

Improving Flight Software Module Validation Efforts: A Modular, Extendable Testbed Software Framework

R. Connor Lange
Cal Poly Computer Science Student
rclange@calpoly.edu
Mentors: Tom Fouser, Cindy Oda

August 25, 2012

Abstract

Ever since Explorer-1, the United States' first Earth satellite, was developed and launched in 1958, JPL has developed many more spacecraft, including landers and orbiters. While these spacecraft vary greatly in their missions, capabilities, and destination, they all have something in common. All of the components of these spacecraft had to be comprehensively tested. While thorough testing is important to mitigate risk, it is also a very expensive and time consuming process. Thankfully, since virtually all of the software testing procedures for SMAP are computer controlled, these procedures can be automated. Most people testing SMAP flight software (FSW) would only need to write tests that exercise specific requirements and then check the filtered results to verify everything occurred as planned. This gives developers the ability to automatically launch tests on the test bed, distill the resulting logs into only the important information, generate validation documentation, and then deliver the documentation to management. With many of the steps in FSW testing automated, developers can use their limited time more effectively and can validate SMAP FSW modules quicker and test them more rigorously. As a result of the various benefits of automating much of the testing process, management is considering this automated tools use in future FSW validation efforts.

1 Introduction

The Soil Moisture, Active and Passive (SMAP) project aims to gather data about the freeze/thaw state and soil moisture levels of the Earth's surface [4]. The flight software internal testing (FIT) team's role in the project is to develop and test flight software. The amount of testing that occurs to validate a portion of the flight software is a non-trivial, multi-faceted process which includes the use of automated scripts and other testing tools. The automated tools used for development are often intertwined with the tools used for testing. Each individual tool is typically combined with various other tools and wrapped into a single executable tool. These combined tools provide high level functionality for specific situations, such as testing the current build of flight software (which includes building the project, checking the code for problems such as unbounded loops, and running test cases). These types of high level tools provide infrastructure to software developers and testers, which ultimately makes their jobs easier.

When any sufficiently large software development project takes place, there is complexity in the software development infrastructure. Various people are building and testing their code and many directories exist for storing certain documentation. As the project grows so does the number of previous builds and the archive of documentation. In order to keep everything straight, software infrastructure tools can be deployed to consolidate summary data for managers and automate testing processes. These tools ultimately save the developers time and make the development team more efficient.

2 Background

When flight software (FSW) is being developed, it is also being tested to ensure that the code meets the requirements of the project. To facilitate testing, the SMAP project has testbeds in two different buildings (Blg. 198 and Blg. 156) that developers can reserve time on and access remotely. However, because not everyone can run tests on the testbed at once, software that simulates the avionics is often used instead. On the SMAP project, the simulation software is called the Workstation Test Set (WSTS). WSTS is vxSim-based and simulates the hardware in the testbed. Anything that is supposed to occur in hardware will run slower in WSTS, but since WSTS is running on computer with a much faster processor than SMAP's RAD750, the software executes faster than in the testbed[3]. Because WSTS doesn't rely on the spacecraft hardware, every developer can run WSTS on their own machine, reducing contention for testbed time and ultimately making the developers more efficient.

Because it can be spawned on local machines and doesn't risk damaging the hardware, WSTS has been an ideal platform for automated testing. WSTS has enabled automatic overnight regression tests and other testing suites to be developed and deployed as part of the SMAP FSW development effort. While having an automated testing suite is useful to developers, the WSTS regression suite operates at a lower fidelity and at a slower speed than a regression suite would operate on the testbed. Further, the WSTS regression suite results come from a simulator, which are only as accurate as the simulator itself. WSTS is not sufficient to ensure that the software will function on the flight hardware because it is very likely that there are small differences with what WSTS is simulating and how the hardware will actually behave. Running flight software in the testbed is also the only way to execute the code in real-time.

