

Irreducible Tests for Space Mission Sequencing Software

Lisa Ferguson¹

NASA Jet Propulsion Lab, Pasadena, CA, 91125

I. Introduction

As missions extend further into space, the modeling and simulation of their every action and instruction becomes critical. The greater the distance between Earth and the spacecraft, the smaller the window for communication becomes. Therefore, through modeling and simulating the planned operations, the most efficient sequence of commands can be sent to the spacecraft. The Space Mission Sequencing Software is being developed as the next generation of sequencing software to ensure the most efficient communication to interplanetary and deep space mission spacecraft. Aside from efficiency, the software also checks to make sure that communication during a specified time is even possible, meaning that there is not a planet or moon preventing reception of a signal from Earth or that two opposing commands are being given simultaneously. In this way, the software not only models the proposed instructions to the spacecraft, but also validates the commands as well.

To ensure that all spacecraft communications are sequenced properly, a timeline is used to structure the data. The created timelines are immutable and once data is assigned to a timeline, it shall never be deleted nor renamed. This is to prevent the need for storing and filing the timelines for use by other programs. Several types of timelines can be created to accommodate different types of communications (activities, measurements, commands, states, events). Each of these timeline types requires specific parameters and all have options for additional parameters if needed. With so many combinations of parameters available, the robustness and stability of the software is a necessity. Therefore a baseline must be established to ensure the full functionality of the software and it is here where the irreducible tests come into use.

II. Background

The Space Mission Sequencing Software has been under development for some time. In the past, error checking had been left to a series of critical feature tests and helped establish a functioning baseline. With the newest version of the software, these tests were no longer adequate to test all of the new features. A new set of tests needed to be created in conjunction with the past tests to establish a firmer baseline that could only be established once every critical feature had been tested. These new tests would be irreducible, testing only the most critical features. With these requirements, the irreducible tests were created.

¹ Summer Intern, Modeling & Verification, NASA Jet Propulsion Lab, Pasadena, CA

III. Objective

The creation of the irreducible tests stemmed from a list of features that were deemed to be crucial to the software's baseline functionality. If a single irreducible test fails, the software contains errors that must be repaired before any further software operations are preformed. The irreducible tests aim to become a basis on which the functionality of the software can be determined. While there is currently software implemented to create a baseline, the irreducible tests are updated to the latest version of the software and run in conjunction to the current baseline software. By always establishing functionality before running the requested software commands, efficiency is ensured by catching errors before the software hangs upon them.

IV. Approach

Previously, baseline software had been implemented, but it was outdated and cumbersome. However, this software became the basis upon which the irreducible tests were modeled. Using both the previous tests and the list of irreducible features, the irreducible tests were created with the goal of absolute succinctness and efficiency. With minimum lines of code the creation, checking, and deletion of timelines is achieved. To expedite the checking of the metadata, present in all timelines, or of data that can be inserted into timelines, generic scripts were written to be called and preform the checks. This allows for easy error checking and makes the individual tests easy to read because only the creation and deletion of the timelines (along with data insertion and reading if necessary) are dealt with in the actual test script. The actual checking of the (meta)data is processed outside the script, thus insuring a uniformity among the data because it is all ran through identical checks making any anomalies immediately noticeable.

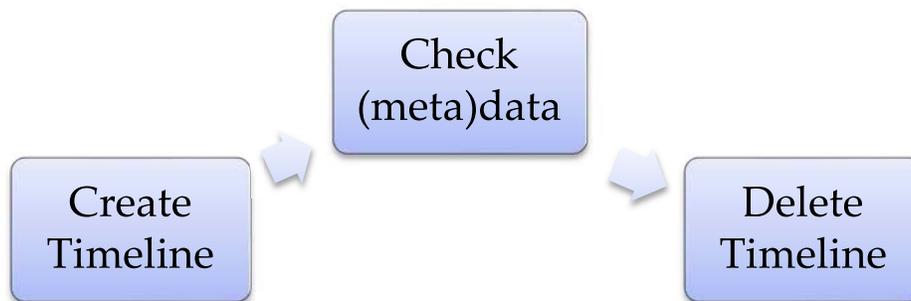


Figure 1: The checking of the metadata and data is preformed outside the script that creates and deletes the timeline.

The timelines themselves are created using URLs to ensure cross-platform functionality; timelines can be created, manipulated and deleted, viewed, and deleted on anything with a browser. To make the URLs immune to the inevitable server migrations and changes in user namespaces, the base URL is defined by global variables. In this way, one file can be updated with the changes for the URL and the changes are perpetuated throughout all of the tests. This makes changes between users and servers nearly effortless and does not require updating the individual irreducible tests. From the base URL, parameters are added to specify the type of timeline being created, the time system desired, along with the value type and time format. The timeline type is the parameter that defines which other parameters are required and which become optional. For all timeline

types a timeline type, time system, and security bit vector are required. For example, the timeline type state requires both a value type and an interpolator type in addition to the standard required parameters. If all the required parameters are met, the optional parameters, unless otherwise defined, are set to their default values.

