

## **COST EFFICIENT COMMAND MANAGEMENT**

**Theresa Brandt \*, C.W. Murphy \*,  
Jon Kuntz \*, & Tom Barlett \*\***

**\* Lockheed Martin Space Mission Systems  
1616 McCormick Road, Upper Marlboro, MD 20774  
Fax: (301) 925-0651  
*tbrandt@eos.hitc.com, cwmurphy@eos.hitc.com, jkuntz@eos.hitc.com***

**\*\* Goddard Space Flight Center  
Mail Code 511, Greenbelt, Maryland 20771  
Fax: (301) 286-1690  
*thomas.barlett@gsfc.nasa.gov***

### **ABSTRACT**

The paper describes the implementation of a Command Management System (CMS) for a National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center (GSFC) control center. Innovations being implemented in the CMS provide the infrastructure for increased operational efficiencies as well as high reuse for future missions, reducing both operations and future development costs. The command management design facilitates error-free operations, enabling the automation of routine control center functions, and allows for the distribution of scheduling responsibility to the instrument teams. The authors' analyses of existing command management systems and use of object oriented design techniques were instrumental in developing a command management system with anticipated 85% reuse.

### **BACKGROUND**

The Earth Observing System (EOS) will include a series of satellites designed to gather data for studying our changing global environment. The EOS Operations Center (EOC) will provide for scheduling, controlling and monitoring of these satellites. The Command Management component of the EOC is responsible for the creation and management of spacecraft loads, including commands stored on-board for later execution, and preplanned ground scripts for real-time execution.

Historically, the implementation of command management functions has been strongly tied to spacecraft design, preventing prior command management systems from being more reusable. An additional impediment to cost savings has been the manually-intensive nature of CMS operations. In the days of mainframe computers, there was less flexibility about the interface with remote instrument teams who had a wide range of existing equipment. The first CMS which moved to a workstation environment was intentionally made to be highly interactive, allowing the user to easily manipulate the data. Unfortunately, because of this human interaction with the software, the system did not lend itself to increased automation. An important goal for the EOS CMS was to increase operational efficiency.

### **OPERATIONS**

In previous NASA/Goddard missions, stored command loads were based on textual plans. The process was labor intensive for operations personnel with few tools to assist in real-time operations. The EOS command management system provides tools needed to manage the planned

operations of the EOS spacecraft and their instruments. Planned operations are scheduled using activities. Activities are groupings of preplanned spacecraft commands, including real-time and stored commands, EOC command language directives used to configure the ground system, command procedures, mode settings, and orbital events (such as sunrise, equator crossing, etc.). These activities and the components that comprise them are all previously defined using several GUI-based tools. Once these definitions are under configuration control they may be utilized and updated continuously throughout the life of the mission.

All of the definition tools provided are graphical. They are available to the Flight Operations Team (FOT) as well as all of the Instrument Operations Teams (IOTs). Both the FOT and IOTs submit definitions for commands, constraints and activities for use in the system. The tools restrict a specific user to developing a plan for his/her specific area of concern only - an instrument planner may only define activities and constraints using commands for his/her instrument. This restriction reduces the possibility of error by distributing the responsibility for a spacecraft subsystem or instrument to the most knowledgeable team.

The activity builder, used to define activities, restricts access to the command and procedure definitions based on the user's login and associated group, whether the user is a member of the flight operations team or one of the instrument teams. Restricted access reduces the risk of error in two ways: (1) a user will never be able to input a spacecraft command that hasn't already been validated and (2) one instrument team doesn't have access to another instrument team's commands or to those commands defined for the spacecraft subsystems.

Another important feature is the validity checks done on the activities as they are being defined. This increases the probability that the Absolute Time Command (ATC) loads can be generated without encountering any constraint violations. Thereby improving performance during ATC load generation, when processing time is more critical.

The planner/scheduler determines the relationship of the defined activities to specific orbital events and other activities and, based on this analysis, schedules the activities on a timeline. The timeline is a graphical representation of the continuous master plan. As the activities are placed on the timeline, activity-level constraints are evaluated. If a violation occurs the user is notified graphically; the activities in violation appear red on the timeline. As other activities are scheduled these violations may be corrected. Activities may be scheduled individually or they may be defined in a Baseline Activity Plan (BAP). BAPs are standing orders for the schedule. The planner/scheduler may also place single commands on the timeline, but this is generally discouraged as it bypasses some of the automatic validation performed before the ATC load is generated.

Once the planner/scheduler is satisfied with the timeline, he/she submits a portion of the timeline, referred to as the Detailed Activity Schedule (DAS), to be generated into the ATC load. The detailed activity schedules are contiguously submitted nominally on a daily basis. The DAS is then expanded into a command schedule.

The command schedule is command-level constraint checked with commands that have already been uplinked in a previous ATC load. If no violations are found, the planned stored commands are generated into a new ATC load and mapped into a memory model. The entire command schedule is incorporated into the ground schedule and the new load is automatically scheduled for uplink.

The ground schedule is a continuous list of the planned commands originating from the expanded detailed activity schedule. The flight operations team requests portions of this schedule, known as ground scripts. The ground scripts are executed in real-time operations and perform the planned commanding of the spacecraft.

