ORION GN&C MODEL BASED DEVELOPMENT: EXPERIENCE AND LESSONS LEARNED

Mark C. Jackson[1]
Charles Stark Draper Laboratory, Houston, Tx, 77062

and

Joel R. Henry[2]
NASA Johnson Space Center, Houston, TX, 77058

The Orion Guidance Navigation and Control (GN&C) team is charged with developing GN&C algorithms for the Exploration Flight Test One (EFT-1) vehicle. The GN&C team is a joint team consisting primarily of Prime Contractor (Lockheed Martin) and NASA personnel and contractors. Early in the GN&C development cycle the team selected MATLAB/Simulink as the tool for developing GN&C algorithms and Mathworks autocode tools as the means for converting GN&C algorithms to flight software (FSW). This paper provides an assessment of the successes and problems encountered by the GN&C team from the perspective of Orion GN&C developers, integrators, FSW engineers and management. The Orion GN&C approach to graphical development, including simulation tools, standards development and autocode approaches are scored for the main activities that the team has completed through the development phases of the program.

Nomenclature

ARINC	=	Aeronautical Radio, Incorporated	GN&C	=	Guidance Navigation and Control
CC	=	Cyclomatic Complexity	GUI	=	Graphical User Interface
CDR	=	Critical Design Review	MBD	=	Model Based Design
CSU	=	Computer Software Unit	PA-1	=	Pad Abort One
EBA	=	Empty Box Architecture	PDR	=	Preliminary Design Review
eML	=	embedded MATLAB	PIL	=	Processor In the Loop
EFT-1	=	Exploration Flight Test One	SDP	=	Software Development Plan
FSW	=	Flight Software	SIL	=	Software In the Loop

I. Introduction

THE Orion Guidance Navigation and Control (GN&C) team has developed GN&C flight software using a Model Based Development (MBD) process that includes developing algorithms in the Simulink® graphical design environment, automatically generating C^{++} code, and integrating the code into an Aeronautical Radio, Incorporated ARINC 653 partition. The Orion GN&C application is large and complex and it was developed by a geographically separated team, so there are many experiences and lessons learned that apply to projects within and outside the aerospace industry.

This paper provides an assessment of the successes and problems encountered by the GN&C team from the perspective of Orion GN&C developers, integrators, FSW engineers and management. The Orion GN&C approach to graphical development, including simulation tools, standards development and autocode approaches are assessed for each of the main activities that the team has completed through the development phases of the program. A chronological approach is taken to communicate both the Orion MBD process and the lessons learned from that

---

[1] Principal Member of the Technical Staff, Charles Stark Draper Laboratory, 17629 El Camino Real, Suite 470, Houston, TX 77058, AIAA Senior Member.
[2] Orion GN&C Software Functional Manager, NASA – Johnson Space Center, 2101 Nasa Parkway, Houston, TX 77586.

process. The following sections provide relevant background on the Orion project as well as problems and successes encountered during the design phase prior to Preliminary Design Review (PDR), during the development phase prior to Critical Design Review (CDR) and during post CDR production. At the end of the paper, lessons learned are summarized and recommendations provided for future MBD projects.

II. Orion Project Background

Three aspects of the Orion project bear on the the MBD process: Application size and complexity, GN&C team makeup and geographical distribution, and legacy tools and infrastructure. This section provides background in each of these areas to allow the reader to understand and assess process decisions and lessons learned.

Application Size and Complexity. The Orion Spacecraft is NASA's vehicle for manned exploration outside of low Earth orbit. The spacecraft consists of three main components: a manned capsule, or Crew Module (CM), a Service Module (SM) and a Launch Abort System (LAS). The CM houses the GN&C subsystem shown in Figure 1. At the center of the GN&C subsystem is the GN&C Flight Software (FSW) which executes on the Vehicle Management Computers (VMC's). This software receives inputs from navigation sensors and pilot controls and displays and commands the appropriate effectors on the CM, SM and LAS to accomplish mission objectives.

The Orion GN&C software operates across a variety of mission phases, including pre-launch, ascent, Earth orbit, transist, loiter, rendezvous, docking, entry and various abort scenarios. During these phases, GN&C communicates with sensors on the CM and SM, and commands effectors on the CM, SM and LAS. The software must operate in both manual and automated modes and must handle commands from the crew and the ground. The software must also execute complex guidance and navigation algorithms while controlling highly dynamic configurations during entry, ascent aborts and orbital maneuvers. The resulting breadth of algorithm types drives a multi-rate architecture to meet CPU usage allocations.

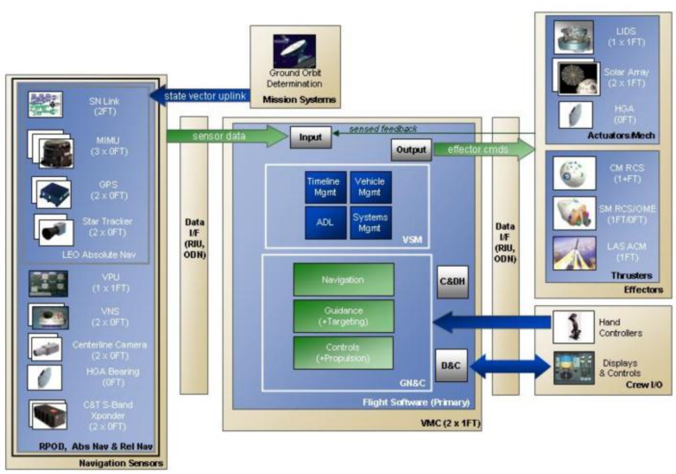


Figure 1. Orion GN&C Subsystem.

The Orion project has evolved since its inception. Originally, the first mission to fly with the GN&C software would have included all three components (CM, SM and LAS) and would have been required to execute nearly the entire breadth of GN&C capability. However, changes in manned exploration schedules and budgets have resulted in a phased development approach consisting of test flights of increasing capability. The first flight of the Orion CM will now be a test flight of the CM only, launched on a commercial booster to an elliptical orbit designed to achieve a high speed entry to test CM thermal protection systems. Termed "Exploration Flight Test One" (EFT-1) this test will limit the required GN&C functionality to navigation-only during pre-launch through booster separation, followed by full GN&C to guide the capsule to a water landing target through the orbital coast, atmospheric entry and parachute landing phases (Figure 2). For EFT-1, the LAS is only a mass simulator, so no GN&C abort algorithms are required for EFT-1. This phased flight test approach means that portions of the GN&C software design are complete and undergoing test and integration with the Orion avionics, while other components and algorithms remain at the post-PDR design complete phase.

GN&C Team. The Orion GN&C team is a large, geographically dispersed team consisting of members with diverse experience and backgrounds. GN&C architecture and algorithm developers include NASA civil servants at Johnson Space Center in Houston, TX, Lockheed Martin employees in Houston, and Denver Colorado, as well as developers from Honeywell, Draper and other sub-contractors located in Florida, Massachusetts, Minnesota and other states. Team members have program experience on the Space Shuttle, International Space Station, Mars exploration programs and commercial satellite development, to name a few. GN&C software development experience ranges from "classical" development processes that use detailed requirements and hand-written code, to more automated processes that depend wholly or partially on automated code generation. Additionally, team members' views on MBD processes tended to vary with their domain expertise, with navigation and guidance developers often preferring text-based algorithms, and controls and architectural designers preferring the data-flow depictions of algorithms afforded by some MBD tools such as Simulink®. As discussed in the sections below, both the geographical separation and diverse experience base of the team impacted the MBD process, both in positive and negative ways.