Unfortunately, a regression suite that operates on the testbed wasn't available and needed to be developed. Due to the existing WSTS regression script's tight coupling with other libraries, lack of modularity, and the team's desire for a flexible system, a completely new regression script had to be developed from scratch. Although the regression code needed to be written from scratch, many existing processes such as Chill and CuteCom could be leveraged to send commands and configure the spacecraft in the testbed. Additionally, existing scripts and wrappers could be utilized to send SSE and flight hardware (FHW) commands messing with the low-level data streams on the testbed equipment via sockets could be avoided.

3 System Details

To maximize system flexibility and code re-use, the Testbed Regression Suite was developed as a modular system. With the constantly changing testing needs of the FIT team,

creating an easily extensible system was a top priority. By dividing the functionality of the Testbed Regression Suite into multiple modules, the system can be easily reconfigured to support different testing efforts. Because of the loose coupling between the modules, it is also possible for a subset of the modules to be used for applications very different from testing; any future project that requires an API for Chill or CuteCom can use the modules to easily develop a new system.

There are several modules provided as part of the Testbed Regression Suite and each represents a part of the regression suite. Although the modules are isolated from one another, modules are organized in a hierarchy where low-level functionality provided by modules such as SSE and CuteCom are ultimately utilized by high-level modules such as the Test Controller or Testbed198. A diagram of the system architecture is shown in Figure 1.

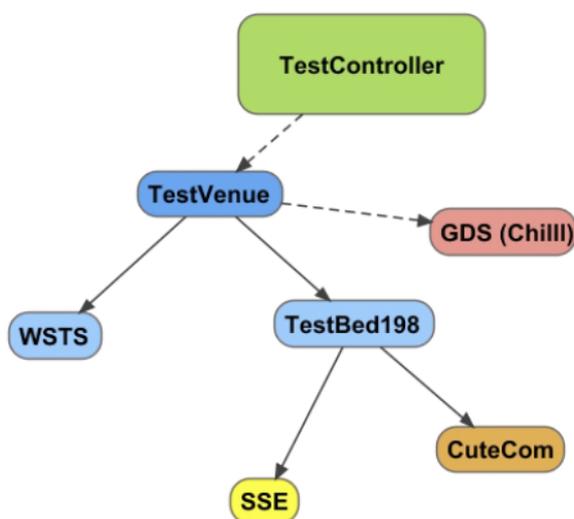


Figure 1: The Regression Suite System Architecture

3.1 CuteCom

CuteCom [2] is an open-source serial device terminal program that is currently used to talk to the hardware in the 198 SMAP testbed. CuteCom is used to load, initialize, and run flight software. Since flight software needs to be restarted for each test in the regression suite and CuteCom has a GUI frontend, the CuteCom module makes use of *sendcc* to send individual CuteCom commands in a headless environment. In addition to sending serial commands, *sendcc* provides the output of the command executed. The returned output can then be analyzed to determine if the command succeeded. For example, when sending the "enet_init" command the expected output in the 198 Testbed is:

```

Loading lnc...initString=null
Loading lnc...initString=1:-1:-1:-1:0x0:0x0:0x3100
unit 1 IOBase 0xc0440300 Ivec 0x1c Ilevel 0x0 Offset 0

```

By verifying that the output of commands matches the expected output, the flight software will always be in the correct state and any test failures won't result from the software loading procedure. If any line in the output stream doesn't match one of the expected conditions, the system will detect that FSW failed to initialize and take corrective action. By default, the FSW initialization sequence will be sent one more time, with the system terminating on an error.

3.2 SSE

Just as the CuteCom module provides an interface for CuteCom to interact with flight software, the Simulation Support Equipment (SSE) module provides an interface to the SSE. This interface consists entirely of turning on and off the SSE. The SSE provides extra feedback to developers and facilitates the testing of flight software. The SSE, like flight software, has a specific turn on procedure that must be followed to avoid problems. However, the SSE turn on procedure has a slight chance of causing damage to the flight hardware if the turn on/off procedure is executed incorrectly. Although an incorrectly executed turn on/off procedure is unlikely to damage the hardware, it will place the system in an unknown state which cause problems when determining the expected results of the system. However unlikely it is that the SSE procedures could damage the hardware, the benefit of having a completely automated testbed versus an almost completely automated testbed (a difference in execution time on the order of seconds) wasn't worth the risk of damaging extremely expensive flight hardware. It is worth noting that the dangers of the SSE module are not a result of the SSE module itself, but rather the entire regression suite crashing during execution. As currently implemented, a regression suite crash would not send the SSE shutdown commands, thus leaving the SSE in an unknown, and potentially dangerous state.

