Lessons Learned Implementing Multi-Mission Sequencing Software

L. Needels[1]

[1]Jet Propulsion Laboratory, 4800 Oak Grove Dr., M/S 301-250D, Pasadena, CA, 91109-8099, USA, Laura.Needels@jpl.nasa.gov

ABSTRACT

All of JPL's deep space missions rely on complex sequencing (ground) software that is responsible for the planning of science and engineering activities, checking command syntax, checking mission and flight rules, and translating the commands into packets which can be uplinked to the spacecraft. The ground software is used by operations teams for preliminary stages of sequence design as well as final stages of sequence development. In previous missions, significant effort was spent validating the software since sequencing errors can cause the spacecraft to enter fault protection or even loss of spacecraft. Over the last decade, in an effort to reduce costs and development time, the sequencing software has been transformed into a multi-mission format.

This paper will describe the software and its uses to provide context for its criticality. The different approaches that have been taken to implement the software in a multi-mission format will be outlined. The advantages and disadvantages as well as the lessons learned during development and maintenance of these different architectures will be discussed. Finally, the use of multi-mission software in operations and the lessons learned from using it will be discussed.

This paper will provide valuable information to organizations exploring the use of multi-mission software, regardless of whether the change is to minimize spacecraft ground software development time or cost reduction. Similarly, the paper will provide insight into some of the steps that can be taken during software development and operational use that will minimize difficulty later.

1. INTRODUCTION

During the era of Faster, Better, Cheaper, in an effort to reduce the costs associated with the planning and sequencing portions of the software used in the uplink process, JPL developed a multi-mission form of planning and sequencing software. The uplink software is developed as two separate components. The multi-mission "core" software provides, in a generic sense, the capability to perform the functions needed to support the uplink process: planning and scheduling events, checking mission and flight rules, monitoring resources, command translation and "packetizing" commands. The "core" software is then "adapted" to a project specific mission. The "adaptation" part of the software task involves providing the models for activities needed for planning and scheduling, converting the Command List for the project into models that can be used for sequence checking, coding project and mission flight rules and the modeling needed to support them, and developing project blocks for repetitive activities. This software has been or is currently being used by Deep Space 1, TOPEX, JASON, Mars Global Surveyor, Mars Odyssey, Genesis, Stardust, Cassini, Spitzer (formerly known as Space InfraRed Telescope Facility), NEAR, Ulysses, Deep Impact, CONTOUR, MESSENGER, DAWN, Mars Exploration Rovers, Phoenix, and Mars Reconnaissance Orbiter.

At JPL, the software used to support the uplink process is known as SEQ, and consists of about a dozen different program sets. It is not necessary for a project to use all of the SEQ programs. Interface specifications for each program are well defined, and if a project chooses to use a non-SEQ program for one or more parts of the system, it can easily be accomplished. Many projects use their own components in some part of the uplink process. However, the SEQ software covers the entire process from science and spacecraft activity planning to radiation. In addition, the SEQ system provides utility software to help build inputs and organize data for reviews.

At the front end, the component of SEQ software used to perform science activity planning is the Science Opportunity Analyzer (SOA). SOA has a very user-friendly interface that allows scientists and other planners to exercise trade studies and preliminary designs for science observations. Search engines are available that will locate times of interest such as fly-bys, bow-shocks, occultations, angles, periapsis and apparent diameter. Once these opportunities have been located, the user can design a specific observation (eg. Continuous Scan, Roll Scan, Mosaic, Point at a Target). Critieria such as the primary target, secondary target, target offsets, duration, primary observer, and secondary observer are entered. Constraint checking for activity duration, distance, exclusion zones, and hardware limits can also be exercised by SOA.

The SEQ component used to do engineering activity planning and resource modeling is Activity Plan

Generator (ApGen). ApGen allows the user, more often a mission planner or a spacecraft engineer, rather than a scientist, to plan activities while monitoring resource constraints. Desired activities might be Deep Space Network (DSN) contacts, science activities, and general engineering activities. The resources monitored might be solid state recorder space, propellant, and battery state of charge. ApGen also contains a scheduler which will allow a user to define activities, some of which are fixed in time (e.g. DSN contacts), and additional desired activities. The scheduler will place activities on the timeline so that resources are not violated. The scheduler will also show which of the desired activities could not be scheduled.

Sequence development and verification is done by Sequence Generator (SeqGen). SeqGen offers 3 distinct capabilities. The first is to help the user generate sequences. A GUI is provided which assists the user in choosing commands and implementing the correct parameters. The second is to merge different sequences to produce a single, master sequence. The third is to provide checking of command syntax, mission rules, and flight rules. At JPL, SeqGen's primary use is for mission and flight rule checking. Historically, the mission and flight rules checked by SeqGen were more timing related, such as "The catbed heaters must be turned on 5 minutes before the thrusters are fired." or "Once the spacecraft has been launched, the mission phase may never be set to pre-launch". SeqGen has also been used for checking commands against spacecraft states, such as "The command may be used only while the spacecraft is nadir pointed." or "The command may not be issued while the spacecraft is in eclipse". However, more recently, SeqGen has also been used to check for resource constraints using models at the command level. An example is using an externally provided power model to monitor the battery state of charge. SeqGen creates a spacecraft sequence file that contains the commands that will be sent to the spacecraft.

