg]} {
            error $msg
        } else {
            error "in $script: $msg"
        }
    }

    return
}
```

The arguments to *SlaveSource* are the script's name, i.e., its file name less the path and extension, the list of arguments, if any, and the full file name of the script. Technically speaking, the first argument is unneeded as it can be computed from the file name, but *SlaveSource* needs both and as the caller always has both they are both passed.

First, a new slave interpreter is created. We don't use a "safe" interpreter as operationally there is no way to give an untrusted script to the Scripting Engine, and it is often useful for scripts to interact with the environment.

Next, the new slave is pushed onto the slave stack by *PushSlave*. This does two things. First, it identifies the slave as the current interpreter, which is necessary for the handling of several of the Scripting Engine's **ACTL** operator directives, and second it loads the Uplink Scripting Language definitions into the interpreter. At present the slave stack is an explicit stack with a maximum depth of 50. This limitation could be removed by relying on the implicit stack of slave interpreters created by recursive calls to *SlaveSource*.

Next, the argument list is placed in the Tcl variable *argv*.

Next, the script is *source*'d into the new slave. The *try…catch* construct is a simple wrapper around the standard Tcl *catch* command; if an error is thrown, then (in this case) the variable *msg* is set to the error message.

Next, if any real errors were reported, the variable *msg* will be non-empty (recall that the *exit* command has been redefined to throw an error with the empty string as its message). Any such errors are reported.

4.4 Managing Asynchronous Waits

The Uplink Scripting Engine is an event-driven application based on Tcl's event loop. Asynchronous waits are handled by entering Tcl's event loop recursively and staying there until the desired event occurs. All of this is done in C code. I'll present two representative examples.

On a *susp* command, the Scripting Engine pauses script execution until an **ACTL RESM** operator directive is received. The *susp* command is implemented in C; here is the relevant code:

```
SeGoal result = waitFor(SeRESUME);

if (result == SeEND)
{
    return endScript(interp);
}

return TCL_OK;
```

The *waitFor()* function is defined as follows:

```
static SeGoal
waitFor(SeGoal goal)
{
    script.waitingFor = goal;
    script.gotEvent = SeNOTHING;

    while (script.gotEvent != SeEND &&
           script.gotEvent != goal)
    {
        Tcl_DoOneEvent(TCL_ALL_EVENTS);
    }
    script.waitingFor = SeNOTHING;

    return script.gotEvent;
}
```

It processes events repeatedly until some event sets *script.gotEvent* either to the goal event or to *SeEND*. The latter code indicates that the **ACTL END** operator directive has been received, terminating all script execution.

Thus, we can see that the *susp* command waits until the operator sends **ACTL RESM**, which sets *script.gotEvent* to *SeRESUME,* or **ACTL END**, which sets *script.gotEvent* to *SeEND*. In the latter case, the *endScript()* function simply sets the command's result string to "Script execution terminated by operator." and returns *TCL_ERROR*.

This pattern is repeated for each of the commands that pause script execution. The *od* command, for example, sends a directive and waits for a response. When the response is received—and there is a timeout mechanism inherent in the M&C protocol, so a

response is always received—the callback will cache the response data and set *script.gotEvent* to *SeRESPONSE*. Since only one directive can be sent at a time, this indicates that the desired response has been received. The relevant code is as follows:

```
result = waitFor(SeRESPONSE);

if (result == SeEND)
{
    return endScript(interp);
}

if (script.lastResp.cat != RESP_COMPLETED &&

    script.lastResp.cat != RESP_STARTED)
{
    return logIgnorableFailure(interp,
        "Expected COMPLETED or STARTED");
}

