Spaceport Command and Control System Automated Verification Software Development

Michael Backus
NASA Kennedy Space Center
Major: Computer Science
Spring Session
Date: 06 04 2017

Spaceport Command and Control System Automated Verification Software Development

Michael W. Backus[1]
Fayetteville State University, Fayetteville, North Carolina, 28301

For as long as we have walked the Earth, humans have always been explorers. We have visited our nearest celestial body and sent Voyager 1 beyond our solar system[1] out into interstellar space. Now it is finally time for us to step beyond our home and onto another planet. The Spaceport Command and Control System (SCCS) is being developed along with the Space Launch System (SLS) to take us on a journey further than ever attempted. Within SCCS are separate subsystems and system level software, each of which have to be tested and verified. Testing is a long and tedious process, so automating it will be much more efficient and also helps to remove the possibility of human error from mission operations. I was part of a team of interns and full-time engineers who automated tests for the requirements on SCCS, and with that was able to help verify that the software systems are performing as expected.

Nomenclature

ACK	=	Acknowledge Response
CLI	=	Command Line Interface (text editor)
CSV	=	Comma Separated Values
DA	=	Development Activity
IDE	=	Integrated Development Environment
KSC	=	NASA Kennedy Space Center
LCC	=	Launch Control Center
LCS	=	Launch Control System
MPEG	=	Motion Picture Experts Group (video file type)
NACK	=	Negative Acknowledged Response
NASA	=	National Aeronautics and Space Administration
NE-XS	=	NASA Engineering – Electrical/Software (branch)
PLC	=	Programmable Logic Controller
PNG	=	Portable Network Graphics (image file type)
SCCS	=	Spaceport Command and Control System
SLS	=	Space Launch System
OCR	=	Optical Character Recognition

I. Introduction

The Space Launch System is NASA's latest launch vehicle, following the successes of the Saturn V (1967-1973) during the Apollo program and the Space Shuttle (1981-2011) for the STS program. With the next generation of launch vehicle comes a continuation of everything NASA has learned over the years, along with the latest in technology and requirements to send humans to Mars and to bring them back home safely. Achieving this goal requires merging several disciplines and technologies on a much larger scale than previous programs. More complexity in the systems comes with a greater risk of something going wrong, so more resources must be poured into testing and verification of the software on multiple levels. Unit tests are written for the software but often times the best way to test a program is to give it a "test drive" similar to what an actual end user may do with it. A lot of effort is spent on this task since it is the most important for safety- every command sent to the launch vehicle must be received and processed properly for everything to go right.

The task of running the Launch Control System (LCS) software through its paces requires the arduous task of going through a series of test steps, sending commands, checking feedback and limit controls, opening up new displays, and so on. While easy to perform, most of the steps can be quite repetitive and monotonous, which in turn

[1]Software Development Intern, NE-XS, NASA KSC, Fayetteville State University

can lead to an even higher chance of human error. By utilizing open source and proprietary software tools, we automated the process of testing SCCS. We used a testing framework that uses a keyword-focused coding approach to automation. Other software we used included an automation test library which was integrated with our testing framework by a previous group of interns. The automated testing framework works over image tracking and the test libraries, which utilize code that was implemented in Python. Other software included computer vision implementations and other optical recognition tools. Additional functions can be developed in Python or from within the testing framework, depending on the development requirements. These were also known as keywords, when developed from within our automation framework in its own, proprietary language.

II. Objectives

The primary focus on this project was to automate test cases for the launch control software. A DA, a piece of paperwork representing a unit of work, was assigned to us through a commercially available workflow tracking tool. To complete this task, we were given access to the test procedures created by developers that operators would normally use in verifying the software by hand. A test procedure consists of a list of numerous test steps that are to be performed in exact order, along with expected responses. In *Figure 1* is a mock-up of a section of test steps. On the left are a set of tasks. An example would be to load up a display command window, change values, and send a command. To the right of each test procedure step are expected outcomes, such as a display changing the value shown or an ACK response from a sent command. Commands are sent to simulated PLCs or servers which are linked to an end item such as a launch vehicle.

Step #	Operator Action	Expected Response
71	On the flight display, send the digital command.	Command has been received by the end item.
72	Reset the value from step 48 by selecting the bypass button. Send the command.	Verify that the values in display GUI 120323112 are changing.