Another SEQ program, Spacecraft Language Interpreter and Collector (SLINC) is used to translate the commands into spacecraft readable binaries and converts the sequences into packets. SLINC calls the Command Translation Subsystem (CTS) program to perform the command translation to bits.

Virtual Machine Language (VML) is a relatively new development in SEQ. VML provides more flexibility in commanding by offering such capabilities as waiting for a spacecraft state to occur, rather than an absolute or relative time to occur. VML consists of 3 components. The first is the VML Flight Code (VMLFC) which is a Flight Software component that offers the additional bonus of standardizing the Flight/Ground interface. The second is the Off-Line Virtual Machine (OLVM) which is a simulator which permits rapid checkout of sequences. The third is the VML Compiler which performs the translation needed to generate the binaries that will be uplinked.

Prior to using SeqGen, an adapter can create or merge the SeqGen input spacecraft model information files using a utility program called Sequence Adapter (SeqAdapt). This program is also be used to check model file syntax.

Another utility program is called Sequence Review (SeqReview). SeqReview can be used to manipulate sequence products into a format that is easier to review by stripping out unnecessary data or by reformatting or reordering data.

The Adaptation team for the Core Software also provides tools and scripts to the project that are useful in the processing that occurs. Even though different projects may have different processes, the Adaptation team attempts to keep the scripts and tools identical so that one version of the program exists in all areas. Changes and corrections can then be made to a single version of the script and pushed out to all projects.

2. LESSONS LEARNED DURING MULTI-MISSION SOFTWARE DEVELOPMENT

Looking back over the last decade, with twenty-twenty hindsight, many lessons have been learned about developing multi-mission software. Below are eight lessons that may be useful to other organizations.

2.1. Make plans for handling legacy software

The first, and perhaps easiest, lesson to understand is that early in the design process, the team should develop a strategy for dealing with legacy software. There are three components to this issue. First, mission needs do evolve. Sometimes, new hardware forces a change, or opens up opportunities for new techniques. Second, a project may have a need to do something radically different. Finally, many spacecraft are designed for much longer lifetimes. These changes result in two different considerations with legacy software.

One item for consideration occurs when newer missions no longer need an old piece of software that is required by an operational spacecraft. Often, the newer missions use completely different software for that function. SEQ has chosen to offer recompilations of the old software under newer versions of the Operating System as long as more than one project is using the software. No new changes or bug fixes will be made to the software. When only one project is using the software,

the software is turned over to the project and is no longer supported.

The second issue that should be considered is when a project chooses to no longer accept new versions of the software. The project may, because of reduction in financial resources or workforce shortages, simply choose not to take the new version of software. Another reason might be that a project is in a critical phase and may not want to risk introducing new software. The strategy SEQ has decided to follow is that it does not require users to take new versions of software. However, if the projects need any sort of change, such as a bug fix, a new capability, or just a recompilation for a new version of the Operating System, it must take the new software.

Regardless of the strategy an organization decides to implement in these areas, it should decide early, and make users aware of this position.

2.2. Develop a strategy for when it is best to fork off the software development

The idea behind multi-mission software is that since many missions can use it, it offers the potential for cost savings, rapid development, etc. However, there may be a project that will require changes to the software that are so dramatic, or need to be made on such a rapid schedule, that it is difficult to support those needs in multi-mission software development.

It may be preferable to fork the software into a project specific version. If this is done, there should be a clear understanding at the outset whether the forked software will ever merge back into the main branch. If the software will be merged into the main branch, the responsibility for the multi-mission verification of the changes in the software should be established. In addition, there should also be open channels for communicating changes in the multi-mission version of the software. Both the multi-mission software and the project specific software should ensure that anomalies in the multi-mission part of the software are communicated to the other party.

Forking the software may be an undesirable outcome to both parties. The project may object because it now has to take on additional responsibility in development and validation of the software. Multi-mission software is appealing to projects because much the validation work is done for them. However, if the software is forked, the project may have to redo all of the testing work. Also, the multi-mission software group may object because they may have a new competitor, or they may have to make some major overhauls to the software to merge project specific changes back in. Even with these drawbacks, there may be times when it just does not make sense to maintain a multi-mission version of the software to meet project needs.

2.3. Develop a testing strategy across all users

There are two facets to the testing lesson learned. The first is, where does multi-mission software testing responsibility end? The second facet is that an easy to use set of regression tests that span all uses of the software should be developed and maintained. While this strategy is just good software practice, it becomes more relevant across multi-mission software or scripts because one project team may be unaware of how a different project team uses the software.

At the onset, multi-mission software developers should decide where their responsibility ends when it comes to testing the software. Does the testing responsibility of multi-mission software developers end with unit and subsystem test cases? Or, should the development team have the responsibility of running project data through the software for verification? Clear responsibilities should be established so that projects understand what their role is, and so that all projects are treated equitably.