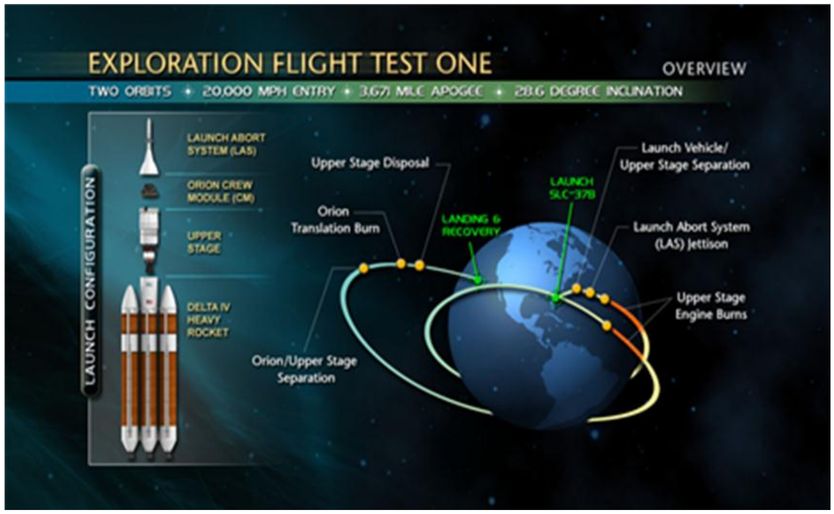


Figure 2. EFT-1 Mission Overview.

Legacy Tools. Prior to selection of a prime contractor, the NASA team had developed a highly capable set of simulation tools and prototype GN&C algorithms adapted from legacy code in the C language. Simulation models and GN&C algorithms were integrated into functioning executables within the JSC "Trick" simulation environment (Ref 1). The simulation was termed the Advanced NASA Technology Architecture for Exploration Studies (ANTARES). Most algorithm development and performance analysis prior to PDR was conducted using the ANTARES. After contract award, Lockheed Martin also developed an independent Trick-based simulation named "Osiris." Osiris was architected to execute the GN&C FSW algorithms as a separate process. This was done in done to allow sharing of the GN&C FSW between Osiris and ANTARES. After PDR, Osiris became the simulation tool used to develop and test GN&C algorithms.

The GN&C algorithms at PDR were derived from a combination of legacy C code algorithms (about 75%) and Simulink algorithms which were autocoded into C and integrated into the prototype C architecture. Several proposals were considered to move the algorithms from prototype code to production software. Ultimately the decision was made to use Simulink in an MBD process for the FSW development, while retaining the legacy C code simulations. It was decided to create a development environment that included the C simulations communicating with a MATLAB/Simulink process as described in the next section. Some of the factors in this decision were:

- The prime contractor's FSW team was staffed under the assumption that autocode would be used to generate the GN&C algorithms – so there were insufficient resources to allow manual coding of algorithms from detailed written requirements
- The MATLAB/Simulink process would allow development and debugging of the algorithms in their native MBD environment
- The legacy simulations were fully developed and functional, so development of a Simulink simulation would take time and resources that were deemed unnecessary
- The Pad Abort One (PA-1) flight test had used a similar process for software development.

Future papers will compare the benefits and drawbacks of the Orion MBD process with traditional hand-code processes. The focus here is to enumerate and explain the techniques, issues and lessons learned from the processes and tools used.

III. Pre-PDR Analysis, Design and Process Development

Prior to PDR, the team focused on generating and validating GN&C requirements and developing preliminary GN&C algorithms. Most of this work was done using the ANTARES and Osiris simulations described above.

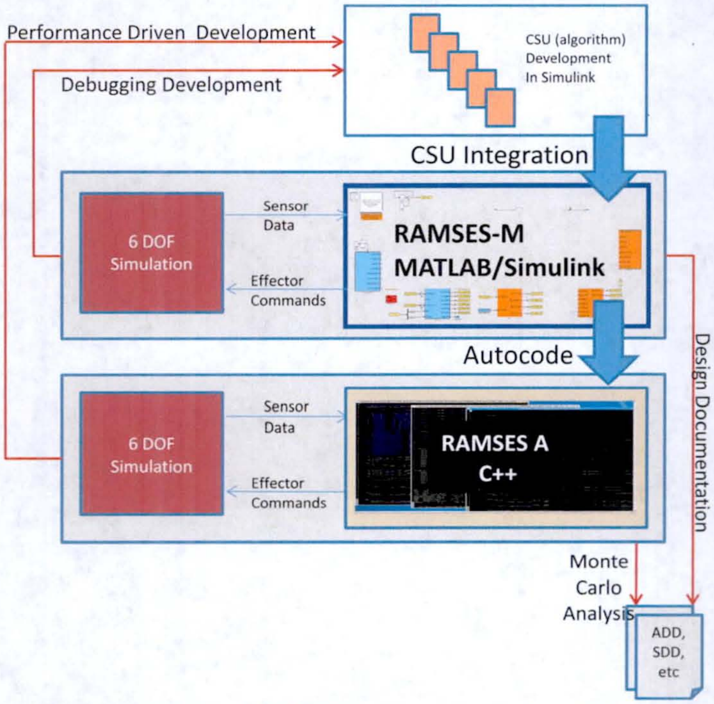


Figure 3. Orion GN&C Development Tools within the Development Cycle.

Given the state at PDR, a plan was generated to move toward an MBD development process. The plan included several critical new tools and capabilities - specifically:

- A Trick simulation to MATLAB process interface to allow algorithm development in Simulink® with legacy truth models in C
- An "empty box" architecture in Simulink® that would house the GN&C algorithms as they were developed
- A suite of Unix and MATLAB scripts that enabled the execution of the combined Trick/ Simulink® tool

Prior to PDR, the team began to implement the plan by developing the tools and processes of Figure 3. First, algorithms would be developed as Simulink diagrams. Often these were based on pre-existing algorithms that were already implemented as prototype C-code. For this reason, a period of time would be spent to "translate" existing prototype C-code into Simulink diagrams.

The Simulink representation of a particular algorithm was called a Simulink Computer Software Unit, or just "CSU." The term "CSU" is used in many programs to define the lowest level testable units, but on Orion, CSU's varied in size and complexity, and most had several testable sub-modules. The Orion terminology is used for this discussion.

Next, the algorithms would integrate into a Simulink framework that allowed execution of the integrated set of algorithms with a 6 DOF simulation. The Simulink Framework was essentially a wrapper around the GNC algorithms that provided execution, mode-ing and debugging in the native Simulink environment. Dubbed the "Rapid Algorithm MATLAB Simulink® Engineering Simulation (RAMSES)," this wrapper eventually housed all of the GN&C algorithms, and provided models of non-GN&C FSW that were required for GN&C execution.

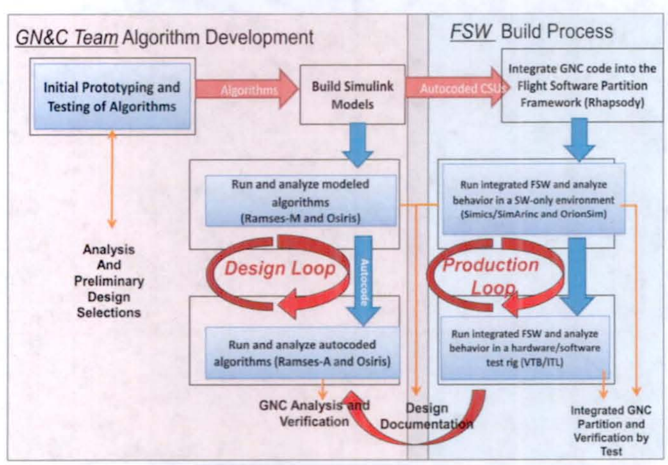


Figure 4. Design and Production Cycles.

The RAMSES wrapper would be driven by a heritage 6 DOF C-code simulation. The decision was made not to build a Simulink simulation, since most of the team was familiar with the legacy simulations, and there was a desire to leverage the existing capabilities. However, this setup did add complexity to the development cycle, so the pros and cons of this approach are discussed below.

Developers were to do most of their developing, debugging and integration work using the native Simulink version of RAMSES,. For analysis work however, RAMSES would be autotcoded in its entirety and executed as a UNIX process with the legacy C simulation.

This would provide higher speed execution for Monte Carlo analysis and some debugging. When execution of the tool was in the native MATLAB process, the tool was referred to as "RAMSES-M" (M for "MATLAB"). When executing the autocode as a compiled process, it was called "RAMSES-A" (A for "Autocode").