3.3 Chill

In order to get feedback from the spacecraft hardware in the testbed, a module needed to be developed that encapsulated all of the uplink and downlink functionality between the spacecraft and the ground systems. Thankfully, this module (named Chill) was pre-existing so only a wrapper needed to be developed. Release 2 (R2) Chill consists of two programs, *chill_down* and *chill_up* (for downlink and uplink respectively). When these two programs are run interactively and launched via *chill*, their Graphical User Interfaces (GUIs) are combined into an uplink/downlink window and numerous windows appear to allow the user to configure the session and send flight software commands. While GUIs are excellent for human users, they aren't very useful to automated scripts. Therefore, the processes are run headless so they don't need to be configured and nothing needs to be clicked.

Since Chill utilizes different ports, hostnames, and SSE information depending on the testbed that it is running in, the Chill module is initialized with a dictionary containing all the relevant configuration information to maximize flexibility. If any values in the dictionary aren't specified, they are set to the defaults of a particular venue. These defaults are sometimes the correct values, but often default to incorrect values. For this reason, each venue should provide a complete dictionary when initializing Chill.

3.4 TestVenue

Unlike all other parts of the system, the `TestVenue` class is merely an interface (via abstract methods) that should be implemented by any test venue such as WSTS or the 198 Testbed. The methods that must be overridden by subclasses are:

- `start` - one-time initialization code
- `prepareForTest` - multiple initialization code (usually used to ensure separate logs)
- `runTest` - the only non-abstract method, runs *fitfunc* by default
- `finishTest` - shuts down anything spawned by the `prepareForTest` method
- `stop` - one-time (final) shutdown code
- `configure` - configures the testbed (setting instance variables, debug state, etc.)

3.5 Testbed198

The `Testbed198` class essentially takes the 198 Testbed Turn On/Off Procedure [1] and automates it using software. Namely, the `TestBed198` class overrides all the abstract methods from the `TestVenue` class to execute the procedure and run tests on the testbed. When the the Testbed Regression Suite runs, the testbed's `configure` method is called first to assign port numbers and host names that will be used in the procedure. Once the Testbed Regression Suite is ready to initialize the testbed, the `start` method is executed, synchronizing timers and powering on the testbed. Shortly after `start` completes, the testing loop begins. Within this loop, the `prepareForTest`, `runTest`, and `finishTest` methods are called. Testbed utilities such as the Ground Data System (GDS) are spawned at the beginning of the loop, used during testing in the `runTest` phase, and then shut down during the end of the loop. As configured, this loop executes once per test script. Specifically, the testing loop encapsulates the following sequence of events:

1. Start CuteCom
2. Start Chill/GDS
3. Send a hardware sytem reset (`HDW_SYSTEM_RESET` cmd)
4. Verify testbed configuration (check boot bank, make sure all commands went through)
5. load, initialize and run FSW
6. execute a regression test
7. Shutdown Chill
8. Shutdown CuteCom

When the final iteration of the loop completes after executing the last test, the `stop` method is executed to shutdown the testbed. This method powers off the testbed and shutdowns the SSE components of the system. Once the final shutdown commands are executed, and the testbed has been cleaned up for the next user, the `Testbed198` class' role in the Testbed Regression Suite is finished.