return TCL_OK;
```

The code is essentially identical to that in the *susp* command, except that *od* goes on to test the response. The *logIgnorableFailure()* function logs test failures, and returns *TCL_OK* or *TCL_ERROR* depending on whether test failures are currently fatal or not.

The *odrej*, *wait*, and *event* commands are implemented in the same way. *event* is somewhat more complicated because it can call itself recursively; therefore, it needs to push a record on a stack for each call.

5. USAGE IN PRACTICE

For our first delivery, *ccp_test* was used for both unit and integration testing. In our current delivery, our new internal Scripting Engine has been used extensively for unit testing, and is beginning to be used for integration testing; moreover, portions of the acceptance test plan are currently being written as Scripting Engine scripts. However, these latter scripts are not yet at a point where analysis is worthwhile. The following table shows some statistics.

		V1.x: External	V3.x: Internal
Unit Test	Raw lines of code	909	8711
	Stripped lines of code	389	4904
	Tests	170	5307
	Density (tests/line)	0.44	1.08
Integration	Raw lines of code	5889	n/a
	Stripped lines of code	2577	n/a
	Tests	1538	n/a
	Density (tests/line)	0.60	n/a

Table 1: Test Density, External vs. Internal Testing

In every case, the code counts are for actual test cases; the test harness code is excluded, except for boilerplate that appears in the test case files. "Raw lines of code" is simply the number of lines of text in each test case file. "Stripped lines of code" is the number of lines after comment lines and blank lines are deleted.

"Tests" is a rough count of the number of conditions tested in each file after boilerplate and test set-up code was deleted. I counted one for each verified event notice, verified monitor data value, verified directive response category (i.e., success or failure), and verified directive response string, and matched line of debugging log.

"Density" is simply the ratio of "Tests" per "Stripped line of code." I arrived at this as a rough measure of how easy it is to write a test case using each framework.

The low code density for unit tests using the External Harness is because many of the unit tests were written for the test harness that preceded *ccp_test*. The 60% value found in the integration tests should be more representative for *ccp_test*-style test cases.

Only one developer (myself) used *ccp_test* for unit testing; by comparison, two developers used the Scripting Engine for unit testing, and we wrote over 30 times as many tests. This increase has two causes. First, the version 3.x system is much larger, and there are more things to test. Second, Scripting Engine scripts have proven to be much less fragile than their *ccp_test* counterparts, and are also easier to write; hence, there's a greater motivation to write them.

However, many of the tests counted in the *ccp_test* column involved detailed inspection of the subsystem debugging log; as this contributed to fragility, relatively few of the Scripting Engine tests are of this kind. If this kind of testing were added to the existing scripts, the total number of Scripting Engine tests would be much higher

The shortest useful Scripting Engine test script is one line of code, performing one test. The shortest useful *ccp_test* script is ~7 lines of code, performing one test.

6. CONCLUSIONS

Regarding internal vs. external testing, the lesson is that simplicity pays.

Our external test harness, *ccp_test*, was capable of testing the entire system including invocation, normal operations, and failure recovery after unexpected task halts. However, it was difficult to get started with, required lots of boilerplate code in each test case, and was extremely fragile because the successful execution of each test depended on many factors external to the feature being tested.

Our internal test harness, the Uplink Scripting Engine, is less ambitious than *ccp_test* but is also considerably simpler to use. It assumes that the software is installed correctly and has already been invoked, thus eliminating a major source of test fragility. It is available at all times, to every developer and tester, and is monitored and controlled using the same interface as the subsystem itself. And although there are test cases it cannot handle, it is nevertheless testing far more test cases than its

predecessor. As an added bonus, it can be used during normal subsytem operations to automate "operator work-arounds".

Regarding the use of Tcl as the scripting language, Tcl proved to be a mature, solid tool. I've frequently left my unit test suite running repeatedly over night or over a weekend to flush out obscure timing-related bugs; and in no case did I see any problems related to Tcl or its libraries. And even though the script invocation and management model is unusual, Tcl's slave interpreter mechanism supplied everything I needed.

Work on the Uplink Scripting Engine is nearly complete. The only planned enhancement is to generalize it for use by other DSN subsystems currently in development. Also, as more developers and testers begin to use it, I expect that there will be convenience enhancements to the Uplink Scripting Language.

7. ACKNOWLEDGEMENTS

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