Figure 4 shows how the above tools were used in the design and production cycles. At the upper left, the initial prototyping, requirements validation and pre-PDR design and analysis was conducted primarily using legacy C-based simulations and algorithms. During the post-PDR phase, the tools of Figure 3 were used to translate and mature the algorithms as Simulink CSU's. The iterative process of development of algorithms in RAMSES-M and performance testing in RAMSES-A was the "Design Loop" and was the primary activity in the post PDR period. As the team entered the post-CDR production phase, autocoded CSU's were delivered to the GN&C FSW team for integration into the GN&C partition. Testing on the GN&C partition occurred with software emulations of the processor environment as well as on the actual Orion processors. Iterations on the GN&C partition due to this testing were referred to as the "Production Loop." Sometimes, errors or changes to the Simulink CSU's were needed, so design change requests were fed back to the Design team for modification using the RAMSES tools. The GN&C Design and FSW teams worked closely together, so response to Design change requests was very rapid. Also, the thoroughness of testing in the RAMSES environment meant that very few errors were found in the Simulink models. Most of the problems encountered were related to inefficient execution, as discussed in the Post-CDR section below. The following sections provide detailed lessons learned for the PDR to CDR design and Post CDR production processes depicted in the figure.

IV. PDR to CDR Development

During the post PDR period, the GN&C and FSW teams transitioned from hand code algorithm prototyping, to an MBD process that produced preliminary versions of the GN&C CSU's in Simulink. During this period, several major efforts were undertaken: development of the RAMSES GN&C wrapper in Simulink, development of an Empty Box Architecture (EBA) which provided the moding and interfaces between CSU's, translation of many of the GN&C algorithms from prototype C code to Simulink CSU block diagrams, and integration of the CSU's by populating the empty boxes.

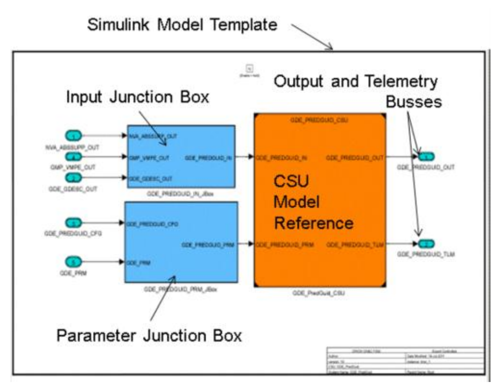


Figure 5. Typical GN&C CSU Diagram.

Figure 5 shows a Simulink diagram of a typical GN&C CSU Junction Box. The orange block is the CSU itself which contains the algorithm. Its interfaces consist of 4 Simulink "buses" which correspond to structured data types. The input and parameter buses enter the CSU from the left and the output and telemetry buses feed the output port ovals on the right. Outputs are those signals needed by other downstream CSU's, while telemetry data are additional data needed for analysis and insight into CSU behavior. When autocoded, the orange block produces a Class with a method whose calling arguments are data structures corresponding to these four Simulink bus types.

To the left of the orange CSU algorithm are "Junction Boxes." These Simulink subsystems route data from upstream CSU's and other sources and multiplex the data into the input and parameter boxes. The orange CSU block is a Simulink model reference block. This means that the functionality within the block is contained in a separate Simulink model (.mdl) file from the rest of the block diagram. This is important for configuration management since changes to the algorithms within this block affect only the associated file.

The method used to provide parameters to a Simulink model reference block is an important design decision to make early in the Simulink design process. In this context, "inputs" are time-varying signals that are operated upon by the CSU to produce the outputs. "Parameters" are quantities that configure the CSU and remain static during most execution calls. These data may be changed by the moding software on asynchronous events, but they otherwise remain fixed. Control gains are examples of parameters, while control errors are examples of CSU inputs. The Orion GN&C team elected to pass parameters into each CSU via a parameter bus as discussed above. This has several important advantages:

- The parameter interface is clearly visible in the diagram
- Parameter data structure types are clearly defined using Simulink Bus definitions

- The parameter interface in the resulting autocode is clear by inspection – as a structured pointer calling argument for C ++ autocode.

However, one major disadvantage of this technique is that parameters must be routed like signals to locations within the CSU. This means that several important Simulink atomic level blocks cannot be directly used. An example is the gain block at the top of Figure 6 that multiplies *u* by K to produce *y*. Since the gain parameter, K, is a member of the parameter bus structure, it must enter the diagram on a signal "wire" as shown at the bottom of the figure. This clutters the diagram and reduces readability. On Orion, special gain blocks were created that had hidden "From" blocks to allow the designer to use a similar gain form as the top diagram, but it is more desirable to use the normal Simulink language whenever possible. Later versions of Simulink, which allow parameters to be specified via other means, are worth consideration for future projects.

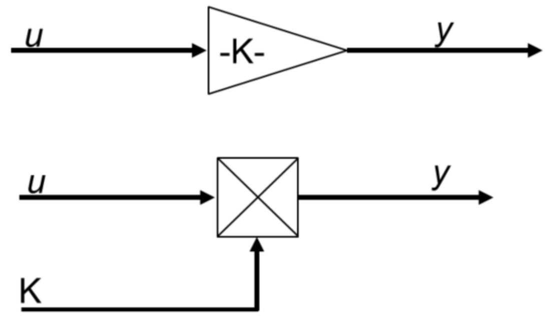


Figure 6. Gain Parameter Comparison.

Figure 7 shows a typical collection of CSU's or a "domain." Each Orange CSU block contains the combination of Junction Boxes and a CSU algorithm block as shown in Figure 5. This example is the CM control domain and contains three CSU's that control the CM by commanding its reaction thrusters. Each CSU receives inputs from upstream CSU's passed into the domain. Each domain also contains moding logic that enables or disables CSU's according to mode commands received from the GN&C moder/sequencer to be discussed later. During exo-atmospheric flight and during the guided entry, the RCS Control CSU is enabled to provide rate change commands to the Thruster Logic. Once under the main chutes, RCS Control is disabled and the Touchdown Roll Control CSU is enabled to turn the CM to a downwind heading for splashdown.

On the Orion project, the Orange CSU boxes are autocoded and integrated as independent functions into the GN&C partition. So from the domain level up, the Simulink diagrams are not used to produce final flight software. Rather, they provide the interfaces and moding to allow CSU development and integration in a Simulink environment for execution in a closed loop simulation.

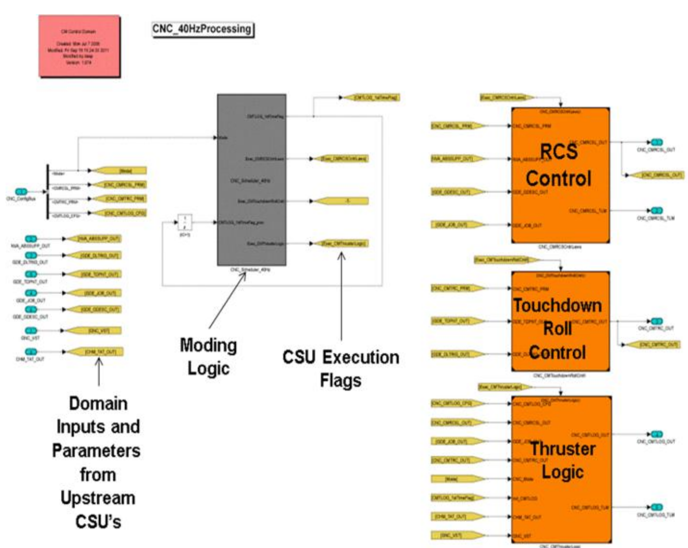


Figure 7. Typical Domain Containing CSU Blocks.

All the CSU's in a domain block execute at the same rate – in this case at the control rate of 40Hz. One of the lessons learned in developing the Orion domain architecture was that collecting CSU's in like rate groups simplified the correct modeling of rate group latencies and interactions. Early domain versions contained all the CSU's of related functionality regardless of their execution rate.