3.6 Test Controller

The Test Controller class is the top-level module in the Testbed Regression Suite and is the only part of the system that is specifically designed for a regression test suite. As the top-level module, the Test Controller is responsible for handling command line arguments, configuring the regression testing process, and controlling the testbed via its interface methods. It is also responsible for setting any necessary environment variables used during the testing process. When the Testbed Regression Suite begins, the Test Controller first parses the command line arguments, which consist of a build folder and a test file. The build folder contains everything that is needed to run regression tests, including a flight software object file and a XML command directory (used for sending commands in Chill). The test file contains a list of the module to be tested, the host to run the test on, arguments that should be passed to the test, and a number of other parameters.

In order to execute regression tests, the Test Controller executes the following procedure:

1. Get the test file name and build directory from the command arguments
2. Parse the test file and consolidate dictionaries representing each test into a list
3. Initialize the assigned TestVenue subclass and perform the one-time startup
4. Execute the TestVenue subclass to prepare for test execution
5. Get a test dictionary out of the list and run it via *fitfunc*
6. Shutdown the parts of the testbed that are started for each test
7. Shutdown the testbed
8. Optionally, post-process the test session using *smapp*

Because all testbeds should derive from the TestVenue baseclass, the Test Controller class is test venue agnostic; no changes must be made to the Test Controller (except ensuring the desired testbed is set) to execute regression tests at a different venue. This property makes the Testbed Regression Suite very adaptable because the Test Controller class isn't coupled with CuteCom, SSE, or Chill. The biggest benefit of this type of system design is the significant amount of flexibility it provides. If a unique type of test venue is developed in the future, the Testbed Regression Suite will work on that testbed, provided a testbed class is developed. If the tester wants to run something other than a regression test on the testbed, the Test Controller class can simply be removed and replaced with a different class that performs the desired tests. Because the Testbed Regression Suite is so flexible, it can be used on all future testing projects and decrease the time required to manually bring up the test venue and run tests.

4 Conclusion

Through the completion of this work, the the Testbed Regression Suite contributes both API wrappers for common testbed operations and a complete regression suite. Both of these results improve developer efficiency by decreasing the amount of time that needs

to be spent on testing. Additionally, as more non-flight projects move towards a modular/component architecture, the code developed as part of this project will be increasingly valuable. In the future, all of the code that is part of the Testbed Regression Suite could be integrated into a much larger system that, as an example, builds the flight software, runs a regression test on the build, and performs some type of analysis at the end. The possibilities are truly endless and it is the sincere hope of the author that the Testbed Regression Suite is utilized by future developers and contributes to the automation efforts at JPL.

5 Future Work

Although the Testbed Regression Suite is mostly complete, there are a lot of aspects of the system that can be improved. First, the various outputs and logs can be consolidated into one area so the results of the Testbed Regression Suite are much easier to view. The logs and output that would need to be consolidated result from CuteCom, chill_down, chill_up, the Testbed Regression Suite itself, and various other places. Second, the error checking the system does to ensure everything is okay could be more robust. Each part of the system (Chill, CuteCom, etc.) is so complex that there simply wasn't time to implement extensive error checking. Lastly, the timing of the testbed classes can be improved. Many of the wait times after commands were determined by varying the timing values until a task worked consistently. Since these wait times are fixed, they have the potential to add a lot of overhead to the system. In addition to the future improvements previously mentioned, there is always room to improve a system or make it even more flexible.

6 Acknowledgements

None of this work would have been possible without funding from NASA/JPL, or the guidance provided by my mentors Tom Fouser and Cindy Oda. This work was completed as part of the JPL Summer Internship Program.

References

- [1] MONTANEZ, L. Smap testbed power on/off procedure. JPL Document 72553, 2012.
- [2] NEUNDORF, A. Cutecom on sourceforge. <http://sourceforge.net/projects/cutecom/>, 2011.
- [3] SOHL, G. Wsts-3.1 user's guide. <https://charlie-lib.jpl.nasa.gov/docushare/dsweb/Get/Document-1466162/WSTS-3.1%20Users%20Guide%20.docx>, 2012.
- [4] YUEN, K. Smap project webpage. <http://smap.jpl.nasa.gov>, 2012.

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the JPL Summer Internship Program (JPLSIP) and the National Aeronautics and Space Administration.