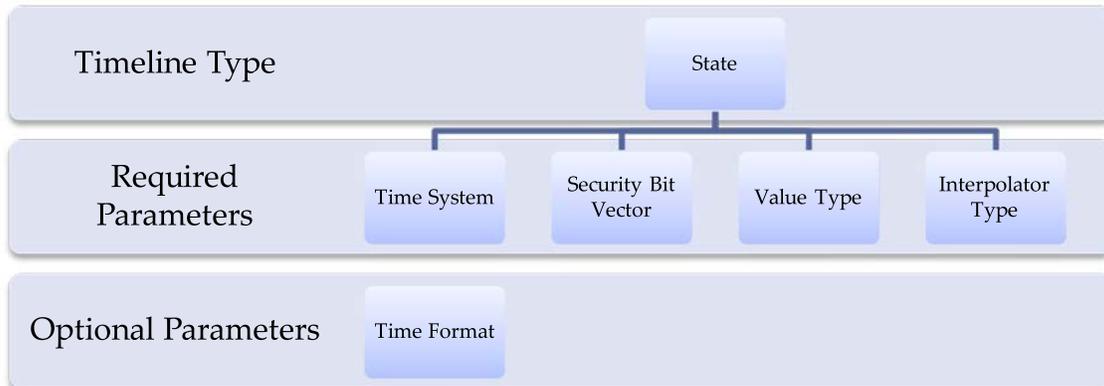


Figure 2: The structure of a timeline with its required and optional parameters.

Data may be inserted into created timelines to simulate previously requested spacecraft operations. Each timeline type can have data inserted into it, although each timeline type requires the data to be formatted differently. The exceptions to this are state and measurement timelines, which share the same data format. Data is stored in HTML files, so that they can be viewed on any browser. For testing purposes, the data is randomly generated using scripts tailored to each data type. The HTML tag `<ins>` is used to denote data that is being inserted, by changing the tag to `` data that was inserted can be deleted leaving the timeline intact. The data files are parsed using a Python function before being inserted into the timelines. While the irreducible tests are written as Bourne Again Shell (BASH) scripts, Python’s HTTP library is much more powerful, therefore the HTTP operations GET, POST, PUT, and DELETE, are handled by Python functions that are called by the BASH script. Using a combination of Python functions and BASH scripts the timelines are created, inserted with data, read, and then deleted.

The base URL and the requested parameters, after the timeline has been created, are exported to a checking script. If no data is being inserted, then only the metadata is checked, else both the metadata and inserted data are checked. These checks consist of determining if the metadata was created and if it was whether the value is either zero or non-zero depending on the specified metadata object. As for timelines containing data, the check makes sure that the data is actually inserted by comparing the system change number (SCN) and looking for a non-zero number, thus indicating a change. The current SCN is used to identify the creation time of a timeline; by setting the SCN parameter in the created timeline URL to “latest” the most current version of that particular timeline is displayed. To simplify the checking scripts, a series of functions were created and then called to preform the actual value comparisons. Functions are called from outside the actual checking script to streamline the debugging process.

After the timeline had been created, data has been inserted, and the metadata and data have been checked, the timeline can be read to the output. Using a Python function and the HTTP GET command, the metadata and any data that has been inserted into the

timeline are read. The output of this command, the HTML code that the newly created timeline URL points to, can then be read in the terminal that the script is run in. This becomes a useful feature if timelines are deleted at the end of the script. Once a timeline is deleted, the URL no longer points to the HTML file containing the metadata and inserted data. If a read is performed before a delete, then the HTML can be viewed in the terminal even if the timeline is deleted or the data itself is deleted.

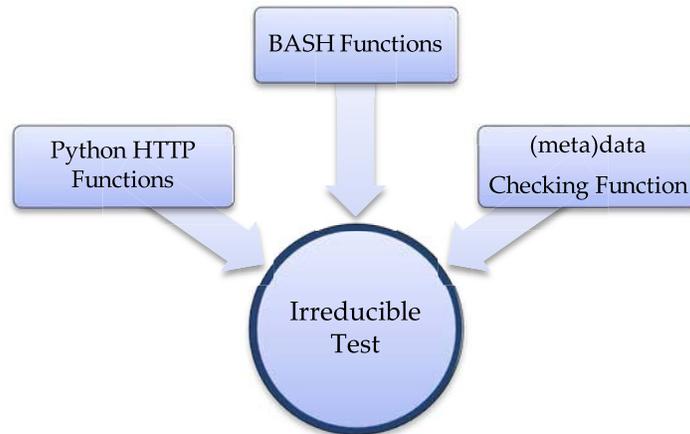


Figure 3: The external scripts and functions accessed by a single irreducible test.