The collection of domain diagrams are wired together with other domain blocks of like rate and collected into the high and low rate blocks of Figure 8. These rate blocks, together with the other blocks in the figure, comprise the RAMSES Simulink development environment in its final form for EFT-1. Inputs, outputs and telemetry are provided by the "IOP" blocks which mimic the Input-Output Partitions in the flight software by providing interfaces to the simulation. Other partitions which interface with GN&C are also modeled, including the C&DH partition and the Timeline Manager (TMG) and Vehicle Manager (VMG) partitions which provide vehicle configuration data and mission segment information to GN&C.

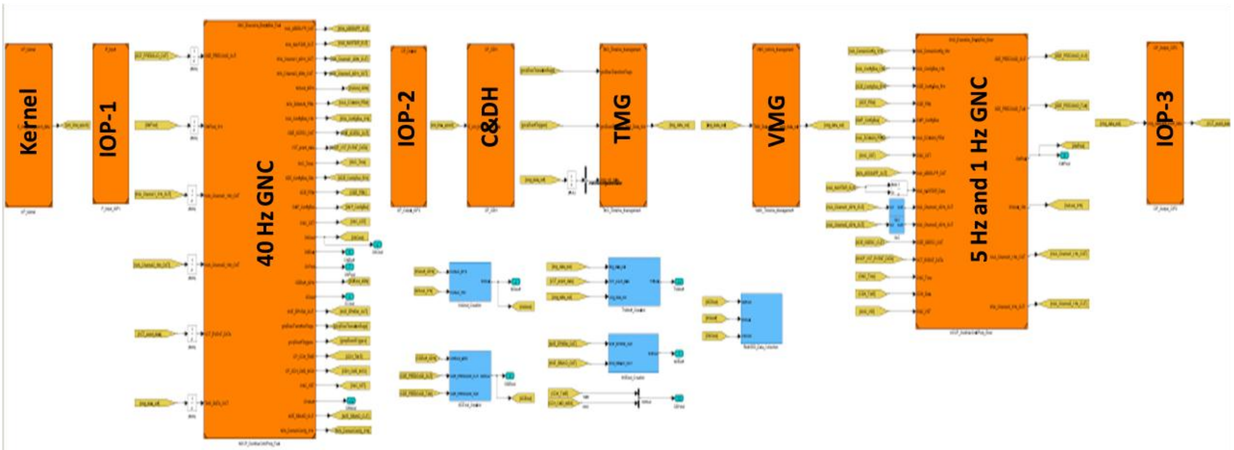


Figure 8. RAMSES Simulink Wrapper for GN&C FSW.

In a typical MBD process, a company or organization may have an existing library of graphical GN&C components to draw from. In the case of Orion, because much of the algorithm prototyping was initially done in C, a “translation” period was scheduled to convert C algorithms into Simulink block diagrams. The intent of the translation period was to re-produce algorithms from existing prototype C code. The entry flight phase was judged to have the most mature algorithms, so entry modules were scheduled to be the first wave of algorithms translated and integrated into RAMSES. This pathfinder process highlighted several aspects of the Orion plan that worked well, as well as important lessons learned related to MBD development as done for Orion. Recommendations for the Mathworks Simulink product are also highlighted, some of which have been communicated to Mathworks and incorporated in later Simulink versions.

Scaleability. MBD tools on Orion required considerable customization to allow them to scale to the project size. The final Orion GN&C application will produce well over 100,000 lines of autocode. The Orion experience was that the Simulink product did not provide an adequate development capability “out of the box” for this size application. The team found that the build and execution speed of the RAMSES-M/A development environment tended to diminish as CSU’s and capability were added. So continual development of custom tools, iteration with Mathworks and dissemination of improved standards and techniques was required to make the development cycle time acceptable. Many of these tools, standards and other lessons learned are useful for similar projects and are described in this paper. Since the EFT-1 flight reduced the size of the overall application, and since many of the tools and techniques are now available and understood by the team, future development is expected to be more efficient, but compromises may still need to be made to isolate flight phases or functions for development as Orion moves forward.

Configuration Management. Projects electing to follow an MBD process should prepare for configuration management requirements that differ from hand code. Two aspects of MBD tools in particular require attention: separation of graphical modules, and merging graphical changes. Early versions of the Simulink tool included all functionality in a single model file (“.mdl” file). This was not practical for a large project, since all developers operated on a single configuration managed object, even though they may have been making changes to separate subsystems within the model. For this reason, Orion used the Simulink *Model Reference Block* (MRB) capability discussed above, which allows subsystems to be expressed in separate model files and “referenced” in the top level diagram. This MRB capability evolved significantly over the Orion development period and new capabilities for parameterization and autocoding of MRBs are now available. Since MRB’s are a virtual necessity from a configuration management standpoint, project architects should thoroughly investigate the latest MRB capabilities prior to deciding on how to integrate and parameterize GN&C algorithm units.

Graphical merges are another important consideration for MBD processes. When multiple changes are simultaneously made to the same file by multiple developers, a merge is required to incorporate updates. Text merge tools are highly effective for hand coded applications, but graphical merge tools are more expensive and less effective than their text counterparts. Orion purchased licenses from third party vendors for graphical merge tools, and found the tools to be useful enough to justify purchase, but not as effective as text merge tools. The limited distribution and training of these tools to the team limited the amount of parallel development that the team was able to accomplish. This was less of a problem at the CSU level, but often CSU developers needed to make changes at the domain level and would have to either wait until another developer completed domain level changes, or make parallel changes and use the graphical merge. Since domain level changes were often in the interface, the number of graphical merges was reduced by wrapping the junction boxes associated with each CSU in another MRB, as discussed above and shown in figures 5 and 7. Also, a one POC per CSU standard was enforced to prevent parallel changes to CSUs. As algorithms matured for EFT-1, parallel development has reduced and the need for merges has diminished. However, the Orion team considers the graphical merge capability an important need for the development of remaining flight phases and continues to look for acceptable options.

Complexity of mixed tool development. The RAMSES development environment – and its associated interface to and hand-coded simulation has been successful for Orion EFT-1 development. However, the use of mixed simulation/FSW development environments is not recommended for the early development phases for projects that do not have a strong legacy of hand coded simulations. The Orion development environment requires a fairly complex set of scripts to start both simulation and FSW processes, load simulation and FSW configurations and manage environment variables and other initialization items. The environment also requires familiarity with several tools, including familiarity with C code, the JSC "Trick" simulation environment, the suite of execution scripts and the full suite of Simulink tools and custom scripts. This means that engineers new to the project need significant training prior to starting development. The team is currently looking at options for development in future flight phases, including simplified scripting and configuration management tools and the creation of simplified medium fidelity Simulink simulations for early algorithm development.

Logging and Debugging. Two of the areas in which the Orion team did significant custom work were data logging and graphical process debugging. The Simulink development tool provides several options for writing to files and to the MATLAB workspace in the form of output blocks. However, these options are not data configurable as required for high complexity applications and they have some incompatibilities with arrayed structure types. For this reason the Orion team developed a data logging subsystem – RAMSES-M Record – to allow user selection of output parameters for analysis and debugging. While this tool will pay dividends for future Orion development, a generic logging capability that is delivered as part of Simulink, that does not require changes to the diagram for configuration is highly desirable.

Additionally, debugging in the native Simulink environment was often difficult. Simulink provides three separate debugging tools, one for Simulink, one for Embedded MATLAB and one for Stateflow. All three are different, so training or familiarization is required for each. The most popular debugger by far among developers was the Embedded MATLAB debugger which provides intuitive, graphical breakpoint insertion, "hover" displays of variable and parameter values, etc. In fact the ease of eML debugging led several developers to develop initial algorithms as large eML blocks, which were later broken down into graphical elements with smaller eML functions contained in lower level blocks. The team also developed custom eML debug blocks which allowed execution control and viewing of data on Simulink buses between Simulink subsystem blocks.

Modeling Standards. Since this is NASA's first major manned project to utilize fully the MBD process with the MathWorks software, a set of Modeling Standards was developed to aid developers. The original source of the document was the MAAB (MATLAB Automotive Advisory Board) Standards and those from the Honeywell LaserRef6 project. Lessons learned from the CEV Pad Abort 1 (PA-1) flight test project were included as well. However, a majority of the content is derived from the development process itself. The document is a living document which is continuously evolving based on developer feedback and autocode performance.