Figure 1. Example (Pseudo) Test Steps. *These are steps that mimic what the test procedures look like. They are not based on actual commands or buttons. (Author owned image)*

Another objective was to create higher level keywords with our testing framework and libraries for the purposes of encapsulating some of the repetitive tasks even further. That is because some steps are in similar groups of three or four that are repeated several times. To do this, we use keywords created with our framework to enlist automation functions to do the tasks we want. Keywords created by our team were stored in several "common" files. As we went through test steps, we continually created keywords to solve specific problems related to those steps. While many steps were on the simpler side, some steps were more difficult, such as trying to see if a display was changing or a needing to find out how many times a specific piece of text shows up on a display. One major task involved a way to launch automation scripts on separate computers remotely, dealing with passwords and a controller written in Python for managing and directing these tasks. One of my tasks included working with a virtual framebuffer to run the LCS software as remote, headless processes and use these as separate testing platforms.

III. Methods

The automation team was split up into three subgroups, which were command verification, measurement processing, and command tracking. Each of these groups deal with the same overall LCS software, but have their own windows and tasks to complete. Each smaller team was led by a full-time engineer and two shadowing interns. I was on the command tracking team which included doing a set of test steps, then going back and verifying that they were tracked properly. We wrote scripts, created custom keywords, and solved specific challenges related to our work. Later we decided that it was more important to get one of the major groups done first, so we all migrated to the command verification portion of the project.

A. Development Environment and Familiarization

Development initially began on our Linux development workstations. Using a software management and version control tool, we were able to install and configure the LCS GUI in question on our machines. Once we started up that GUI we could begin running test scripts with our automation testing framework. However, we soon realized that there were many limitations in working on our development workstations and not on the actual operations set, so it was better to simply work in the Launch Control Center's Firing Room. Within the Firing Room are special workstations which allow full access to the LCS software. They run a Unix flavored operating system as well and are set up to eventually support actual rocket launches. This way we could run the different servers, arbiters, and models required to test the software.

The Firing Room contains many rows of identical workstation systems which are divided into the development set while the others are on the LCS set. The LCS workstations are highly sought after for test engineers because they have the latest released software versions on them. They represent what the actual, final software will look and behave like. The development set has an older version on them and also requires more manual work to get going, including loading models and starting processes. Scripts developed on the development set may produce errors once tested on the LCS set. In addition to this, the development set also has slightly different fonts and display resolutions than the LCS set, which complicates the process. Despite the drawbacks, I spent as much time as possible developing on a development set, since getting an effective amount of time on an LCS set was much more difficult. I knew that in the end I would have to tweak anywhere between 5% to 20% of my finished scripts once I finalized the test procedures on an LCS set, but the tradeoff was worth it since I could typically get time on a development set for a full workday without any issues. It's also important to note that most of the edits were that of font changes or creating screenshots.

B. Writing & Running Scripts

Writing a script for the testing framework is rather straightforward. Files can be named in such a way as to allow for them to run in a sequence, such as 01.filename, 02.filename, and so on. For our work in automation, we matched file names to the test procedure steps, which made them easier for us to find. It is worth noting that these files can also be saved in multiple folders downstream, and our software will run these files recursively according to folder names. Several test steps can also be combined just as long as they're named something easy for us to read, such as 12-15.filename. Each script includes a space for settings, documentation, variables, and test cases. Note that the automation framework considers test cases keywords, while we consider test cases to be an entire set of test procedures. Scripts can be written in the automation IDE, gedit, or CLI editors such as vim, or emacs. Documentation and settings for each automation script usually includes a description of the test step being performed, along with required paths and dependencies set.

```
*** Settings ***
#author: Michael Backus
Documentation
...   | On the "Simulator" display click on the image "Start_Button."
...   | Execute the command.
...   | Response: Check that the values in each display are changing.

Library           MainOS
Resource          ${CURDIR}/../common/libraries

*** Variables ***
@{DISPLAYS}=   User_Interface_01    User_Interface_02

*** Test Cases ***
Perform actions for DISPLAYS in List
        :FOR    ${item}    IN     @{DISPLAYS}
                [ITERATE Actions For Each Display...]
```