Functions are also used to check the exit codes of every operation performed within a script. By checking for successful exit codes at the end of every operation, errors are found quickly and at their first occurrence in the script. The line number of the function is a parameter of the function itself, thus the failure of the function causes the line number where the error occurred to be sent to the output. This makes finding the source of an error easy to find. With the optional exit parameter in the exit code functions, the script can be terminated at the first sign of an error or if a critical operation fails, thus errors are even easier to discover and diagnose. Aside from checking exit codes, the HTTP return codes are also outputted. This allows for easy viewing of the result of any operation performed in the script. For example, if a timeline creation returns a 303 code, then the create operation was a success, however, if a 400 code is returned, then the create operation was considered a “bad request” and wasn’t created. The variety of HTTP return codes allows for specific troubleshooting to occur at the server level.

The architecture of the irreducible tests is centered on the detection of any possible error and the ease of repair of those errors. The irreducible tests are meant to create a stable baseline for the rest of the software to rely upon; therefore any error present in the elemental functions must be repaired immediately. Through the use of Python function to handle the HTTP, BASH functions to check error codes, data and metadata checking scripts, and HTTP return codes, the irreducible tests are easy to read and make software errors apparent. By providing as much information about a discovered error (eg line number, HTTP return code) as possible, errors can be easily found and repaired.

V. Results

The ultimate goal of the irreducible tests is to help optimize the Space Mission Sequencing Software before its release. By making sure that the most critical features in the software are at their optimal functionality and that there are no undiscovered errors in the software, the Space Mission Sequencing Software will become a powerful tool in the sequencing and verification of spacecraft operations. With the creation of the irreducible tests, previously unknown errors were revealed and remnants from previous software versions were uncovered. Thus the irreducible tests succeeded in discovering and reporting software errors. One goal of the irreducible tests was to test every possible combination of parameters for any timeline type. By doing this, errors were discovered that only occurred with one specific combination of parameters. Had these irreducible tests not been in place, these errors could have been overlooked because of the exacting parameters that were necessary for the errors to arise. The tests were also instrumental in discovering parts of software that were either obsolete remnants of previous versions or had never been upgraded. Many times an error that appeared to have a simple issue (e.g. updating a variable name), revealed much larger issues. Conversely, an extremely complex error would be caused by something as simple as using the wrong time format for a data type. Each error that was discovered could then be reported to the software development team for either removal or repair. The irreducible tests succeeded in discovering previously unknown errors in the software and facilitating the solutions of the errors.

VI. Discussion

While the irreducible tests can be considered a success in the arena of error finding and reporting, the tests were not able to be used to their full extent. The irreducible tests were created to be a tool for both debugging and for testing the scalability of the software. The tests were created and used for debugging, but the scalability was never implemented. Using a program called JMeter, virtual users can be simulated to run the irreducible test. Multiple users can be created to simulate virtual machines to stress the servers under the pressure of simultaneous access. The tests are ready for virtual use and can be used in JMeter any time.

Aside from the virtual stress testing, the irreducible tests performed their debugging role to their full capabilities. While the tests were not exercised to their full extent, their use as a debugging tool became apparent. In this area, the tests proved to test the functionality of the software and provide a stable baseline for the whole.

VII. Conclusion

In this paper, it has been shown that the irreducible tests can be used for fault finding in the Space Mission Sequencing Software. It has presented that the tests can also be used to establish a baseline to keep as many errors as possible from the software. The irreducible tests will continue to be created and updated as the software evolves towards release. For future implementations, the irreducible tests can be used for simulating virtual users and stress testing the software and server. With the irreducible tests, the software now has means of establishing a functioning baseline and a way to test scalability before expanding onto the cloud.

Acknowledgments

I would first like to acknowledge Paul Wolgast, my mentor. All the assistance and understanding he has given me during my internship has helped me better understand my work and enjoy my time at JPL.

I would also like to acknowledge Steven Parkin, my co-mentor. He helped me better understand the programming languages I was using and the general architecture of the Space Mission Sequencing Software.

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

Reference(s)

SEQ Revitalization. Ed. Paul A. Wolgast. SEQR Development Team, 12 May 2012. Web. 15 Aug. 2012. <[https://jplis-ahs-003.jpl.nasa.gov/confluence/display/SEQR/SEQ Revitalization](https://jplis-ahs-003.jpl.nasa.gov/confluence/display/SEQR/SEQ+Revitalization)>.