The main purpose of the Standards was to enhance the consistency, readability, efficiency and compatibility of the many models that were being developed amongst a large group of developers.

There are currently 155 standards and guidelines in the Orion GN&C MATLAB/Simulink Standards document (available at **www.mathworks.com/aerospace-defense/standards/nasa.html**).

To complement the ORION standards, a set of Model Advisor scripts tests were created to streamline the checking of many of the non-objective standards. These checks could be run automatically to determine any standards violations. There are currently 68 model advisor checks for the 155 ORION Modeling standards.

When autocode errors/incompatibilities were found, the source Library blocks were modified to avoid them or a standard was written to avoid them. Many times a model advisor check was also written to automate the finding of a pattern. Since the standards document and the Model Advisor configurations underwent significant modification, having these elements under configuration control was essential.

Modeling Library. The standard set of Simulink blocks contained many blocks that are not compatible with the RTW Embedded Coder or the GN&C architecture. To clarify which blocks are compatible, the team created the Orion Library. The Orion Library contains only blocks that are compatible with the Embedded Coder and the Fixed-Step solver, conformed to the modeling standards, and whose resulting autocode complied with the project's SDP. Also, no blocks that require variables/parameters from the workspace are included since the architecture does not support these as described above. This blockset (Figure 9) provides the GN&C and FSW developers with a standardized set of tools that are certified to be compatible with the modeling standards and will work seamlessly with the Orion GN&C architecture.

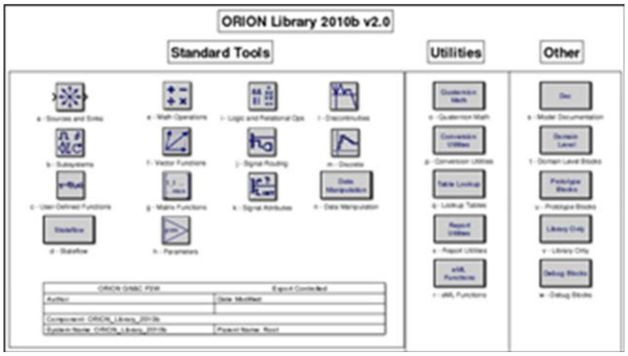


Figure 9. Orion Library.

The ideology applied when designing the ORION library and Modeling Standards was that the models should speak for themselves. A reviewer should ideally be able to review a model and understand all of the details of the algorithm being modeled without having to click inside a block and review the block settings or have prior knowledge of a block's functionality. Data types, integration types, data limiting, etc. should all be clear from the diagram alone. Block illustrations should convey the functionality of the block.

The Simulink tool allows for the creation of unique GUIs for each block. The ORION library uses these unique GUIs to hide certain block settings that need to remain consistent throughout the model for compatibility or autocode format (e.g. the sample rate).

Another issue with the standard set of Simulink blocks is that not all of the significant block characteristics are visible. For example, the standard Simulink "Constant Block" has a setting for limiting the maximum and minimum value of the output. However, this restriction does not show up on the icon for the constant block. The only way to know if this setting is used is to open up the block dialog itself. Reviewing the block GUI dialog is not an option when the block is printed or viewed in a document. Since these models are considered self-documenting they should be as readable, descriptive and transparent as possible. All of the blocks used in the ORION Library either have the parameters hidden that should not be changed or the icons have been modified to adequately show the functionality of the block based on the parameter selection.

Blocks in the ORION Library are color-coded to distinguish between block types and enhance readability – as required by the standards. For example, an eML block in the Orion GN&C Library is colored grey whereas the standard Simulink version looks identical to any other subsystem block.

Autocode Configuration Settings. The ORION Library includes the official configuration settings used for the autocode. There are numerous options available for formatting the autocode. Many times it is unclear how a setting will affect the autocode until the result is analyzed and compared. The focus of the configuration settings are on efficiency first, testability second, and readability third. Although Orion did not require formal reviews of the autocode, it is still very useful to maintain traceability for debugging purposes, so readability is still important. The ORION settings were mostly chosen by a trial and error approach. The project ran several trade studies to understand the effect of the configuration settings on the autocode.

The configuration settings have options to both autocode a model and compile it. During development, both the capability to produce the code and compile it were used. This meant that any compilation issues early and not when the code was integrated into the larger project.

The autocoding tool includes the option to automatically compile the generated code after completion of each model. This option allowed us to find compilation issues early and at the model level instead of when the project was compiled in its entirety.

The ORION project utilized the "referenced configuration set" ability for managing the settings for the simulation/autocoding. These settings can be managed on a model by model basis or via a referenced set that all models point to. This allowed us to manage the configuration settings for all of the models in a single object. This prevented us from having to manage configuration changes on a model by model basis, which would not have been compatible with the CM system. In this case, every time a change is made to a single setting, all of the models would need to be revised.

As discussed above, the CSUs were integrated into the partition at the Junction Box layer. Due to the object-oriented format of the Rhapsody level code, the interface required the use of the C++ (Encapsulated) target. The 2010b version of Simulink was the first release to include this target and the team found many incompatibilities that required worked arounds.

Modeling Template. The library also includes a Template file, illustrated in Figure 10, for use in creating a CSU model and the subsystems contained within it. This has proven to be a very useful tool to ensure that all of the models are consistent and compatible from the start. The template can be thought of as a formal schematic representing the algorithm design that is also the direct functional representation.

This template has the following features:

- Standard configuration settings (through config set reference). The configuration sets contain the settings for model simulation and custom autocode options (see “Autocode Configuration Settings” above).
- Information block in the lower right section of each level of the model, to display the project name, version, author, CSU name, subsystem name, and parent subsystem name. This allowed users to navigate printed models.
- Version data specific to the project’s CM tool, which is automatically modified every time the model is checked in to the system.
- Model size constraint borders, that allow each level to be printable on 8½”x11” or “ 11”x17” paper, which prevents large unprintable models (each level of the model contains these borders)
- Standard input, output and parameter port stubs.
- Annotation block for adding comments to the diagram

Training on Tools. Initial Training on the use of Simulink was fairly quick and concise. A week long session was given to go over the early version of the ORION Modeling Standards, the ORION Library and to show examples of early prototype work. However, a functional EBA was not available yet, and the RAMSES-M tools for closed-loop simulation connection were not complete. The unit testing framework was also not available, so developers had limited ability to build and test models.

At the time, thorough Simulink experience was limited to a few individuals, the team at large was not familiar with the tool. The standards that were used to train the team were preliminary, and many issues with the autocoder were not yet known. All of these factors led to a shaky start to the development cycle. Future projects should use the lessons learned herein to have development tools ready, and to provide more complete training on standards, development processing, unit testing, and general MBD development.

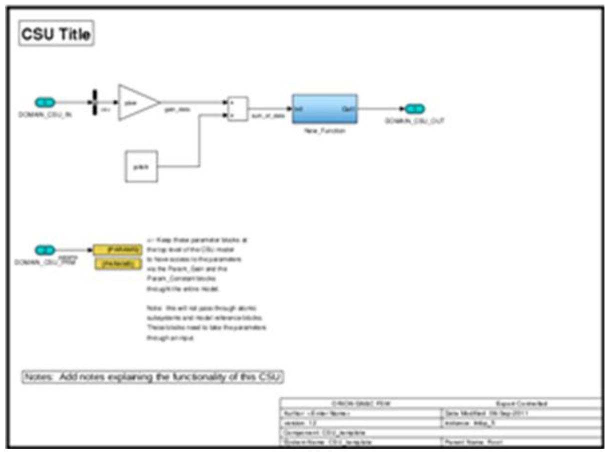


Figure 10. Orion GN&C Model Template.