Figure 2. Example (Pseudo) Automation Script. *This is a mock example of what an automation script could look like for testing SCCS. Names of displays or commands have been modified for security reasons. (Author owned image)*

Under the test cases block is where we actually write our scripts. Depending on the complexity of the step, this part can comprise anywhere between one to dozens of lines of code. In *Figure 2*, a list of two displays are iterated through and a block of keywords are executed per each. For this example, it is pretty simple: simply click on a start data button then click on the send button. After this, check if a display is changing or not by not finding the default value, which was a number. In this case, we have used an image. If all keywords are completed, then a test case passes, but in this case, we use two *if* statements to pass or fail the test case based on if an image is present.

Each test step listed in the test procedures must include some sort of verification check. Expected responses could be as simple as putting into focus a window that was expected to launch after a command was sent, to confirmation that a display was changing, to confirm a value was set on another display, and so on. Other expected responses my prompt for some kind of pulling of data, such as recording a value within a display.

C. Creating Keywords

Keywords are analogous to functions or methods and are easy to create. We simply start by either looking at the keywords already in existence or by referencing source files within the testing library. Most keywords can be created by simply combining other keyword blocks into scripts that simplify steps into fewer lines. For example, let's say that you wanted to load up a history window (or any dropdown menu) instead of having to click on the menu, create an image with the custom image tool based on some text, click on it, then remove the created temporary images. Instead, we can create a keyword, named something like "Click on Menu ${item}" where ${item} would be a string of text to look for. This keyword would comprise all the steps listed above encapsulated into one line of code but with the added bonus of being able to pass an argument to the keyword.

This custom image tool is a simple Java program that was created by a previous intern. It simply creates four small image files with text that has been defined. After this, an automation script is called to search for a part of the screen that matches the small screenshot that was created. This is one way to bypass setting up any kind of optical character recognition but has the side effect of creating a lot of image files. Fortunately, a keyword existed to clean up after the custom image tool runs which removes these temporary image files. Another method of grabbing images from a set of text or images is to take a region screenshot with a free program such as ksnapshot. You can then search and hover, click on, or do whatever else you wish to do with them.

D. Command Tracking

Command tracking collects data while SCCS is running. Test procedures for this part of software verification were oftentimes very similar to both commanding and measurement processing, with the added step of verifying the data collected. Once loaded, there are several steps in getting the browser set up for retrieving data, which includes setting time and date limits, the set on which the commands were issued, and workstation type. Once this is done and a configuration type file has been selected, we can then view the tracked data. We able to develop a single keyword within the automation framework that consisted of only a few arguments that got us to this step. From here we were able to use and create keywords to help assist in finding and tallying up all of the actions listed that were performed in the specified time of the LCS session.

E. Extending the Toolset

My part in extending the toolset involved working with a virtual framebuffer to launch processes that could be used in running the LCS software, effectively simulating another workstation locally. This was just one part of a solution to a major problem that arose once we found out that many of the test procedures involve some sort of parallel requirements. Because the development workstations in the LCC are tied to the same server, or set running the software, testing requires that the set and all of their identifiers are in the same state regardless of which user performs actions. Even the instance of users on different workstations sending a conflicting command at the same time must be verified. I composed a shell script that would launch a virtual framebuffer along with setting the resolution and displays of that server. Further commands involved launching software and being able to record the screens since these are launched as processes. For this I used another open source program which allowed recording of this virtual server as an MPEG-4 file, or MP4, and fully featured options for video compression/encoder, framerates, resolution, and so forth. This was very important in seeing the feedback in what was actually done on this server. I then coordinated with Susan Pemble, one of our full-time software engineers, to incorporate and test this into our automation framework on the development and LCS set.

Another intern, Andrew Hwang, worked on a script to manage logins, making automating test procedures that needed to run in parallel much easier for us. Tom Plano worked on the controller for directing the tasks of launching several automated scripts on different machines, whether they were physical boxes, headless systems, or simply processes running in the background. Meriel Stein worked on an Excel macro that allowed us to avoid duplicate work assignments and help to clean up our management tool. Together we each developed our own part of solutions to solve the challenges faced on this project. We participated in peer review for each other's code prior to final delivery.