## SOFTWARE DEVELOPMENT SAVINGS

Due to the long-term, multi-mission nature of the EOS program, our goals from the beginning were to isolate mission specific functionality and to identify software that could be shared and reused for future missions. Analysis of previous command management systems provided the cornerstone, enabling improvements over prior CMS implementations.

### ANALYSIS

During the early phases of system requirements definition for the EOS mission, the planning and scheduling functions were separated from the command management functions to facilitate reuse in the planning and scheduling subsystem, which is less dependent on spacecraft design than the CMS. The CMS development effort began with a study of existing systems performing functions similar to the functions of the EOS CMS. The goal of the study was to explore and document lessons learned, particularly in the areas of operational use of command management systems and evolvability of current CMS concepts and design. Ten NASA/Goddard missions and three National Oceanic and Atmospheric Administration (NOAA) missions were studied. The NASA/Goddard missions are scientific satellites and range from the complex (Hubble Space Telescope) to the simple (Small Explorer series). The NOAA missions studied support geosynchronous and polar orbiting weather satellites.

The study of existing command management systems included the study of documentation, interviews with flight operations personnel, and interviews with developers (Command Management Prototype Study). The interviews with flight operations personnel provided many ideas for operational efficiencies, as discussed previously. Interviews with developers provided insight into the processes that typically had been used in CMS development. The lessons learned from the development process are discussed below.

The primary problem with CMS development has been the lack of reusability in the CMS software. The development of a CMS has proceeded through lengthy requirements analysis, design, and implementation phases. Although CMS functionality is generally the same from mission to mission, the variations in spacecraft design have been thought to necessitate a completely new development effort. The one exception in the systems studied was the Small Explorer series. The CMS for the first in the series, Solar Anomalous and Magnetospheric Particle Explorer, was designed using Object Oriented methodology and the software was coded in C++. This CMS was extensively reused in the CMS for the next two missions in the series, Fast Auroral Snapshot Explorer and Submillimeter Wave Astronomy Satellite. The Small Explorer model was not appropriate for the EOS situation, however, because its reuse was highly dependent on having a similar spacecraft design. For EOS, each spacecraft will be designed and built by various manufacturers. As a result, the designers of software for the EOS ground system have no guarantee that the design of any spacecraft in the series will be similar to the design of any other spacecraft in the series.

Using the knowledge of CMS functionality gained from the study, the designers of the EOS CMS have separated the mission specific from the generic functions in both the requirements definition and the design.

### REQUIREMENTS

Critical to the success of any development effort is the analysis and refinement of the requirements of the system. These requirements should ultimately reflect the true intent of the requester's needs. They should not describe implementation approaches or design restrictions, nor provide

proposals for solutions to the problems. The analysts must be able to remove any of these artifacts from the requirements along with any ambiguities, inconsistencies, and errors. The analysts must work closely with the customer to refine the requirements so they accurately define the objectives of the system.

In developing the Command Management System for the EOC, Lockheed Martin Space Mission Systems employed a rigorous requirements analysis phase. The system requirements, originally defined by the customer, were analyzed and expanded to develop the software requirements. Ultimately, the software requirements are used as the baseline for the design and development of the system.

During the initial period of the requirements analysis phase, the system requirements were analyzed and refined based upon an updated operational concept for the EOC. This process involved close coordination with both the customer, NASA, and the user community, the FOT. The second phase of the requirements analysis focused on developing and refining the software requirements. Again, this process involved heavy interaction with both NASA and the FOT.

The process of refining the software requirements continued throughout the detailed design phase. This continued refinement assured NASA and the FOT that the system would meet their specifications. The continued attention to requirements analysis allowed the development team to identify those functionalities which were generic, not only within a subsystem, but across the entire system. It also helped to identify the capabilities which were specialized. This knowledge along with the use of the object-oriented design methodology, allowed the development team to design and implement a system poised for maximum reuse.

## DESIGN

The object oriented design approach shifts the development effort into analysis with emphasis on data structure before function. The object provides a more stable design environment that moves through the stages of analysis and design to implementation (Rumbaugh, 146). Changes in requirements generally effect functionality rather than object data and the relationships between the data. Therefore, changing requirements are less likely to impact the design. As more details have become available to the CMS development team the object model design has been refined. This seamlessness is yet another advantage of using object oriented design.

Object-oriented design and the C++ coding language have provided many advantages for reuse and streamlining. Object oriented inheritance, dynamic binding, the ability to override functionality, polymorphism, and encapsulation have all enabled the development team to design and implement an innovative command management system.

The EOC command management system is currently composed of six processes: schedule controller, ground schedule, spacecraft memory model, load catalog, image processor, and command constraint model (Command Management Design Specification). The schedule controller manages the processing of the detailed activity schedule to produce an ATC load and a ground schedule. The ground schedule process maintains a time-ordered list of all the preplanned commands scheduled in the detailed activity schedule: stored commands, real-time commands, and ground directives. The user selects a portion of the ground schedule, referred to as the ground script, for execution, automatically issuing commands to the satellite. The spacecraft memory model is responsible for managing the mapping of command loads into the models, both ATC and relative time command sequence (RTCS), and mapping table loads. The load catalog process oversees the creation of all load types: ATC, RTCS, table, flight software and instrument microprocessor loads. It builds the binary uplink load, produces reports, and catalogs the valid uplinkable load. The image processor maintains uplink images of the ATC, RTCS and table loads and has the ability to analyze and compare dump data to the uplink images or other dump images.