Algorithm Modeling. Not all algorithms benefit from graphical dataflow implementation. Certain GN&C algorithms are ideally suited for graphical representation in a data flow format like Simulink. The overall flow of GNC data at the top level, and many of the embedded control laws are expressed naturally as block diagrams. Some attributes of algorithms that are not as easily expressed as data flow diagrams include iteration, expression of complex equations, state machines, low level data manipulation and object-oriented representations. The Simulink environment provides tools to allow embedding algorithms with some of these attributes into the graphical dataflow layers. These include embedded MATLAB scripting, Stateflow diagrams and S-functions to call externally coded algorithms. The final Orion design for EFT-1 is one example of a successful mix of these languages. In particular, the GN&C sequencer was coded using an object-oriented Unified Modelling Language (UML) tool. The code for the sequencer was incorporated into Simulink via an S-function interface. The Orion navigation algorithms made significant use of Embedded MATLAB (eML) to allow expression of vector and matrix equations while maintaining top level and intermediate level dataflow diagrams in Simulink. Orion guidance algorithms made heavy use of Stateflow diagrams to implement iteration and eML functions to implement complex guidance equations. Issues with testing stateflow algorithms were sufficiently overcome for EFT-1 but Orion continues to communicate with Mathworks on further improvements. Finally, Orion also developed the GN&C sensor interfaces using Simulink and eML. This provided consistency in delivery format and autocoding processes for GN&C components, but there was little gained in clarity or abstraction. Future projects may wish to consider allowing hand code or UML code for hardware interfaces and integrating them into the Simulink project using S-fuctions.

As the project matured and the team learned more about the benefits and limitations of the tools, a chart (see Table 1) was created to help aid developers to use the most suitable tool for the algorithm being modeled.

Table 1. Standards for Use of Simulink Language Tools.

Algorithm Type	Simulink	Stateflow	eML	Notes/examples
Simple Logic •if/then •switch/case •for/while loops	X	X	X	Ex: If/then with <5 paths and no nesting
Complex Logic •nested if/then •nested switch/case •nested for/while loops		X preferred	X	Ex: If/then with numerous paths and multiple levels of nesting
Simple/Short Numerical Expressions	X			Ex: <6 consecutive operations, <6 variables/signals
Complex/Lengthy Numerical Expressions	X		X preferred	Ex: >6 consecutive operations, >6 variables/signals
Numerical Expressions containing continuously valued states	X*			Ex: Difference equations, integrals, derivatives, filters *The actual integrator function can be written in eML
Combination of: •Complex Logic •Simple Numerical Expressions		X		iterating a counter is considered a simple numeric calculation
Combination of: •Simple Logic •Complex Numerical Expressions	X For Logic		X For Math	•Can use only Simulink, only eML or use Simulink for the logic and eML for the math
Combination of •Complex logic •Complex Numerical Expressions		X for Logic	X for Logic and/or Math	•Use Simulink or eML for the numerical calculations •Stateflow should invoke the execution of this subsystem using a function-call
Modal Logic		X		Where the control function to be performed at the current time depends on a combination of past and present logical conditions

As Orion approached the Critical Design Review (CDR) for EFT-1 the GN&C algorithms were well positioned to begin the process of changing them from preliminary functioning requirements to production ready graphical source. Twenty six CSU's were functionally complete, integrated into RAMSES and performing well in Monte Carlo analysis. These were comprised of more than 200 testable units, most of whom had modified complexity metrics that met the Orion SDP standard.

V. Post-CDR Analysis Production

After CDR, GN&C focus shifted from algorithm development to evolving the models to include attributes of good software engineering. This section discusses process and model attributes that affected testing and review of GN&C CSU's.

Most GN&C model based development projects create algorithms that are autocoded from data-flow diagrams and fit into a larger framework that may be developed by hand or "semi-automatically" using UML or other graphical tools. The selection of the "level" of autocode is an important project decision and should be made early in development. The autocode level may be anywhere from autocoding only small units, to autocoding entire rate groups or autocoding the entire GN&C application. For Orion, the GN&C application used autocode that included the CSU model reference block of figure 5 as well as the input and parameter junction boxes. This means that the interface to the UML-developed application code was defined by the the output data buses (structures) of upstream CSUs, as well as the various parameter bus types. Parameter data were collected into CSU-specific parameters, vehicle and physical constants and CSU

1 CSU Requirements

Table 1-1 SRS Requirement Description

CSU Req.	Name	Description / *Rationale*	Parent Req.
DLTRIG.0001	Deploy FBC Parachutes with Velocity Trigger	Prior to FBC jettison the Descent and Landing Triggers (DLTRIG) CSU shall set the FBC Chute Deploy trigger to true when the Nav source flag is Unaided (3) and the velocity is at or below a threshold. / *This velocity trigger is the backup algorithm that enables the chute sequence to start in the event that the navigated altitude is not deemed acceptable.*	GID-354
DLTRIG.0002	Deploy FBC Parachutes with Smart Deploy Trigger	Prior to FBC jettison the DLTRIG CSU shall set the FBC Chute Deploy trigger to true when the Nav source flag is set to GPS-Aided (1) or Baro-Alt (2), the altitude is at or below the FBC Smart Altitude threshold and the computed root sum square (RSS) of the input Pitch and Yaw rates are greater than or equal to the FBC Smart Threshold. / *This trigger allows for the FBC parachutes to be deployed above the normal deployment altitude to prevent the vehicle rates from becoming too excessive.*	GID-354 and GID-180
DLTRIG.0003	Deploy FBC Parachutes with Altitude Trigger	Prior to FBC jettison the Descent and Landing Triggers CSU shall set the FBC Chute Deploy trigger to true when the Nav source flag is set to GPS-Aided (1) or Baro-Alt(2), the altitude is at or below the FBC Jettison Altitude threshold. / *This is the normal deploy altitude and acts as the floor for the entire chute sequence to start.*	GID-354 and GID-180
DLTRIG.0004	Jettison Drogue Parachutes with Unaided Nav Altitude	After the Drogue Parachutes have been deployed but prior to Drogue jettison the DLTRIG CSU shall set the Drogue Jettison Trigger to true based on a timer threshold when the Nav source flag is Unaided (3). / *This timer trigger is the backup algorithm that allows for the Drogue parachutes to be jettisoned if the Nav altitude is not deemed acceptable.*	GID-349
DLTRIG.0005	Jettison Drogue Parachutes with GPS-Aided or Baro-Alt Nav Altitude	After the Drogue Parachutes have been deployed but prior to Drogue jettison the DLTRIG CSU shall set the Drogue Jettison Trigger to true based on a set of algorithms when the Nav source flag is GPS-Aided (1) or Baro-Alt (2). The Drogue Jettison Trigger shall be set to true when all of the following conditions are true: 1) minimum time on Drogues has been exceeded, 2) current altitude is less than or equal to Main Parachute deployment altitude, and 3) Smart Drogue Jettison Trigger command is set. / *This set of algorithms ensures that the vehicle altitude and rates are within an acceptable Main Parachute deployment box.*	GID-349

Figure 11. Example DDR's.

command parameters. These parameters could be modified by the sequencer at activity boundaries (GN&C mode changes).

This "CSU plus Junction Box" interface represented a compromise that allowed the higher level application to function as an object-oriented design and to use efficient hand coded mode-ing logic, while still eliminating the hand coding of GN&C algorithms. The inclusion of junction boxes also meant that most of the CSU-to-CSU interfaces were defined in autocode and carried over from the RAMSES Simulink design environment to the GN&C application. The disadvantage of this methodology was that the domain-level mode-ing logic, CSU parameterization and any additions of new bus types did not carry over from RAMSES to autocode. The Orion experience was that errors due to hand coding the moding and bus-level interfaces were rare, but their development required some time and resources.

In spite of the fact that the autocode was at the CSU + Junction Box level, unit testing of the CSU was done at the inner CSU level. This prevented the CSU designer from needing knowledge of other CSU output data types and made the CSU test drivers independent of other CSU or domain level development. The junction box mapping is inspected and tested as part of integrated testing.