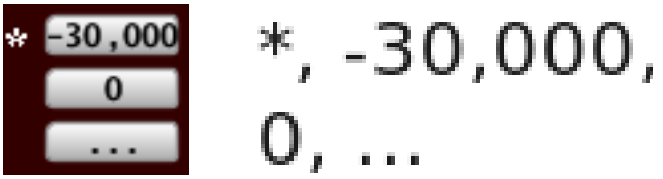

Figure 3. OCR Example. *On the left, a small area of an LCS display. Note the buttons and their values. The asterisk symbol is not a button, but would still be recorded. On the right, the resulting OCR values. OCR can take an image and extract their alphanumeric values into a data structure such as a hash. Values such as location in screen space can also be stored alongside the recorded value.*

F. Future Enhancements

This internship's primary focus was to get as many test procedures done as possible while continuing to develop the automated testing keyword and tool library. An example is shown in Figure 3. Open source optical character recognition software that utilizes machine learning from a data set of images on the screen would be very useful. Images can still be created, but this process is faster because it can differentiate even if the background or text changes to another color. The recorded text and numerical values along with their location on the screen are then stored in a file that can later be accessed. Implementing such a feature could open up our automation framework to a much broader use across other software projects at Kennedy Space Center

IV. Results

We have succeeded in automating test cases and procedures for SCCS and the LCS system. Once fully completed by future teams of interns, this will complete the software integration testing of SCCS and LCS. We were able to run our scripts on the actual software and hardware that is going to be used to launch NASA's Space Launch System, the next generation of rockets destined to go to Mars. There is still going to be more work involved in completely automating all of the procedures, but through our work we have shown that it is both possible and more efficient to do so.

While there are some parts of the test procedures that are difficult to automate, this can and should be done fully to help increase the efficiency of the space program. Making software verification quicker means more funds can go to help cover other costs. While the Saturn V was the largest rocket that ever achieved operational status and the Space Shuttle was the most complex vehicle NASA ever built, Block Two of the SLS is larger than the Saturn V and the vehicle that it's carrying- the Orion capsule - is much more complex than the Apollo capsule. More complexity equals greater risk and things that could go wrong, so it is imperative and critical that we have as many layers as needed to ensure the software works properly.

Acknowledgements

I would like to thank all of those who worked with me during my time working in the automation group. Jason Kapusta, our lead software architect, was very helpful in guiding and brainstorming ideas during our weekly meetings. Terri Waldron, our project manager for automated testing, did a great job with keeping us on track for finished results while at the same time understanding the inconsistency with set time. Kyle Besser, while no longer a member of the automation team, assisted with various issues that arose from our automation test framework and programming languages. I would like to thank Susan Pemble for being a lead on our team who always came up with more efficient methods of doing our work. These were my daily technical contacts who, along with Roger Zoerner and Mark Poff, made up the backbone of the automation team and development support in the firing room of the LCC.

I wish to thank the new interns Tom Plano, Meriel Stein, Abraham Glasser, and returning and yearlong interns Andrew Hwang and Mark Rodriguez for their excellent coding contributions and hard work. I want to thank Jamie Szafran who eagerly offered advice and tips whenever I asked and for always being funny yet serious in keeping us focused. I would like to especially thank my mentor Caylyne Shelton who was always available to sort out any technical issues with the LCS GUI and always provided great life advice. I would also like to thank our Branch Chief Oscar Brooks for hosting the interns right beside his office and for him displaying his remarkable leadership and experience for management in getting everything set up for us.

I am honored that I had the opportunity to contribute a part to the future mission successes at KSC by automating software testing and verification for SCCS. Automation of test procedures will continue onward after our group leaves and other projects may follow from what we have learned. The journey to Mars is composed of a multitude of smaller steps, each of which contain their own problems to solve and hurdles to overcome. As NASA comes closer to completing its most challenging goal yet, Mars appears more and more within reach. And automating test procedures in order to ensure code integrity helps bring us that much closer to humanity's next giant leap forward into the heavens.

References

[1] Barnes, Brooks, "In Breathtaking First, NASA's Voyager 1 Exits the Solar System." *http://www.nytimes.com/2013/09/13/science/in-a-breathtaking-first-nasa-craft-exits-the-solar-system.html?_r=0*," 12 Sept 2013. Web. 01 Nov 2016