Finally, the command constraint model analyzes a sequence of commands to determine if a pre-defined constraint is violated. It performs this task on command loads, command procedures, and activity definitions.

The use of object oriented techniques in the design of the CMS facilitated the isolation and separation of generic functionality from mission specific. The generic, or common, functionalities were designed as parent or base classes in the object models. The specialized functionalities were then designed as child or derived classes. Derived classes inherit from the base classes thereby enhancing reuse. Object oriented design promotes sharing at several different levels. The inheritance feature enables both data and behavior to be accessed from derived classes. The load catalog model provides an excellent example of inherited behavior. The load object model base class provides the capability to ingest load contents, build the load data packets, create the load uplink and ground images, and compute the load cyclic redundancy check (CRC). The load object model derived classes (ATC, RTCS, table, etc.) create reports and perform other specialized functionality as required. As new requirements or missions are added, the base class code is reused, while new derived classes are added to the model, providing new functionality. The existing classes will not require code modifications.

Due to the fact that the mission specific data and behavior have been isolated, revisions for future satellites have been intrinsically insulated. Inheritance allows for extension of the design. It allows for new subclasses that may add new features. Yet at the same time, it is restrictive in that a subclass contains inherited features. This means that the functionality of creating a load will be inherited from the base class, but that the specifics of RTCS load generation, for example, are designed and implemented into the subclass. The RTCS load subclass is, therefore, restricted to ingesting the load contents, building the load data packets, creating the load uplink images, etc. The RTCS load may, however, override a behavior of the base class by redefining it. The function could first call on the base class functionality and then add additional functionality or it could completely rewrite the functionality. When the behavior is requested the appropriate function is called based on the type of object calling the function; i.e., the hierarchy and the dynamic binding provided by the compiler determines the correct function, rather than determining the correct behavior from the calling code. As an example, one of the instrument microprocessors could use a different algorithm to determine the CRC for an uplink load. The CalculateCRC function for that microprocessor would be overridden in its derived class.

The EOC directive object model provides an example of two types of reuse. It is software that is not only shared throughout the current system, but it will be reusable on future projects. The object model consists of both ground directives and spacecraft commands. We envision that most satellites will have commands or ground directives that are used to control the spacecraft. Therefore this object model will be reusable in many future systems. The specifics of the directives are isolated in the child classes. All the directives have a keyword or mnemonic; all have a scheduled time of execution; all have an association with a spacecraft. These attributes exist in the base class, but the specifics, such as binary generation, are in the derived classes. The command binary is considered to be mission specific. Isolating the specifics has reduced the need for rework, enabling us to accommodate the next mission, while changing or replacing very few classes, if any.

Polymorphism shifts the burden of deciding what implementation to use from calling code to the class hierarchy (Rumbough, 8). This means if two classes (e.g. ATCLoad and RTCSLoad) have logically similar operations (named the same, e.g. CreateLoad) the calling code doesn't have to determine which of the class' behaviors it needs, because it is working with one of the specific classes. Thus the CreateLoad operation for that specific class is automatically called. Polymorphism simplifies the calling code, enabling it to be generic. The calling code may always call CreateLoad and not have to determine that it needs ATCLoad::CreateLoad or RTCSLoad::CreateLoad. This also makes the calling code generic such that if a new load type is

added in the future, this code won't need to be updated; a new class may need to be created with an operation of its own - CreateLoad.

## IMPLEMENTATION

Currently, the EOS CMS development effort is in the implementation stage. The total number of classes planned during the design phase was 171. Of these, 96 were planned for an initial release and 75 for the final release. Code and unit test of the first release is complete, with 111 classes actually being implemented for the CMS. Of these, 13 are mission specific, yielding an 88% reuse estimate. We anticipate a similar reuse percentage for the second release.

## CONCLUSION

From the operations concept/requirements phase through implementation, the EOS CMS has consistently focused on cost efficiency. Its highly reusable system, developed using object-oriented methodologies and C++, significantly reduces development cost for future spacecraft. Operations costs are minimized by providing operations personnel with the appropriate tools to simplify their daily work and automate, when possible. As a result of many studies, interviews, and evaluations over the entire life cycle, the EOS CMS developers have been successful in designing a highly reusable, cost efficient system.

## REFERENCES

EOSDIS Core System Project. Flight Operations Segment (FOS) Command Management Design Specification for the ES Project, Hughes Information Technology Corporation, Upper Marlboro, Maryland, 1995.

Murphy, C.W. and Neil Clabough, EOSDIS Core System Project. Command Management Prototype Study Phase Results Report for the ECS Project, Hughes Applied Information Systems, Upper Marlboro, Maryland, 1994.

Rumbaugh, James, et al. Object-oriented Modeling and Design, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.