Because a MBD process was used, no detailed, implementation level requirements were needed, and the Simulink diagrams provided a certain amount of insight into each CSU's design. However, the MBD process still required documentation in several areas. First, the Software Requirements Specifications (SRS's) for Orion often did not specify enough detail to drive unit testing of each CSU, so Derived Design Requirements (DDR's) were created and documented in a "CSU memo" for each CSU (see examples in Figure 11). The CSU memo also included several other sections that documented important design information. The major CSU document sections were:

1. Derived Design Requirements – "Shall" statements used to drive unit testing with parent requirements in the Orion SRS
2. Design and Theory – which provided the mathematical and logic formulation for GN&C algorithms
3. CSU Interface – included automatically generated tables of inputs, outputs and parameters created from CSU interface bus types
4. Parameter Configuration Set Design – provided information about how to set configurable parameters
5. Assumptions and Limitations
6. Implementation Reference – link to the CSU model in html form
7. Unit Test Descriptions

On the Orion project, the Simulink CSU diagrams are treated as the source for CSU algorithms. For this reason, no formal inspections of the autocode were performed. However, formal inspections of the CSU model diagrams were rigorously conducted and these included reviews of the autocode for efficiency. Also, all unit tests that are developed and executed on the model have been executed on the autocode.

Model Maturation. The model maturation process was well defined (see Figure 12). This process was adopted from Honeywell. It gave clear steps for how to develop a model in a way that it would be testable, standards compliant, and ready for inspection.

To aid the developers when designing CSUs a "Developer Checklist" was created. This had a list of major items that needed to be "checked" off at each stage in the process (CSU Inspection readiness,

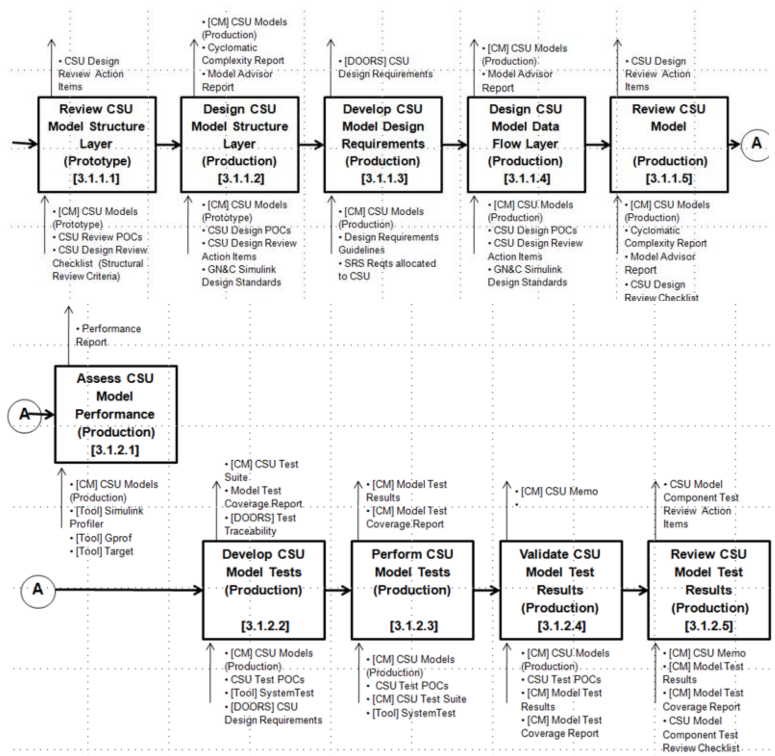


Figure 12. Model Maturation Process.

CSU Unit Test readiness, etc.). At first, many developers were unsure if their models were mature enough to even integrate into the EBA. This checklist gave them a better picture of the maturity of the models and helped increase the quality of the models that were being reviewed.

Some examples of this development checklist entrance and exit criteria for the Model Test Review include:

- Are all low level requirements satisfied
- Is the code/model properly notes/commented
- Is the model broken up into individually testable units
- Is the complexity of each testable unit below 20
- Does the model pass all of the Model Advisor checks
- Do the existing Unit Test achieve 100% model coverage
- Is the model "autocod-able" (does the autocoding process complete and compile successfully)

Testable Units. As familiarity with the autocoder increased, the need to define a "testable units" became more important. The goal was to have a one-to-one match between the source Simulink Model, Stateflow Chart, or Embedded MATLAB function and the resulting CPP function or method. This one-to-one match helped tracking of testing between the source "models" and the generated functions. It also helped avoid re-testing code due to excessive in-lining or function duplication.

Figure 12. External eML Function Called by Two Parent Models.

One of the biggest headaches of testing the autocode from our models was how inconsistent the autocoder is with various model sources. An independently testable unit at the modeling level does not always translate into and individually testable unit in the autocode. Some source blocks are in-lined in the parent model's code, whereas others are separate functions wither in the parent cpp file or in the shared utilities section.

To partially get around this issue, autocode "directives" were used in eML functions (e.g. %eml.inline("never")) and block settings to force the autocoder to create individual methods/functions from the source eml/Statelfow function. This solved some issues but did not result in the ideal 1-to-1 situation because autocoded eml/Stateflow functions were coded as methods of the parent model. This caused issues with testing the method directly and eliminated sharing these functions between models. Figure 13 illustrates a single external eML function called by 2 separate Models and the placement of the autocode that represents the eML function. All of this duplication of code places a heavier burden on the backend testing of the actual autocode.

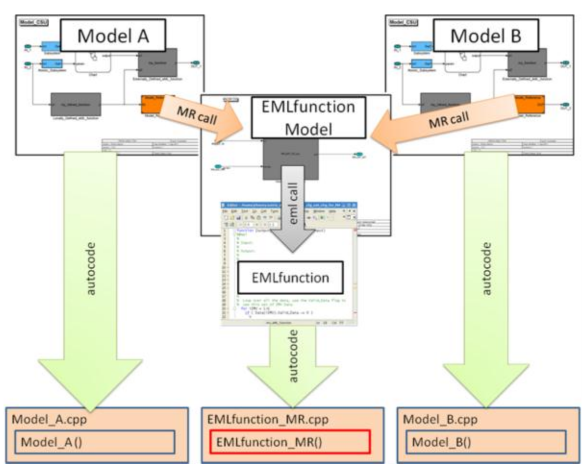


Figure 13. Model Reference Blocks as Testable Unit Wrappers.

The only way to ensure absolute 1-to-1 model to code was to use "Model References" (Figure 13). If each source function were a Model Reference, it could be called throughout the project and result in a single instance in the autocode. Another advantage of the MBR approach was that each MBR could be directly tested in a Simulink driver as a unit – and the resultant test drivers functioned well with the Mathworks "System Test" verification tool. Figure 14 illustrates how a model reference containing an eML function is placed in the autocode.

However, many of our functions were created using eML and to a smaller degree, Stateflow. Eml functions cannot call Simulink Models (aka Model References) – so if an algorithm needed to be written in eML, none of the shared Simulink or Stateflow models could be called. Figure 14 illustrates the calling ability between the Simulink, Stateflow, and eML tools. From this you can see that eML is restricted to only calling other eML functions.

Due to this issue, strong limitations were placed on the use of eML functions that were shared between models. This caused problems with developers that favored eML for their algorithm development.

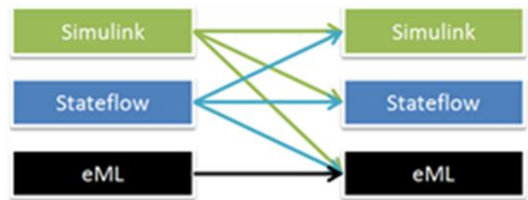
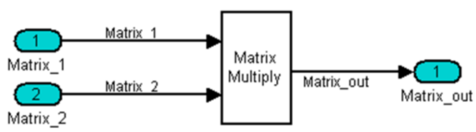


Figure 13. Legal Execution Hierarchy.

Develop meaningful complexity requirements for MBD. During early Orion development, functionality and capability were given priority over modularity, complexity and other software engineering considerations. This was due partly to the need to use GN&C algorithms early to perform time domain analysis for vehicle systems integration and due to the fact that complexity definitions and guidelines were evolving. The Orion project Software Development plan (SDP) required that the cyclomatic complexity (CC) (Ref. 2) of functional units be no more than 20 (hand code and autocode). It was soon discovered, however that automatically generated code often had higher CC than hand code and testable units in the model didn't translate into testable units in the autocode (as discussed above).