With respect to the second facet, SEQ has an adaptation team that maintains a nearly multi-mission set of scripts/utilities that are delivered to projects. Since adapters on one project may need to make changes to the toolset, it is highly recommended that an automated testing system is established so an adapter can run through a set of multi-mission tests to ensure that changes don't break something currently working. One project's adapter/developer may not know how another project uses the software, so that adapter/developer will have difficulty ensuring that something is fully tested. Having a test suite available and easy to use when the modifications are being made is preferable to having changes made, and delivered to a project, then finding that the changes negatively impact a different project, causing a redelivery.

2.4. Implement a system for tracking deployments

In SEQ, software deployment is handled by the adaptation teams. Given that SEQ currently supports approximately 10 different projects with the common set of scripts/ utilities, the SEQ adaptation team finds it is extremely valuable to have a tool to track what versions of scripts/utilities each project is currently using. When it is time to make a new delivery to a project, the adapters can execute the version management tool to find out what changes have been made since their last delivery, and decide if they want to include these changes in their delivery. With several

hundred individual scripts/utilities, it is difficult to ensure that all the modifications are picked up by a team. For example, if a change is made to the utilities that affected 10 scripts/utilities, using this tool helps prevent only part of the changes being picked up, and a project receiving incomplete changes.

2.5. Develop a clear statement of use strategies for the software

Many organizations have different standards for development for different types of software. For example, "mission critical" software is very heavily tested, while "personal utility" software may not receive any testing at all. It is important that a multi-mission software project decides on the classification of the software that it is developing and ensure that any project that wants to use the software understands this criteria. There are no issues if a project's uses of the software are for less critical uses than the software is developed against. However, it will help prevent a project from using software for a more critical task than the software has been developed for. A project could then decide if it wants to provide the additional support needed to bring the software up to the level of criticality the project needs, or if the project wants to use other software/methods for the more critical task.

2.6. Evaluate different architecture styles for multi-mission software

Multi-mission software can be developed with different architecture styles. SEQ has used two different methods, and currently does not have an opinion on which is preferable. One style used by SEQ is to develop a core piece of software, and use little languages to adapt it to project needs. The second style is to provide software that has separate threads internally to perform the adaptation to the project.

Developing a core piece of software that gets adapted using little languages provides the advantage that a project does not need to make any changes to the core software to use it. The project can simply use the little language to adapt it for its needs. However, developing this style of software is often more expensive, more complicated, more time consuming, and more difficult to ensure robustness.

Developing software that has separate threads internally to perform the adaptation to the project is often easier, cheaper, and faster to develop. However, every project must modify core software to obtain the functionality it needs. Another ramification of this style may exist when missions have a proprietary need, because this format does not readily support that requirement.

2.7. Evaluate the needs for an architecture that will support shared hardware resources

Several of the JPL projects share computer resources and staffing for sequencing activities. This is a truly multi-mission installation. Most projects want to freeze the software for some length of time up to and through mission critical events. Depending on the number of projects, and the number and duration of mission critical events, it may be extremely difficult to get changes to the software approved. Some of the JPL mission software freezes have lasted several months. Because of the needs for changes during extended freezes, the SEQ Adaptation team has begun to develop ideas for a architecture that will allow for new software to be installed on shared resources during software freezes. While the obvious solution would be to eliminate the shared computer resources, this solution may add significant hardware and maintenance cost. The new architecture has not been studied thoroughly, it will be examined if a new multi-mission installation is planned.

2.8. Be careful of developing "quick and dirty" tools or prototyping software

The last lesson learned, while not specific to developing multi-mission software, is a fundamental lesson just the same. The impact may be significantly greater on multi-mission software than on project specific software for this problem.

In the first case, prototype software was developed for a mission critical function, and it was liked so much by the users that it was adopted. Unfortunately, it had been developed as prototype software without the rigor needed to support mission critical software. And, since the prototype version of the software was reasonably capable, there was little support for spending the money or taking the time needed to develop it in a more formal manner. It was only after several years, and the apparent short comings in extensibility of the software, that funding was made available to develop the software in a manner more appropriate for such critical software.

In the second case, software was developed to provide a "quick and dirty" analysis capability that was extremely easy to use, so that rapid design trades could be made early in the mission, rather than using the software that handles more sophisticated models. In this case, the software was easy to use, however, users resisted giving up any of the capability for detailed modeling. The users pushed the software to become "not so quick and not so dirty".

In both these cases, user desires and expectations were not managed, which caused negative impacts on the software.

3. SUMMARY

The development of multi-mission software at JPL has been going on for more than 10 years. It has provided some real benefits to projects in the areas of cost savings and development time. However, several problems related to multi-mission software development have been encountered in recent years. While none of the problems would change the decision to develop software in a multi-mission format, these lessons should be thought about at the beginning of the development project. Hopefully, future developments will be able to learn from these lessons:

- Make plans for handling legacy software
- Develop a strategy for when it is best to fork off the software development
- Develop a testing strategy across all users
- Implement a system for tracking deployments
- Develop a clear statement of use strategies for the software
- Evaluate different architecture styles for multi-mission software
- Evaluate the need for an architecture that will support shared hardware resources
- Be careful of developing "quick and dirty" tools or prototyping software.

4. 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.