The Cyclomatic Complexity metric is a very useful way to measure the testability of the code. However, it posed a bit of a challenge for functions that were developed in Simulink because there is no direct way to find the CC of the code without actually autocoding the model and analyzing the autocode. Simulink provides the ability to find the Model Complexity, but on Orion, the SDP requirement is applied to the autocode only, regardless of model source. Also, the standard way of measuring the CC of the autocode did not reflect the testability of the source model. This disconnect is mainly due to the liberal use of for loops in the autocode for data initialization, data assignment, and vector/matrix math. The standard way of calculating CC counts each for loop as an additional unit of complexity whether the loop is static or not. This issue caused the Model Complexity and the Code Complexity to differ greatly and added confusion to the developer. Figure 15 illustrates how a single block in the model is translated into code that has a complexity of 3 due to the nested static for loops arising from a matrix multiplication. These additional static for loops do not add to the complexity of the code.

Matrix Multiply block of Model Complexity 0:



Resulting Autocode of Complexity 3:

```
...
for (i = 0; i < 3; i++) {
    For (i_0 = 0; i_0 < 3; i_0++) {
      Example_Y.C[i + 3 * i_0] = 0.0;
      For (i_1 = 0; i_1 < 3; i_1++) {
        Example_Y.C[i + 3 * i_0] = Example_U.A[3 * i_1 + i] * Example_U.B[3 *
          i_0 + i_1] + Example_Y.C[3 * i_0 + i];
      }
    }
  }
...
```

Figure 14. Effect of Autocoding on Complexity.

Ensuring that the resulting autocode for each function would have a CC value of <20 is difficult. The team found that if static for-loops are ignored, the CC for autocoded functions closely matched the source model complexity. Since these static for loops did not truly add complexity to the code, it was agreed that our CC metric should exclude them. To measure the modified CC, a tool was developed that calculated complexity and ignored static for loops. The CC limits were reviewed in both the Model Inspection and Unit Test Inspection.

In the 2010b version of Simulink, the complexity of a model is not easy to obtain and caused issues in metric reporting, this issue has been improved in future versions.

Other Metrics. Using SLOC is a good way to calculate project metrics such as project size and for estimating work load for future tasks. However, with MBD, model size is a better metric than lines of code. The SLOC count of the autocode was not consistently proportional to the size of the Model for many reasons. Instead, a "Model Size" metric was created to take into account components from all three tools (Simulink/Stateflow/eML) and is calculated via the following formula (calculated automatically via script):

Model Size = Simulink Blocks + lines of eml code + Stateflow Transitions

The ORION project did not have a SLOC limit requirement for functions or files since function complexity was governed by Cyclomatic Complexity requirements. But, the team found this tool useful in tracking progress and project size.

Model coverage vs Code coverage. Ideally, to minimize the testing difference between Models and Code, the coverage of the model should be the same as the code. This is not always the case. In addition to the code reuse issues described above, the autocoder will optimize out certain blocks, remove unreachable paths (dead code) based on specific use, and insert protection around certain operations (e.g. integer overflow). Also, some blocks that may have internal paths, are not represented as such in the source model. In Simulink, a block is either executed or is not executed; internal branching is not revealed until the block is autocoded. Some of these discrepancies between the model and autocode can be prevented by changing options in the configuration setting and some could be removed by modifying block settings. However, 1-to-1 model and code coverage was not 100%.

Unit Testing. Simulink System Test (Figure 16) was chosen as the official Orion GN&C tool for Unit Testing the models. Developing the tests from within the System Test tool proved troublesome due to both stability issues and limited functionality. For example, error reporting for simulations was limited to either pass or fail, with no insight to the cause of problems. As the team grew more accustomed to the tool, we found workarounds for many of the issues. A Standards and Guidelines document was created for unit testing as well. This document listed the standard formats, process, and APIs for developing unit tests for models. This document help standardize the unit tests across the entire GN&C project.

Due to the general issues of System Test, our unit testing framework was designed to rely as little as possible on System Test itself. All of the input data, initialization routines, and comparison data were created outside of the tool. System Test was basically used to execute the test, return results, and generate the test report and coverage files. The quality of the test and coverage reports were a highlight of the tool. Due to the independent way our unit tests were developed, switching to a more capable tool in the future is expected to be fairly seamless.

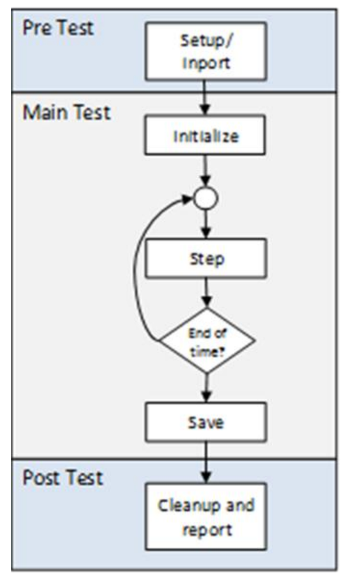


Figure 15. System Test Execution Flow.

LDRA and SIL/PIL Mode integration. The official Code analysis tool on Orion was LDRA (Liverpool Data Research Associates) Testbed. LDRA Testbed provides the core static and dynamic analysis engines for our software. By default, The MathWorks tools only worked with Bullseye Coverage tool with the C target. The GN&C FSW team worked with Mathworks support to achieve compatibility with LDRA for generating unit test scripts that could be run with LDRA.

One highly useful feature of the Simulink tool is the ability to run the autocode of a model in "SIL" or "PIL" mode. This allows the developer to generate unit tests with the source models, then run the same tests on that actual compiled FSW (SIL mode) and on the actual FSW running in an emulated target environment (PIL mode). Unfortunately, our pioneering use of the C++ Encapsulated Target and our use of the LDRA Tools, the SIL and PIL mode were not yet compatible "out of the box". Upon request from Orion, The Mathworks provided a patch to enable this functionality. Later versions of Simulink have LDRA compatibility built in and support SIL and PIL mode for the C++ (encapsulated) target.

VI. Conclusions

The Orion GN&C team has successfully used a MBD process to generate software for GN&C algorithms for the EFT-1 mission. The team incurred considerable up-front cost for the transition to an MBD process, but downstream benefits are now being realized. Some of the upfront costs were unavoidable, but others may be avoided in future programs by heeding the lessons learned enumerated here. Some of the costs included:

- A steep learning curve for engineers not familiar with MBD tools
- Slow and complex development tools and processes
- Configuration management issues.

These issues will be mitigated in algorithm development and test for future Orion missions since many of the tools and techniques have been established. The team now has mature MBD coding standards and automated standards checking via the Model Advisor. Faster build and autocode times have arisen from improvements in Mathworks tools and better application of the tools. And the team has a better understanding of how to perform configuration management with MBD artifacts as described above.

Some of the benefits that GN&C is now observing include:

- No schedule time was needed for hand coding GN&C algorithms (60,000+ SLOC were autocoded by CDR)
- Detailed requirements review was replaced by review of MBD artifacts which had proven functionality
- Automated test framework and report generation has simplified testing and production of test artifacts
- Automated standards checking tools (e.g. Model Advisor) and graphical artifacts have facilitated the inspection process

The Orion GN&C team is ready to complete GN&C software integration for the EFT-1 mission and to move forward to generate GN&C algorithms for other flight phases. Other programs desiring to use an MBD process should not start from scratch. Many tools, techniques and lessons learned are available from the authors and other Orion GN&C team members.

Acknowledgments

To acknowledge all the contributors to the Orion MBD process is not possible since it was a product of a large, diverse team as described above. However, the authors wish to acknowledge certain key individuals who made particularly large contributions, including Scott Tamblyn from NASA, David Shoemaker from Lockheed Martin, David Oelschlaeger and Kevin Morrill from Honeywell and Ian Mitchell from Draper Lab.

References

[1]Trick Simulation Environement, Trick User's Guide, Version 2007.20, Automation, Robotics and Simulation Division, NASA Johnson Space Center, November, 2009.

[2]McCabe, T., "A Complexity Measure," IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-2, NO.4, pp 308-320, December, 1976.